# MIPS32® 34K™ Processor Core Family Software User's Manual

**Document Number: MD00534**
**Revision 01.02**
**December 19, 2006**

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

Template: nB1.01, Built with tags: 2B MIPS32 PROC

# Table of Contents

# List of Figures

# List of Tables

*Chapter 1*

# Introduction to the MIPS32® 34K™ Processor Core Family

The 34K™ core from MIPS Technologies is a high-performance, low-power, 32-bit MIPS®RISC processor core family intended for custom system-on-silicon applications. The core is designed for semiconductor manufacturing companies, ASIC developers, and system OEMs who want to rapidly integrate their own custom logic and peripherals with a high-performance RISC processor. A 34K core is fully synthesizable to allow maximum flexibility; it is highly portable across processes and can easily be integrated into full system-on-silicon designs. This allows developers to focus their attention on end-user specific characteristics of their product.

The 34K core is ideally positioned to support new products for emerging segments of the digital consumer, network, systems, and information management markets, enabling new tailored solutions for embedded applications.

The 34K family has four members: the MIPS32® 34Kc™ core, the MIPS32 34Kc Pro™ core, the MIPS32 34Kf™ core, and the MIPS32 34Kf Pro™ core

- The 34Kc is a 32-bit RISC core for high performance applications
- The 34Kf core adds an IEEE-754 compliant floating point unit
- The 34Kc Pro core offers the CorExtend™ capability
- The 34Kf Pro core has both the floating point unit and the CorExtend capability

The term *34K core* as used in this document, generally refers to all cores in the 34K family. When referring to characteristics unique to an individual family member, the specific core type will be identified.

On a 34K core, instruction and data caches are configurable at 0, 8, 16, 32, or 64 KB in size. Each cache is organized as 4-way set associative. The data cache features non-blocking load misses. On a cache miss, the processor can continue executing instructions until a dependent instruction is reached. Both caches are virtually indexed and physically tagged. Virtual indexing allows the cache to be indexed in the same clock in which the address is generated rather than waiting for the virtual-to-physical address translation in the TLB.

The core implements the MIPS32 Release 2 Instruction Set Architecture (ISA) as well as the MIPS16e™ Application Specific Extension (ASE) for code compression. The core also implements the MIPS MT Application Specific Extension. This ASE defines the architectural state and new instructions that allow multithreading on a MIPS core.

The MMU of the 34K core may be TLB based or a simple fixed mapping translation mechanism. If TLB-based, micro TLBs in the fetch and load/store units cache the latest address translations from the larger joint TLB.

The Multiply-Divide Unit (MDU) is fully pipelined and supports a maximum issue rate of one 32x32 multiply (MUL/MULT/MULTU), multiply-add (MADD/MADDU), or multiply-subtract (MSUB/MSUBU) operation per clock.

The basic Enhanced JTAG (EJTAG) features provide CPU run control with stop, single stepping and re-start, and with software breakpoints through the SDBBP instruction. Support for connection to an external EJTAG probe through the

Test Access Port (TAP) is also included. Instruction and data virtual address hardware breakpoints as well as the MIPS Trace mechanism can be optionally included.

The bus interface implements the Open Core Protocol (OCP), with 64-bit read and write data buses. The bus interface may operate at the same or a lower clock rate than the core itself.

The rest of this chapter provides an overview of the MIPS32 34K processor core and consists of the following sections:


- Section 1.1 "34K™ Core Features"
- Section 1.2 "34K™ Core Block Diagram"


## 1.1 34K™ Core Features

- 9-stage pipeline
- 32-bit Address Paths
- 64-bit Data Paths to Caches
- MIPS32-Compatible Instruction Set
  - Multiply-add and multiply-subtract instructions (MADD, MADDU, MSUB, MSUBU)
  - Targeted multiply instruction (MUL)
  - Zero and one detect instructions (CLZ, CLO)
  - Wait instruction (WAIT)
  - Conditional move instructions (MOVZ, MOVN)
  - Prefetch instruction (PREF)
- MIPS32 Enhanced Architecture (Release 2) Features
  - Vectored interrupts and support for an external interrupt controller
  - Programmable exception vector base
  - Atomic interrupt enable/disable
  - GPR shadow sets
  - Bit field manipulation instructions
- MIPS16e Application Specific Extension
  - 16 bit encodings of 32-bit instructions to improve code density
  - Special PC-relative instructions for efficient loading of addresses and constants

    - Data type conversion instructions (ZEB, SEB, ZEH, SEH)
    - Compact jumps (JRC, JALRC)
    - Stack frame set-up and tear down “macro” instructions (SAVE and RESTORE)
- MIPS MT Application Specific Extension (ASE)
    - Support for 1 or 2 Virtual Processing Elements (VPEs)
    - Support for 1-9 Thread Contexts (TCs)
    - Inter-Thread Communication (ITC) memory for efficient communication & data transfer.
- Programmable L1 Cache Sizes
    - Individually configurable instruction and data caches
    - Sizes of 0, 8, 16, 32, or 64 KB.
    - 4-Way set associative
    - Up to 4 non-blocking loads
    - Supports Write-back with write-allocation and Write-through without write-allocation
    - 256-bit (32-byte) cache line size, doubleword sectored - suitable for standard 64-bit wide single-port SRAM
    - Virtually indexed, physically tagged
    - Cache line locking support
    - Non-blocking prefetches
- Data and Instruction ScratchPad RAMs
    - Separate RAMs for Instruction and Data
    - addressable up to 1MB
    - 64-bit OCP interfaces for external access
- R4000 Style Privileged Resource Architecture
    - Count/compare registers for real-time timer interrupts
    - Instruction and data watch registers for software breakpoints
- Standard Memory Management Unit
    - 16/32/64 dual-entry MIPS32-style JTLB per VPE with variable page sizes
    - JTLBs are sharable under software control

    - 4-9 entry instruction TLB
    - 8 entry data TLB
- Optional Memory Management Unit
    - Simple Fixed Mapping Translation (FMT)
    - Address spaces mapped using register bits
- OCP Bus Interface Unit (BIU)
    - 32b address and 64b data
    - Core/bus ratios of 1, 1.5, 2, 2.5, 3, 3.5, 4, 5 and 10 are supported
    - Supports bursts of 4x64b
    - 4 entry write buffer - handles eviction data, write-through, uncached, and uncached accelerated store data
    - Simple Byte enable mode allows easier bridging to other bus standards
    - Extensions for management of front side L2 cache
- CorExtend™ User Defined Instruction capability (34Kc Pro and 34Kf Pro)
    - Optional support for the CorExtend feature allows users to define and add instructions to the core (as a build-time option)
    - Single or multi-cycle instructions
    - Source operations from register, immediate field, or local state
    - Destination to a register or local state
    - Interface to multiply-divide unit, allowing sharing of accumulation registers
- Multiply-Divide Unit
    - Maximum issue rate of one 32x32 multiply per clock
    - Early-in divide control. Minimum 11, maximum 34 clock latency on divide
- Floating Point Unit (34Kf and 34Kf Pro only)
    - IEEE-754 compliant floating point unit
    - Compliant to MIPS 64b FPU standards
    - Supports single and double precision datatypes
- Coprocessor2 Interface

    - 64-bit interface to user designed coprocessor
- Power Control
    - No minimum frequency
    - Power-down mode (triggered by WAIT instruction)
    - Support for software-controlled clock divider
    - Support for extensive use of fine-grain clock gating
- EJTAG Debug Support
    - CPU control with start, stop and single stepping
    - Software breakpoints via the SDBBP instruction
    - Optional hardware breakpoints on virtual addresses; 0 or 4 instruction and 0 or 2 data breakpoints per VPE
    - Test Access Port (TAP) facilitates high speed download of application code
    - Optional MIPS Trace hardware to enable real-time tracing of executed code

## 1.2 34K™ Core Block Diagram

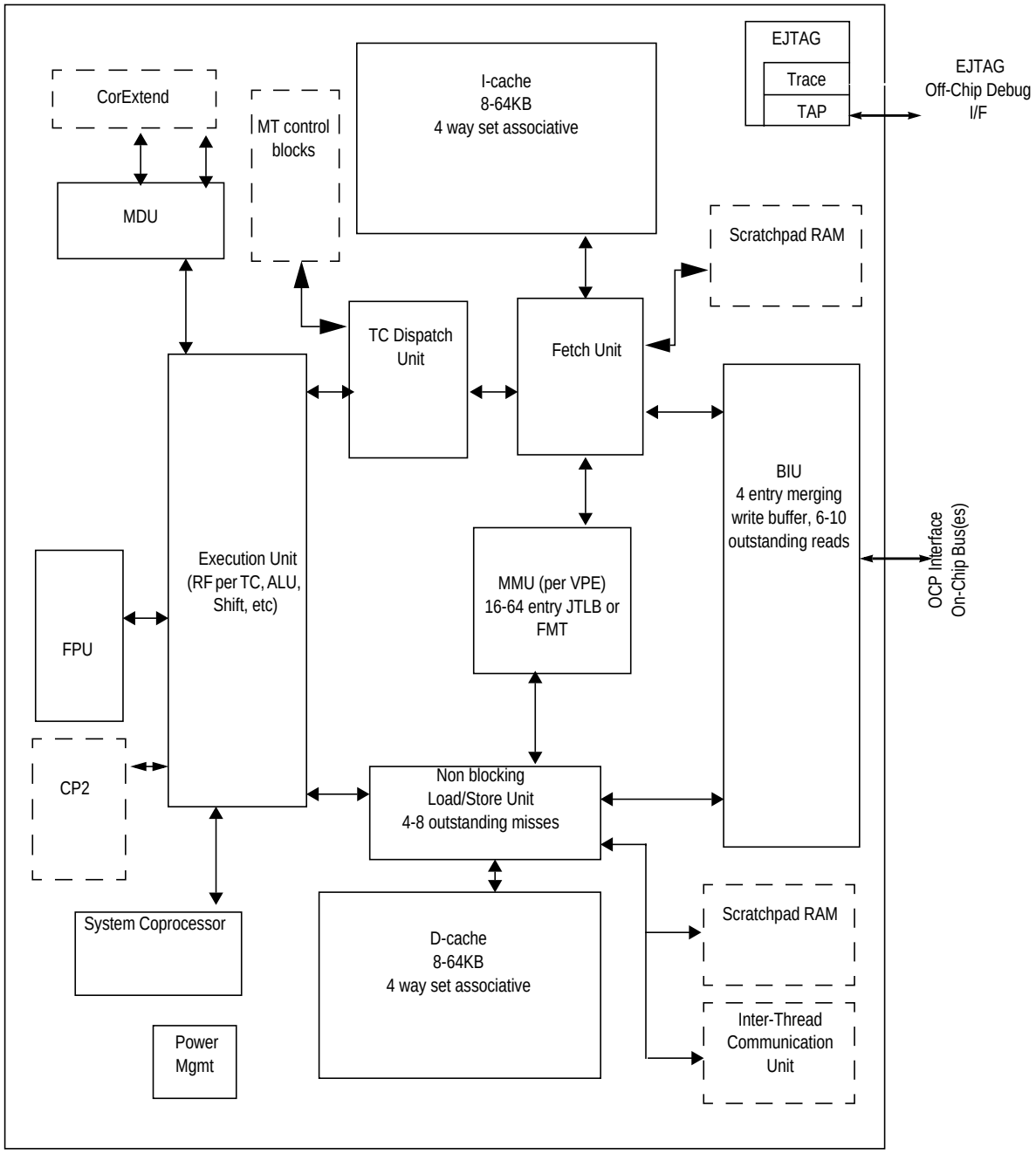
The 34K core contains a number of blocks, as shown in the block diagram in Figure 1.1. The major blocks are as follows:

- Execution Unit (ALU)
- Multiply-Divide Unit (MDU)
- System Control Coprocessor (CP0)
- Memory Management Unit (MMU)
- Floating Point Unit (FPU) - only in 34Kf
- Cache Controller
- Bus Interface Unit (BIU)
- Power Management
- MIPS16e support
- Instruction Cache (I-cache)
- Data Cache (D-cache)
- Enhanced JTAG (EJTAG) Controller

- CorExtend™ User Defined Instructions (UDI)

Figure 1.1 shows a block diagram of a 34K core.The MMU can be implemented using either a translation lookaside buffer or a fixed mapping (FMT). Refer to Chapter 5, “Memory Management of the 34K™ Core” for more information.

**Figure 1.1 34K™ Processor Core Block Diagram**



## 1.2.1 Logic Blocks

The following subsections describe the various logic blocks of the 34K processor core.

### 1.2.1.1 Execution Unit

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

The execution unit includes:

- 32-bit adder used for calculating the data address
- Logic for branch determination and branch target address calculation
- Bypass multiplexers used to avoid stalls when executing instruction streams where data-producing instructions are followed closely by consumers of their results
- Zero/One detect unit for implementing the CLZ and CLO instructions
- ALU for performing bitwise logical operations
- Shifter and Store aligner
- Floating Point Unit Interface
- Coprocessor2 Interface

### 1.2.1.2 Multiply/Divide Unit (MDU)

The Multiply/Divide unit performs multiply and divide operations.The MDU consists of a pipelined 32x32 multiplier, result-accumulation registers (HI and LO), multiply and divide state machines, and all multiplexers and control logic required to perform these functions. This pipelined MDU supports execution of a multiply or multiply-accumulate operation every clock cycle. Unlike previous cores, there is no dependence between operand size and issue rate for multiplies. Divide operations are implemented with a simple 1 bit per clock iterative algorithm and require 35 clock cycles in worst case to complete. Early-in to the algorithm detects sign extension of the dividend, if it is actual size is 24, 16 or 8 bit. the divider will skip 7, 15 or 23 of the 32 iterations. An attempt to issue a subsequent MDU instruction while a divide is still active causes a pipeline stall until the divide operation is completed.

On Pro Series cores, the MDU accumulator is shared with the CorExtend block. Many CorExtend instruction types can make use of the HI/LO accumulation registers.

### 1.2.1.3 System Control Coprocessor (CP0)

In the MIPS architecture, CP0 is responsible for the virtual-to-physical address translation, cache protocols, the exception control system, the processor's diagnostics capability, operating mode selection (kernel vs. user mode), and the enabling/disabling of interrupts. Configuration information such as cache size, set associativity, and presence of build-time options are available by accessing the CP0 registers. Refer to Chapter 7, "CP0 Registers of the 34K™ Core" for more information on the CP0 registers. Refer to Chapter 11, "EJTAG Debug Support in the 34K™ Core"

for more information on EJTAG debug registers. Most of CP0 is replicated per VPE. A small amount of state is replicated per TC and some is shared between the VPEs.

#### 1.2.1.4 Memory Management Unit (MMU)

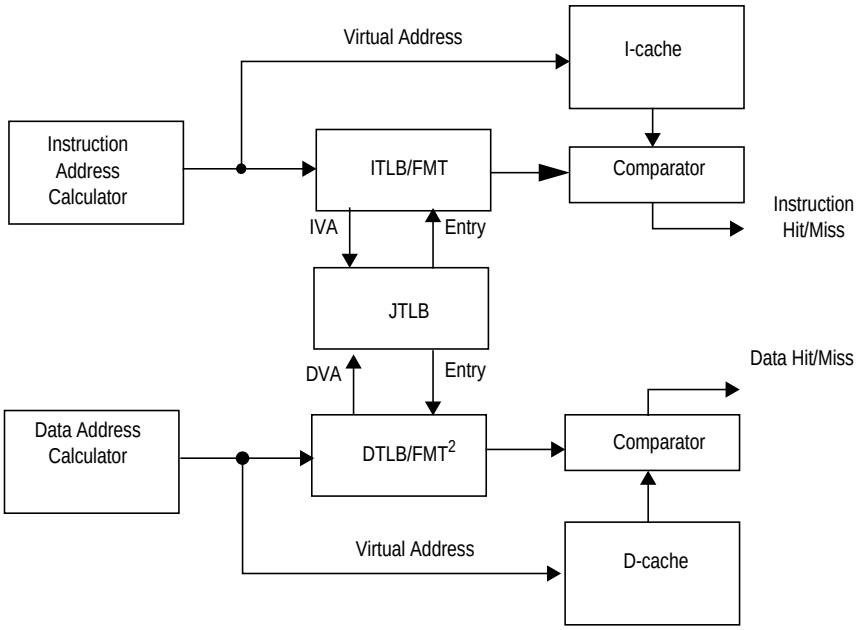
The 34K core contains an MMU per VPE that interfaces between the execution unit and the cache controllers, shown in Figure 1.2.Although the 34K core implements a 32-bit architecture, the Memory Management Unit (MMU) is modeled after the MMU found in the 64-bit R4000 family, as defined by the MIPS32 architecture.

By default, the 34K core each MMU is based on a Translation Lookaside Buffer (TLB). The TLB consists of three or four translation buffers: a configurable 16/32/64 dual-entry fully associative Joint TLB (JTLB) per VPE, a 4-9 entry fully associative Instruction TLB (ITLB) and a 8-entry fully associative data TLB (DTLB). The ITLB and DTLB, also referred to as the micro TLBs, are managed by the hardware and are not software visible. The micro TLBs contain subsets of the JTLB. When translating addresses, the corresponding micro TLB (I or D) is accessed first. If there is not a matching entry, the JTLB is used to translate the address and refill the micro TLB. If the entry is not found in the JTLB, then an exception is taken.

The core optionally implements a FMT-based MMU instead of a TLB-based MMU. The FMT replaces the ITLB and DTLB and the JTLB is removed. The FMT performs a simple translation to get the physical address from the virtual address. Refer to Chapter 5, “Memory Management of the 34K™ Core” for more information on the FMT.

Figure 1.2 shows how the address translation mechanism interacts with cache accesses.

**Figure 1.2 Address Translation During a Cache Access**




#### 1.2.1.5 Fetch Unit

The fetch unit is responsible for providing instructions to the execution unit for all TCs. The fetch unit includes:

- control logic for the instruction cache
- MIPS16e instruction recoder

- Dynamic branch prediction
  - 512 entry bimodal branch history table for predicting conditional branches
  - 4 entry return prediction stack for predicting return addresses
- 8 entry instruction buffer per TC to decouple the fetch and execution pipelines
- Interface to Instruction ScratchPad RAM

When executing instructions from multiple TCs, a portion of the IBF is used as a skid buffer. Instructions are held in the IBF after being sent to the execution unit. This allows stalled instructions to be flushed from the execution pipeline without needing to be refetched.

#### 1.2.1.6 Thread Schedule Unit (TSU)

This unit is responsible for dispatching instructions from different Thread Contexts (TCs). An external policy manager assigns priorities for each TC. The TSU determines which TCs are runnable and selects the highest priority one available. If multiple are available, a round-robin mechanism will select between them fairly.

The policy manager is a customer configurable block. Simple round-robin or fixed priority policies can be implemented by tying off signals on the interface. A reference policy manager is also included that implements a weighted round-robin algorithm for long-term distribution of execution bandwidth.

#### 1.2.1.7 Instruction Cache

The instruction cache is an on-chip memory array of up to 64 KB. The cache is virtually indexed and physically tagged, allowing the virtual-to-physical address translation to occur in parallel with the cache access rather than having to wait for the physical address translation. The tag holds 20 or 21 bits of the physical address, a valid bit, a lock bit, and optionally a parity bit. There is a separate 6b array which holds data for all 4 ways to be used in the Least Recently Used (LRU) replacement scheme. Some precode information is included in the instruction cache data array. An additional 6b per pair of 32b instructions is used to enable quick detection of branches and jumps in the fetch unit. If parity is implemented, a single bit covers the 6b recode and 8b cover the 64b data.

The core supports instruction cache locking. Cache locking allows critical code to be locked into the cache on a “per-line” basis, enabling the system designer to maximize the efficiency of the system cache. Cache locking is always available on all instruction cache entries. Entries can be marked as locked or unlocked (by setting or clearing the lock bit) on a per-entry basis using the CACHE instruction.

The LRU array must be bit-writable. The tag and data arrays only need to be word-writable.

#### 1.2.1.8 Load/Store Unit

The Load/Store Unit is responsible for data loads and stores. It includes:

- Data cache control logic
- 4-8 line fill/store buffer
- ScratchPad RAM interface

#### 1.2.1.9 Data Cache

The data cache is an on-chip memory array of up to 64 KB. The cache is virtually indexed and physically tagged, allowing the virtual-to-physical address translation to occur in parallel with the cache access. The tag holds 20 or 21 bits of the physical address, a valid bit, a lock bit, and optionally a parity bit. A separate array holds the LRU bits (6b), dirty bits (4b), and optionally, dirty parity bits (4b) for all 4 ways. The data array is optionally parity protected with 1b per 8b of data.

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

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

The physical data cache memory must be byte writable to support sub-word store operations. The LRU/dirty bit array must be bit-writable.

#### 1.2.1.10 Bus Interface Unit (BIU)

The Bus Interface Unit (BIU) controls the external interface signals. Additionally, it contains the implementation of a collapsing write buffer. This buffer is used to merge Write-Through transactions as well as gathering multiple writes together from dirty line evictions and uncached accelerated stores. The write buffer consists of 4 32B entries.

#### 1.2.1.11 Power Management

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

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

- Register-controlled power management
- Instruction-controlled power management

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

Refer to Chapter 10, “Power Management in the 34K™ Core” for more information on power management.

#### 1.2.1.12 MIPS16e™ Application Specific Extension

The 34K core includes support for the MIPS16e ASE. This ASE improves code density through the use of 16-bit encodings of MIPS32 instructions plus some MIPS16e-specific instructions. PC relative loads allow quick access to constants. Save/Restore macro instructions provide for single instruction stack frame setup/teardown for efficient sub-routine entry/exit. Sign- and zero-extend instructions improve handling of 8bit and 16bit datatypes.

A decompressor converts the MIPS16e 16-bit instructions fetched from the instruction cache or external interface back into 32-bit instructions for execution by the core.

#### 1.2.1.13 EJTAG Debug

All cores provide basic EJTAG support with debug mode, run control, single step and software breakpoint instruction (SDBBP) as part of the core. These features allow for the basic software debug of user and kernel code. A TAP controller is also included, enabling communication between an EJTAG probe and the CPU through a dedicated port. This provides the possibility for debugging without debug code in the application, and for download of application code to the system.

An optional EJTAG feature is hardware breakpoints. A 34K core may have four instruction breakpoints and two data breakpoints per VPE, or no breakpoints. The hardware instruction breakpoints can be configured to generate a debug exception when an instruction is executed anywhere in the virtual address space. Bit mask and Address Space Identifier (ASID) values may apply in the address compare. These breakpoints are not limited to code in RAM like the software instruction breakpoint (SDBBP). The data breakpoints can be configured to generate a debug exception on a data transaction. The data transaction may be qualified with both virtual address, data value, size and load/store transaction type. Bit mask and ASID values may apply in the address compare, and byte mask may apply in the value compare.

Another optional debug feature is support for MIPS Trace that enables real-time tracing capability. The trace information can be stored to either an on-chip trace memory or an off-chip trace probe. The trace of program flow is highly flexible and can include the instruction program counter as well as data addresses and data values. The trace features can provide a powerful software debugging mechanism.

Refer to Chapter 11, “EJTAG Debug Support in the 34K™ Core” for more information on the EJTAG features.

#### 1.2.1.14 CorExtend™ User Defined Instructions

This optional module contains support for CorExtend user defined instructions. These instructions must be defined at build-time for the 34K core. The CorExtend feature is a capability of the 34Kc Pro and 34Kf Pro cores. This feature makes 16 instructions in the opcode map available for customer usage, and each instruction can have single or multi-cycle latency. A CorExtend instruction can operate on any one or two general-purpose registers or immediate data contained within the instruction, and can write the result of each instruction back to a general purpose register or a local register. Implementation details for CorExtend can be found in the *34K™ Pro Series™ CorExtend™ Implementor’s Guide* (MD00348)

Refer to Section Table 15.5 “MIPS32 SPECIAL2 Encoding of Function Field” for a specification of the opcode map available for user defined instructions.

*Chapter 2*

# Pipeline of the 34K™ Core

The 34K processor core implements a 9-stage pipeline. The pipeline allows the processor to achieve high frequency while minimizing device complexity, reducing both cost and power consumption. This chapter contains the following sections:



## 2.1 Pipeline Stages

The pipeline consists of nine stages:

- IF - Instruction fetch First
- IS - Instruction fetch Second
- IR - Instruction recode (MIPS16e only)
- IK - Instruction kill (MIPS16e only)
- IT - Instruction fetch Third
- RF - Register File
- AG - Address Generation
- EX - EXecute
- MS - Memory Second

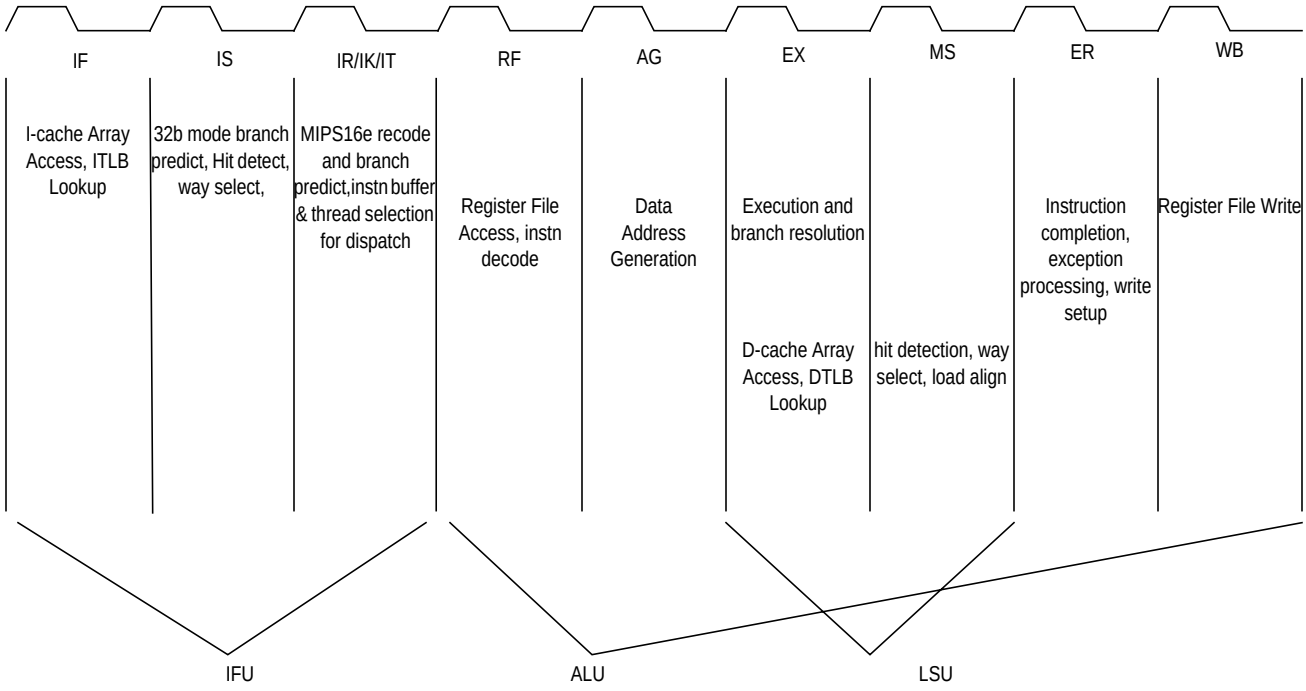
- ER - Exception Resolution
- WB - WriteBack

Two additional stages are conditionally added to the fetch pipeline after the IS stage when executing MIPS16e code. The IR and IK stages are generally bypassed while executing 32-bit code. Unlike the 24K core, the 34K core always has the IT stage and is never bypassed which makes it a 9-stage pipe.

A 34K core implements a bypass mechanism that allows the result of an operation to be sent 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 basic pipeline organization. The various parts of the pipeline are described in more detail in this chapter.

**Figure 2.1 34K™ Core Pipeline Stages**




## 2.1.1 IF Stage: Instruction Fetch First

- I-cache tag/data arrays accessed
- Branch History Table accessed
- ITLB address translation performed
- EJTAG break/watch compares done

### 2.1.2 IS - Instruction Fetch Second

- Detect I-cache hit
- Way select
- MIPS32 Branch prediction

### 2.1.3 IR - Instruction Recode (MIPS16e only)

- MIPS16 recode
- MIPS16 branch prediction
- Stage is bypassed when executing MIPS32 code

### 2.1.4 IK - Instruction Kill (MIPS16e only)

- Kill MIPS16 instructions (due to branches as an example)
- Stage is bypassed when executing MIPS32 code

### 2.1.5 IT - Instruction Fetch Third

- Instruction Buffer
- Branch target calculation
- Thread selection for dispatch based on policy manager

### 2.1.6 RF - Register File Access

- Register File access
- Instruction decoding/dispatch logic
- Bypass muxes

### 2.1.7 AG - Address Generation

- D-cache Address Generation
- Bypass muxes

### 2.1.8 EX - Execute/Memory Access

- Skewed ALU
- DTLB
- Start DCache access

- Branch Resolution

### 2.1.9 MS - Memory Access Second

- Complete DCache access
- DCache hit detection
- Way select mux
- Load align

### 2.1.10 ER- Exception Resolution

- Instruction completion
- Register file write setup
- Exception processing

### 2.1.11 WB - Writeback

- Register file writeback occurs on rising edge of this cycle

## 2.2 Instruction Fetch

The IFU is responsible for supplying instructions to the execution units and handling the results of all control transfer instructions (branches, jumps, etc.). The IFU operation encompasses five pipe stages: IF (**I**nstruction fetch **F**irst), IS (**I**nstruction fetch **S**econd), IR (Instruction Recode), IK (Instruction Kill) and IT (**I**nstruction fetch **T**hird). The instruction cache tags and data are accessed in IF, and the hit determination and the first part of the 32-bit mode target calculation is done in IS. The IR and IK stage handle MIPS16e recoding. The remainder of the 32-bit mode target calculation as well as instruction buffering to the ALUis done in the IT stage. This instruction buffering decouples the IFU from the rest of the pipeline, allowing fetches to proceed even if the processor execution is stalled for some reason. The fetch pipeline and cache bandwidth is 64 bits, supplying up to two instructions per cycle in MIPS32 mode, which allows the IFU to get ahead of the ALU and shields the execution pipeline from some IFU miss penalties.

On the very front of the pipe, before the IF stage there is a mux to select the TC that will start a fetch. Potentially in every cycle a different TC can be fetched assuming that there is more than one TC that is fetchable. Based on the choice of the TC, its predicted program counter value is chosen as the next instruction fetch address.

In addition, the instruction buffers are replicated per TC so that each TC can continue to fetch as and when it gets an opportunity to be fetched and stores its instructions in its instruction buffer for dispatch. For example, when one of the TCs has an instruction cache miss, the other threads can keep fetching as long as the they hit in the cache. When there are two outstanding cache misses, a TC with a third miss would be blocked from being fetched but TCs that hit in the instruction cache could continue being fetched.

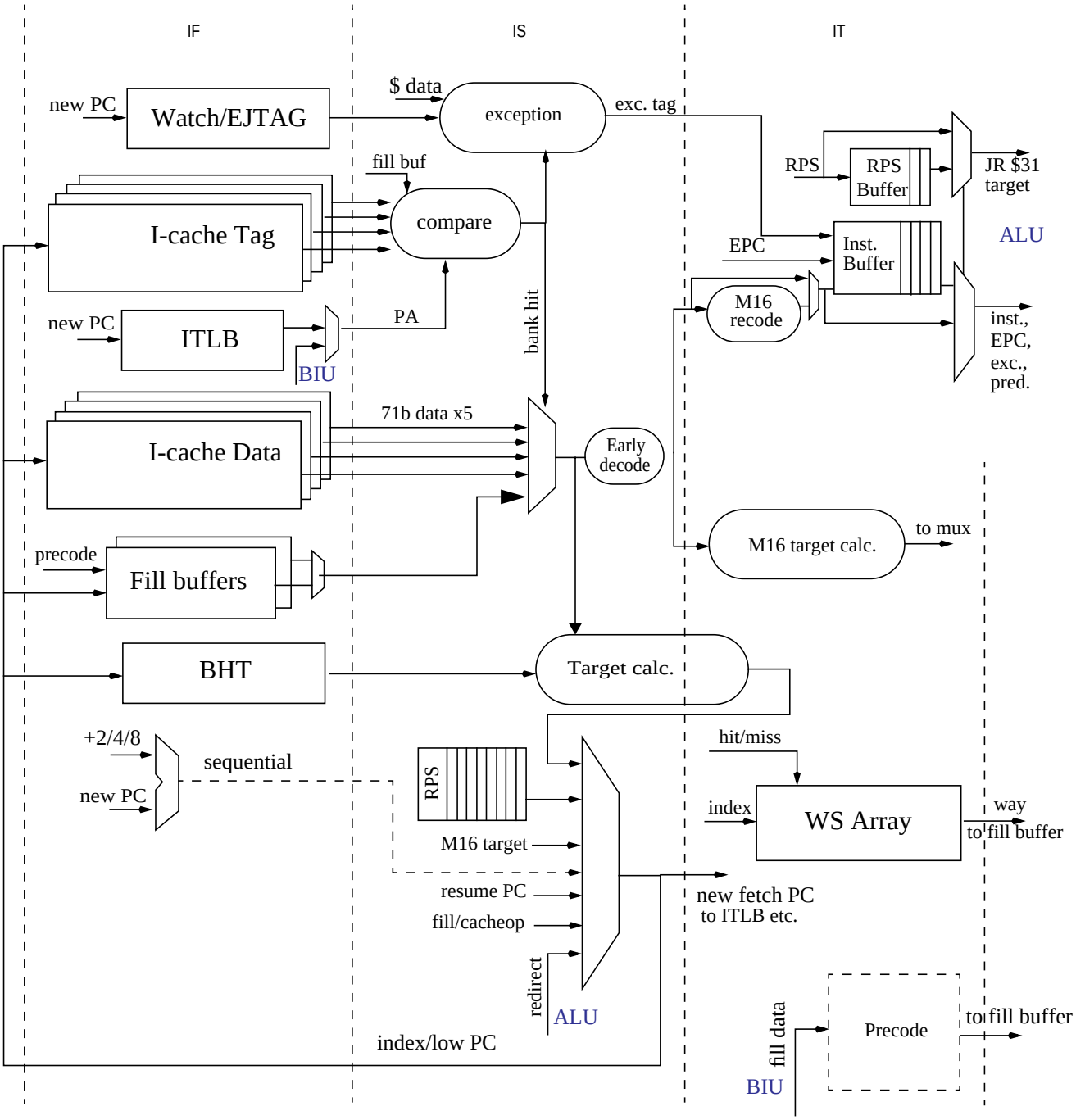
In the IT stage of the pipe, a dispatch scheduler unit chooses amongst all the TCs with an instruction available in its instruction buffer for dispatch.

The instruction buffer in the IT stage of the pipe also doubles as a skid buffer. In the event of a cache miss on a load followed by a dependency on the load data, the dependent instruction as well as any subsequent instructions for that

TC that might have been issued from the IT stage are flushed back to the instruction buffer. This is done to avoid stalling the whole pipe so that the other TCs can continue with their operations. However, by flushing these instructions to the instruction buffer instead of causing a refetch from the cache, when the dependency for this TC is resolved, this TC can start issuing immediately instead of waiting for a full fetch from the caches.

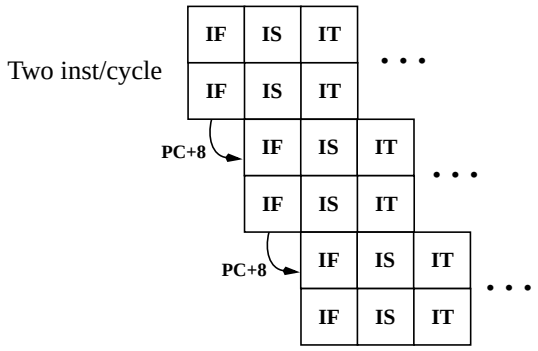
Figure 2.2 shows the general datapath of the IFU along with major structures. In order to avoid complexity, the figure below does not include the TC selection mux for fetching (at front of pipe) or the TC selection mux in the policy manager in the IT stage.

**Figure 2.2 IFU Block Diagram**



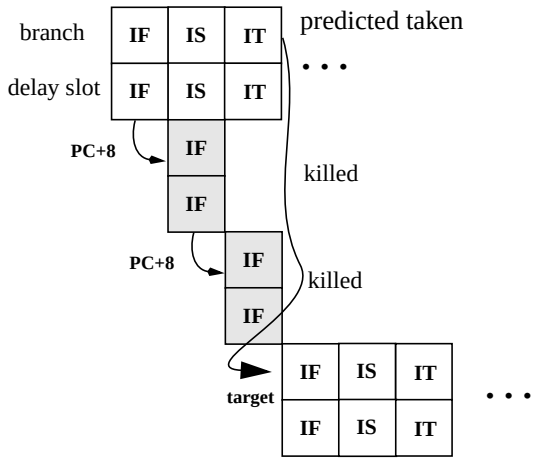

The following diagrams illustrate the timing of various IFU operations. The simplest of these is the sequential fetch path, in which the next fetch PC is incremented by 8 bytes in parallel with the cache lookup. If each fetch hits in the cache, the IFU can provide two instructions per cycle and will quickly fill up the instruction buffer, after which it will stall based on a buffer full signal. However, if there are other TCs that do not have their instruction buffer full, the IFU will continue to fetch those TCs, avoiding a stall in the IFU.

**Figure 2.3 Timing of 32-bit Mode Sequential Fetches**



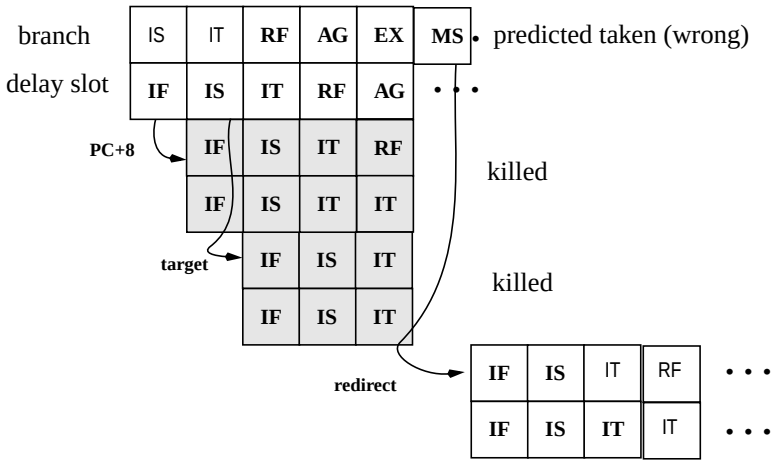

Another common situation is a control transfer instruction (branch/jump). The calculation of the target for 32-bit mode instructions starts in the IS stage, but does not complete until the IT stage. For a predicted taken path this means that if the delay slot of that branch is in the same fetch bundle, there will be a 2 cycle bubble since the sequential fetches will not be used. If the delay slot is in the next fetch bundle, there will be a 1 cycle bubble.

**Figure 2.4 Timing of 32-bit Mode Branch Taken Path**



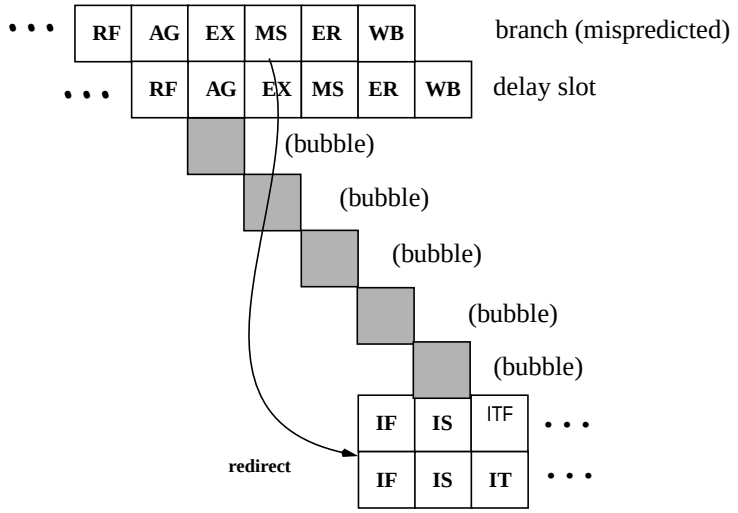

For conditional branches, the control transfer is most likely speculative, based upon the branch history table. The resolution of this branch by the ALU will be calculated in the EX stage and will be used by the IFU in the MS stage, resulting in a several-cycle fetch bubble. The following figure illustrates one possibility assuming the instruction buffer is empty and the delay slot is in the next fetch bundle.

**Figure 2.5 Fetch Timing of 32-bit Mode Branch Mispredict**




The delay slot lessen the impact of a mispredict on the execution pipeline, though. Assuming no stalls, the ALU sees a five-cycle bubble:

**Figure 2.6 Execution Timing of 32-bit Mode Branch Mispredict**




## 2.2.1 Branch History Table

A branch history table (BHT) will be accessed in parallel with the cache in the IF stage. This table is a 512-entry bimodal predictor. The table is indexed with bits 11:3 of the VA and each entry contains a two bit saturating counter that indicates whether a branch is taken or not. The indexing is down to bit 3 because in 32b code there can only be one branch every 64b because of the branch delay slot. In MIPS16e code, the smaller instructions and lack of delay slots means that up to 4 branches can exist within a 64b fetch bundle and will share the same BHT entry. However, in typical code, the branch density is lower than in 32b code and keeping the same 64b indexing maintains reasonable prediction accuracy. It is also important to note that the on the 34K core, all the TCs use the same BHT.

Unlike some previous MIPS processors, the 34K core uses the BHT to predict branch likely instructions. Architecturally, these are specified to only be used when a branch is taken > 95% of the time. However, the default settings of many compilers use these even when that is not the case. The delay slot characteristics (the delay slot is only executed if the branch is taken) allow a useful instruction to be placed in the delay slot instead of a NOP. When used in this fashion, dynamic prediction is much more accurate than statically predicting a branch likely as taken.

Unconditional branches (BEQ r0, r0 and BGEZAL r0) are detected by the precode logic and will be statically predicted taken, bypassing the BHT.

The ALU verifies the correctness of the prediction when the branch reaches the EX stage. In the case of a mispredict, the instructions on the mispredicted path will be killed and the fetch will be redirected to the correct instruction. This will cause a 5 cycle bubble in the pipeline.

#### 2.2.1.1 Branch Target Calculation

Branch target calculation is done in the IT stage. This alleviates a critical timing path in the IFU and removes the need for replicating the branch target logic on all 4 ways of the cache. In the case of a taken branch, the following two fetches will be killed (only one if the delay slot is in the following fetch). This added cycle is generally covered by the instruction buffer. A string of taken branches will slowly drain the instruction buffer as only two instructions are fetched every three cycles.

### 2.2.2 Return Prediction Stack

The return prediction stack (RPS) is a simple stack to hold return addresses. Every time a JAL, JALR ra, or BGEZAL is seen, the link address is pushed onto the stack. When a JR ra is executed, a link address is popped off of the stack. If calling convention is maintained and the stack doesn't overflow, this will have very high prediction accuracy. The RPS contains 4 entries.

The ALU will verify the correctness of the prediction in the EX stage. If the prediction was wrong, the fetch will be redirected in the MS stage and there will be a 5 cycle bubble from the misprediction.

JR that don't use ra are not predicted. The IFU will stall until the ALU reads the register file. The timing on this will be the same as for a return mispredict.

On the 34K core, there is one RPS shared between all the TCs. However, only one TC can use it at any point in time. A TC begins to utilize the RPS when it is the only TC that is runnable (i.e. when the 34K core enters single-threaded mode).

### 2.2.3 ITLB

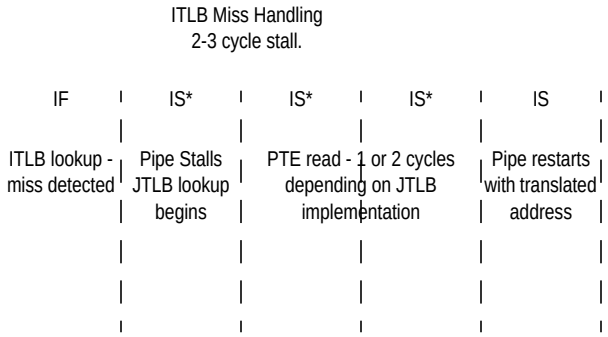
The IFU relies on a small subset of TLB entries stored locally in a four-entry ITLB to translate the PC into a physical address for tag comparison. The ITLB stores mappings for 4KB or 1MB pages or sub-pages (i.e. if the JTLB page is 64KB, only the 4KB sub-page containing the desired virtual address will be mapped into the ITLB). The ITLB access occurs in parallel with the primary cache lookup. If there is a miss in the ITLB, the BIU must look up the entry in the main JTLB.

On the 34K core, to avoid multiple misses from many TCs thrashing the ITLB, there is a per-TC ITLB entry. Thus, a TC has access to one dedicated per-TC ITLB entry in addition to the three other common ITLB entries. The TC specific ITLB entry is loaded with the last used translation by that TC when that translation gets evicted out of the three common ITLB entries.

A miss in the ITLB will be detected in the IF stage, and the IFU will stall in the IS stage. The virtual address and the miss indication will be sent to the BIU during IF, allowing the JTLB to start a lookup in the next cycle. The latency of the JTLB lookup can be impacted by several factors. The JTLB can be busy processing a DTLB miss or a TLB operation, delaying the start of the JTLB lookup. Also, the JTLB access time depends on how it is implemented. An SRAM-based PFN array will take an extra cycle over a flop-based version, yielding a 3 cycle latency instead of 2. Once the JTLB data is returned, the IFU will resume directly in the IS stage. When there are multiple TCs running, after an ITLB miss request is dispatched to the JTLB, other TCs can keep fetching behind this ITLB miss. If another TC has an ITLB miss, then that TC will only restart fetching once the first ITLB miss has been serviced by the JTLB. In other words, at any point in time there can be only one outstanding ITLB miss but the ITLB miss is non-blocking for other TCs.

The cacheability attributes can be reduced to one bit (uncached/cached). An ITLB entry will also record the associated JTLB entry, so that for a JTLB write, the ITLB can invalidate its copy if present. The ITLB uses a true LRU replacement algorithm.

**Figure 2.7 Timing of an ITLB Miss**

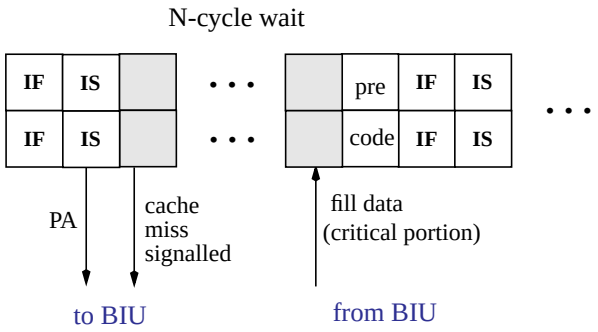



## 2.2.4 Cache Miss Timing

A miss in the instruction cache will be detected in the IS stage. The IFU will allocate one of the entries in the fill buffer and send the translated physical address and the miss indication to the BIU during the next cycle. The IFU will then enter an idle state and, assuming no redirect event, will replay the IF stage once the data returns from the BIU. Prior to writing into the cache, the IFU precodes the instructions with some additional information about branches/jumps that help speed up fetch unit processing of those instructions. Precoding the instructions and the write into the fill buffer will happen in the cycle the BIU returns the data, and in the following IF stage the data can be bypassed from the fill buffer. Thus, the IFU portion of the cache miss penalty is normally 4 cycles. The total miss penalty could range from a minimum of 10-12 cycles for an L2 hit to 50 or more for an access to main memory.

In the case of the 34K core with multiple TCs running, a cache miss is non-blocking. Up to three outstanding cache misses are supported and it is possible for other TCs to keep fetching as long as they are hitting out of the instruction cache even with three pending cache misses.

**Figure 2.8 Timing of a Cache Miss**




### 2.2.5 MIPS16e™

The IFU is responsible for recoding MIPS16e instructions. Before the MIPS16e instruction is sent to the ALU, it is recoded into a 32b instruction. Some additional state is used for the MIPS16e instructions that do not have a direct counterpart in the MIPS32 instruction set (such as PC-relative loads and adds). This recoding step is handled in an additional pipeline stage that is only active when executing MIPS16e code.

Each cycle, the recode logic processes 32b of the instruction stream and puts 1-2 instructions in the fetch buffer. Most instructions can be generated two at a time, but there are two exceptions. JAL(X) instructions are 32b. When the JAL(X) is in the 32b fetch window, it will be recoded in one cycle. If the JAL(X) starts in the middle of a fetch window, the first instruction will be recoded in the first cycle and the fetch window will be shifted so the JAL(X) can be recoded in the second cycle. EXTENDs are handled the same way - the EXTEND and the instruction it is extending are only recoded when they are in the fetch window together. Since a single fetch of 64bits can result in up to 4 MIPS16e instructions, in MIPS16e mode, the processor fetches every other cycle. On the 34K core, it is possible for these empty fetch cycles to be consumed by any other TC running in MIPS32 mode and thus keep the fetch pipe busy.

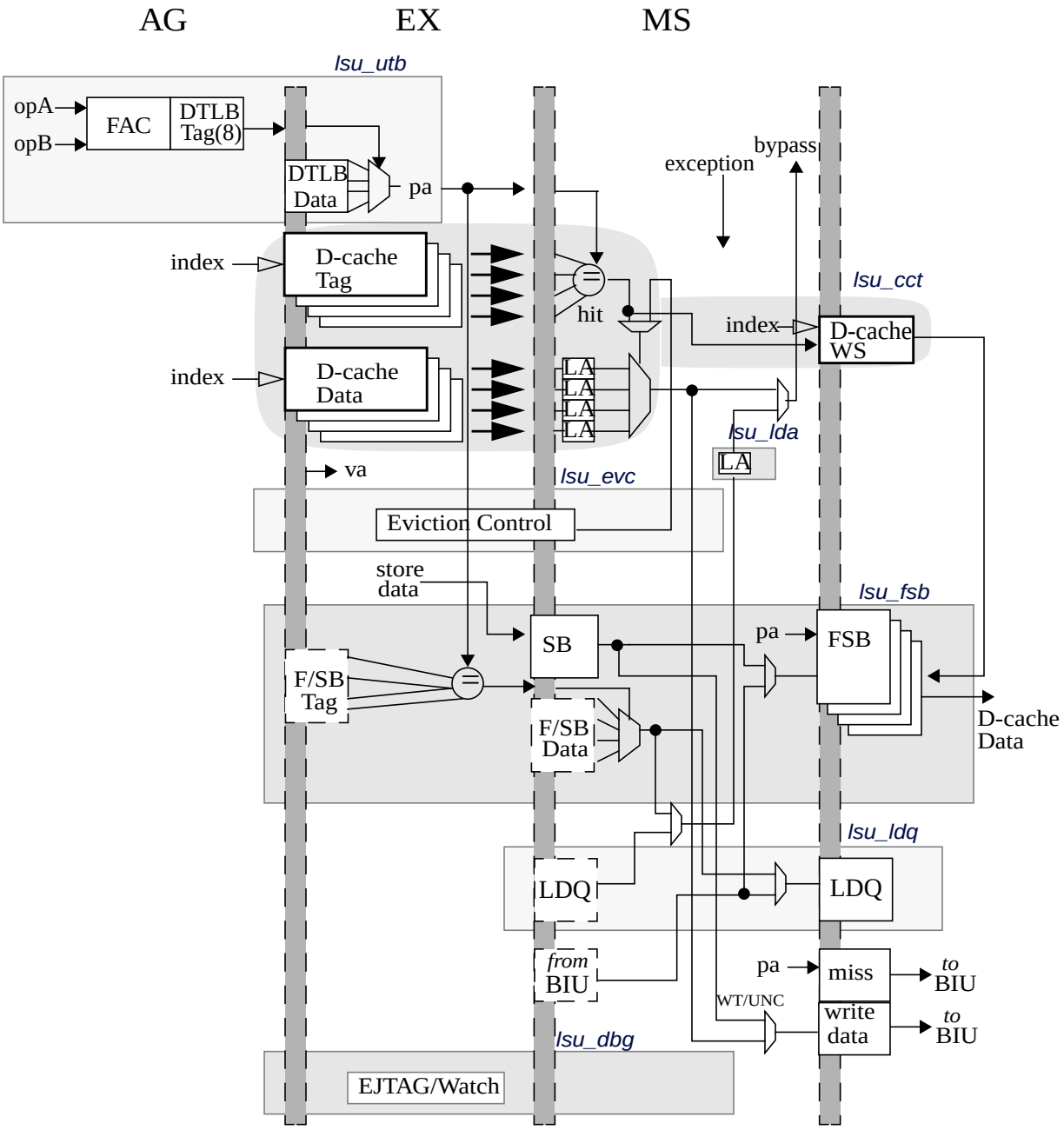
**Table 2.1 Recode bandwidth**

| First 16b | Second 16b | 32b Instns generated |
|---|---|---|
| 16b instn | 16b instn | 2 |
| Extend | 16 instn | 1 |
| 16b instn | Extend/JAL(X) | 1 |
| JAL(X) | | 1 |

## 2.3 Load Store Unit

The Load Store Unit (LSU) is responsible for loads and stores. This primarily includes the data cache control logic.

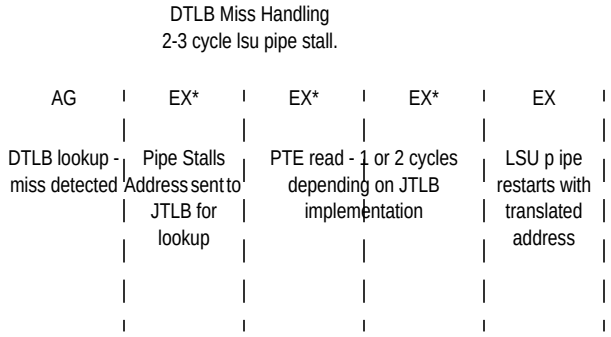
**Figure 2.9 LSU Pipeline**




## 2.3.1 DTLB

The data cache access begins in the AG stage. The ALU generates the virtual address in this stage. In parallel, the source operands are passed to the LSU and the 8 entry DTLB is accessed. If there is a miss in the DTLB, the LSU will stall and give the address to the BIU to lookup in the JTLB. If there is a hit in the JTLB, the page information will be returned to the LSU and the access will continue. Since it is only the LSU pipe that stalls on a DTLB miss, it is possible for other non load-store instructions to keep progressing down the ALU pipe.

**Figure 2.10 DTLB Miss Timing**




The DTLB will only store mappings either for 4K or 1M pages or subpages of a larger JTLB entry. A DTLB entry will also record the associated JTLB entry, so that for a JTLB write, the DTLB can invalidate its copy if present. The DTLB uses a pseudo-LRU replacement algorithm. If the Fixed Mapping MMU is used instead of a TLB, the address translation will be done in the EX stage and there will never be a DTLB miss.

## 2.3.2 Data Cache Access

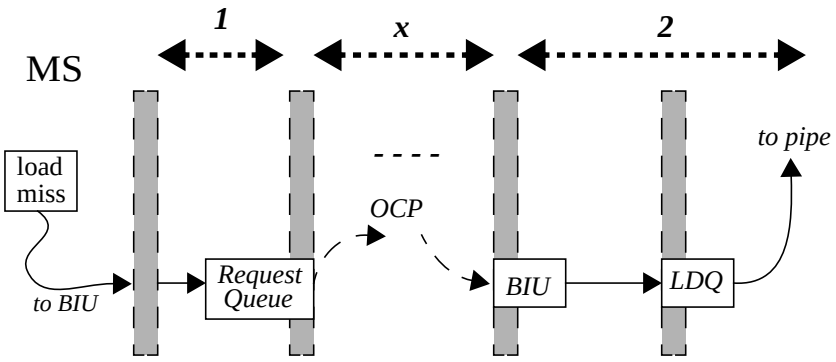
The data cache access is done during the EX stage. The tag and data arrays are accessed and the values are saved in flops for use in the MS stage. In parallel with the array lookup (in EX), the physical address is used to do an early tag compare on entries in the Fill Store Buffer (FSB) and Store Buffer (SB).

The SB is a single entry buffer that is used to stage store data into the other structures. It is fully bypass-able, allowing a load immediately after a store to the same address to execute without stalls. From the SB, the store data will move into the FSB if the store hits in the cache or it is an allocating miss. The store data is then written into the cache opportunistically.

During the MS stage, the data cache tags are compared to the physical address to determine whether a reference hit in the cache or not. If there is a hit, the way select (WS) array will be written to mark the most recently used way, and load data will be bypassed back to the ALU. On a cache miss, an FSB entry is allocated to hold the fill data as it returns from the BIU. The WS array is read and the replacement way is determined. If the line selected for replacement is dirty, an eviction will begin and the dirty data will be written back to memory. A load miss will also allocate an entry in the Load Queue (LDQ). This buffer is used to hold the aligned load data while it is being staged back into the ALU.

The core portion of a load miss is shown in Figure 2.11. It takes one cycle to get from the LSU through the BIU and out onto the OCP bus. It takes at least 1 cycle for the data to be returned. Then 2 more cycles are required to get the data back to the ALU.

**Figure 2.11 Cache Miss Timing**




### 2.3.3 Outstanding misses

The 34K core features non-blocking D-cache misses. In the cases where the following instructions are not dependent on the load data, the core can continue executing instructions while the miss is being processed. The core can handle multiple outstanding misses.

- Up to 8 independent cache lines - this includes lines requested for loads, stores, and prefetches. Multiple requests to the same line can be merged.
- Up to 9 load misses - Up to 9 separate loads can be outstanding. The loads can be to different cache lines or multiple loads can be to the same cache line.

### 2.3.4 Uncached Accesses

Uncached accesses are handled pretty similarly to cached ones. The cacheability of the reference is not known until the address translation is completed in the EX stage, so the cache access is performed anyway. On an uncached reference, a miss will be forced. Uncached loads will request the exact amount of data required and allocate an FSB and LDQ entry. Uncached loads are non-blocking just like cached misses. Uncached stores will be sent to the BIU.

To the LSU, uncached accelerated stores look the same as uncached stores. They are handled differently in the BIU though. The BIU will attempt to gather uncached accelerated stores and do a bursted write to improve bus efficiency.

## 2.4 MDU Pipeline

The autonomous multiply/divide unit (MDU) has a separate pipeline for multiply and divide operations. This pipeline operates in parallel with the integer unit (ALU) pipeline and does not stall when the ALU pipeline stalls. This allows multi-cycle MDU operations, such as a divide, to be partially masked by system stalls and/or other integer unit instructions.

The MDU consists of a 32x32 booth recoded multiplier array, separate carry-lookahead adders for multiply and divide, result/accumulation registers (*HI* and *LO*), multiply and divide state machines, and all necessary multiplexers and control logic.

The MDU supports execution of a multiply operation every clock cycle. Divide operations are implemented with a simple 1 bit per clock iterative algorithm with an early in detection of sign extension on the dividend *(rs)*. An attempt to issue a subsequent MDU instruction which would access the *HI* or *LO* register before the divide completes causes a delay in starting the subsequent MDU instruction. Some concurrency is enabled by the separate adders for the mul-

tiply and divide data paths. The MDU instruction may start executing once the divide is ensured of writing to the *HI* and *LO* registers before the MDU instruction will access them. A MUL instruction, which does not access the *HI* or *LO* register, may start executing anytime relative to a previous divide instruction.

Table 2.2 lists the delays (number of cycles until a result is available) for multiply and divide instructions. The delays are listed in terms of pipeline clocks. In this table 'delay' refers to the number of cycles the CPU must stall the second instruction to wait for the result of the first instruction.

**Table 2.2 MDU Instruction Delays**

| Size of Operand 1st Instruction[1] | Instruction Sequence | | Delay Clocks |
|---|---|---|---|
| | 1st Instruction | 2nd Instruction | |
| 32 bit | MULT/MULTU, MADD/MADDU, or MSUB/MSUBU | MADD/MADDU, MSUB/MSUBU, or MFHI/MFLO | 0 |
| 32 bit | MUL | Integer operation[1] | 4 |
| 8 bit | DIVU | MFHI/MFLO | 7 |
| 16 bit | DIVU | MFHI/MFLO | 15 |
| 24 bit | DIVU | MFHI/MFLO | 23 |
| 32 bit | DIVU | MFHI/MFLO | 31 |
| 8 bit | DIV | MFHI/MFLO | 9[2] |
| 16 bit | DIV | MFHI/MFLO | 17[2] |
| 24 bit | DIV | MFHI/MFLO | 25[2] |
| 32 bit | DIV | MFHI/MFLO | 33[2] |
| any | MFHI/MFLO | Integer operation[1] | 4 |
| any | MTHI/MTLO | MADD/MADDU, MSUB/MSUBU | 1 |
| any | MTHI/MTLO | MFHI/MFLO | 0 |

[1] Integer Operation refers to any integer instruction that uses the result of a previous MDU operation.
[2] If both operands are positive, then the two Sign Adjust stages are bypassed. Delay is then the same as for DIVU.

In Table 2.2 a delay of zero means that the first and second instructions can be issued back to back in the code without the MDU causing any stalls in the ALU pipeline. A delay of one means that if issued back to back, the ALU pipeline will be stalled for one cycle.

**Table 2.3 Multiply Instruction (updating *HI/LO*) Repeat Rates**

| Instruction Sequence | | Repeat Rate |
|---|---|---|
| 1st Instruction | 2nd Instruction | |
| MULT/MULTU, MADD/MADDU, MSUB/MSUBU | MADD/MADDU, MSUB/MSUBU | 1 |

The repeat rate of 1 for MULT/MULTU/MADD/MADDU/MSUB/MSUBU to MADD/MADDU/MSUB/MSUBU are achieved by feeding the result of the $M3_{MDU}$ stage for the first instruction back into the $M3_{MDU}$ stage for the second instruction.

**Table 2.4 MUL Repeat Rates**

| Instruction Sequence | | Repeat Rate |
|---|---|---|
| 1st Instruction | 2nd Instruction | |
| MUL | MUL (no data dependency) | 1-3[1,2] |

[1] There is no data dependency between first and second MUL. Otherwise, the repeat rate is the same as for MUL to integer operations in Figure 2.2

[2] MULs can be issued at the maximum rate of 3 every 5 cycles. Three can be issued back to back, but a fourth one would stall.

## 2.4.1 Multiply Pipeline Stages

The multiply operation begins in $B_{MDU}$ stage, which would be the EX stage in the integer pipeline. The booth recoding function occurs at this time. The multiply calculation requires three clocks and occurs in the $M1_{MDU}$, $M2_{MDU}$, and $M3_{MDU}$ stages. The carry-lookahead-add (CLA) function occurs at the end of the $M3_{MDU}$ stage. In the $A_{MDU}$ stage, the result is selected from the multiply data path, *HI* register, and *LO* register to be returned to the ALU for the MFHI, MFLO, and MUL instructions. If the MDU instruction is not one of these, the result is selected to be written into the *HI/LO* registers instead. The result is ready to be read from the *HI/LO* registers in the $W_{MDU}$ stage.

The following figures illustrate a multiply (accumulate) instruction and the interaction with the main integer pipeline. These figures are applicable to MUL, MULT, MULTU, MADD, MADDU, MSUB, and MSUBU instructions

**Figure 2.12 Multiply Pipeline**


**Figure 2.13 Multiply With Dependency From ALU**

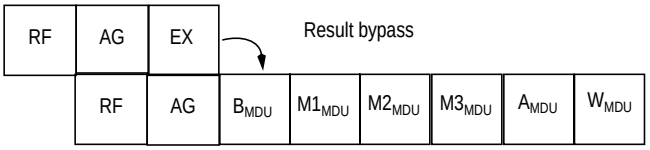


**Figure 2.14 Multiply With Dependency From Load Hit**

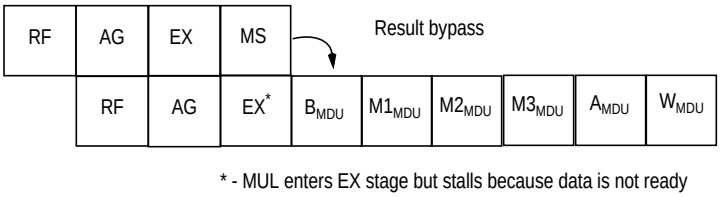
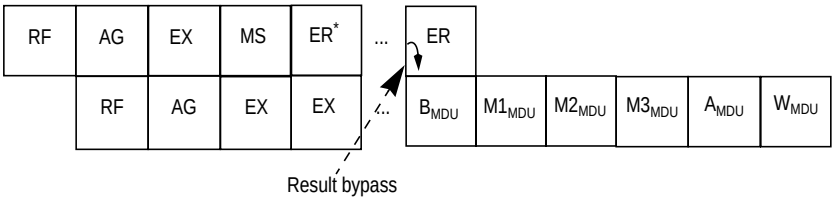


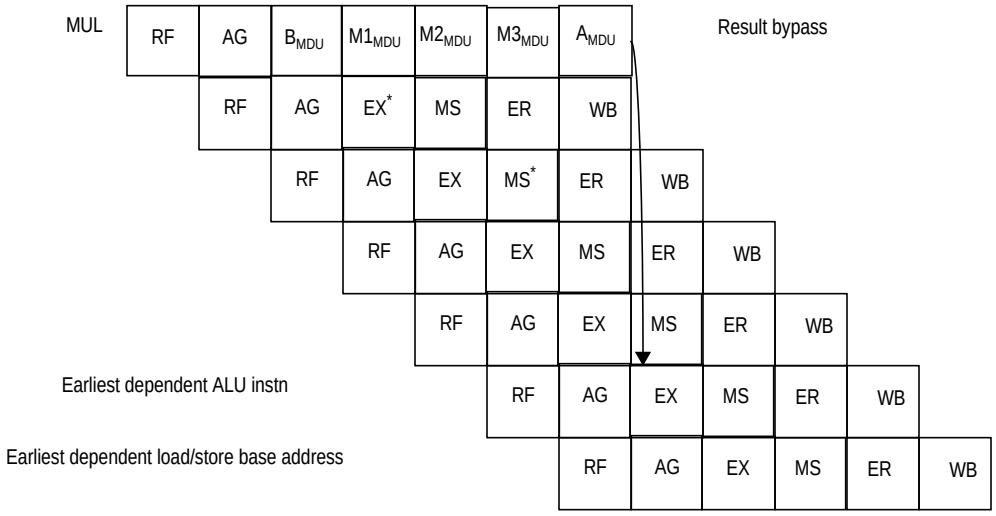

**Figure 2.15 Multiply With Dependency From Load Miss**




The following figure shows the results of the GPR targeted MUL instruction being bypassed to a later instruction. Independent instructions can execute while the multiply is happening. If a dependent instruction is found, it will stall until the result is available. When the MUL completes, it will arbitrate for access to the write port of the register file. If the integer pipe is busy with other instructions, the MDU pipeline will stall until the result can be written.

If the MUL target is being used as the base address for a load or store instruction, it needs to be bypassed by the AG stage, so one extra cycle will be required.

**Figure 2.16 MUL Bypassing Result to Integer Instructions**



## 2.4.2 Divide Operations

Divide operations are implemented using a simple non-restoring division algorithm. This algorithm works only for positive operands, hence the first cycle of the $M_{MDU}$ stage is used to negate the *rs* operand (RS Adjust) if needed. Note that this cycle is spent even if the adjustment is not necessary. In cycle 2, the first add/subtract iteration is executed. In cycle 3 an early-in detection is performed. The adjusted *rs* operand is detected to be zero extended on the upper most 8, 16 or 24 bits. If this is the case the following 7, 15 or 23 cycles of the add/subtract iterations are skipped. During the next maximum 31 cycles (4-34), the remaining iterative add/subtract loop is executed.

The remainder adjust (Rem Adjust) cycle is required if the remainder was negative. Note that this cycle is spent even if the remainder was positive. A sign adjust is performed on the quotient and/or remainder if necessary. The sign adjust stages are skipped if both operands are positive.

Figure 2.17, Figure 2.18, Figure 2.19 and Figure 2.20 show the worst case latencies for 8, 16, 24 and 32 bit divide operations, respectively. The worst case repeat rate is either 14, 22, 30 or 38 cycles (two less if the *sign adjust* stage is skipped).

**Figure 2.17 MDU Pipeline Flow During a 8-bit Divide (DIV) Operation**

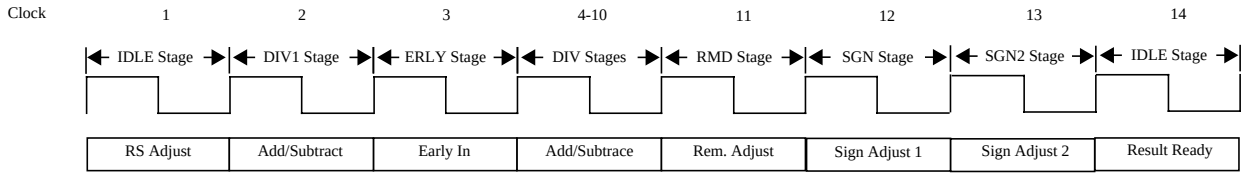



**Figure 2.18 MDU Pipeline Flow During a 16-bit Divide (DIV) Operation**

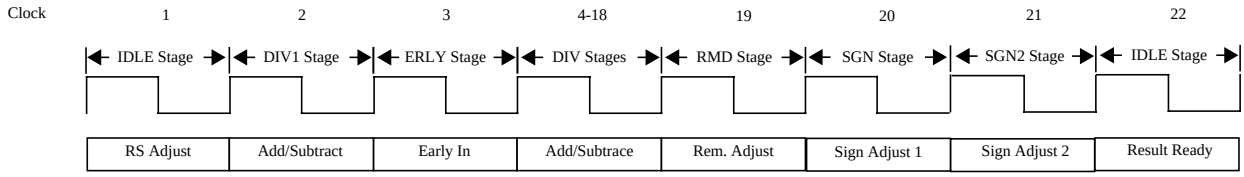



**Figure 2.19 MDU Pipeline Flow During a 24-bit Divide (DIV) Operation**

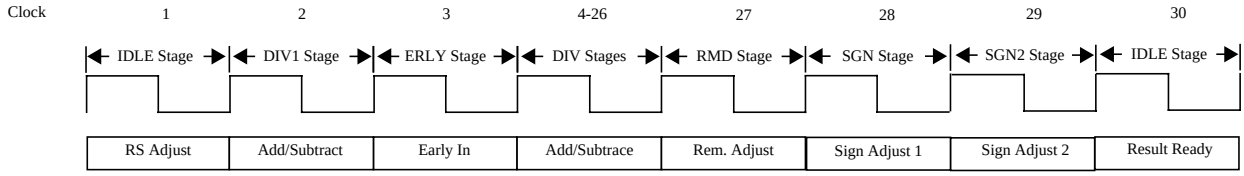
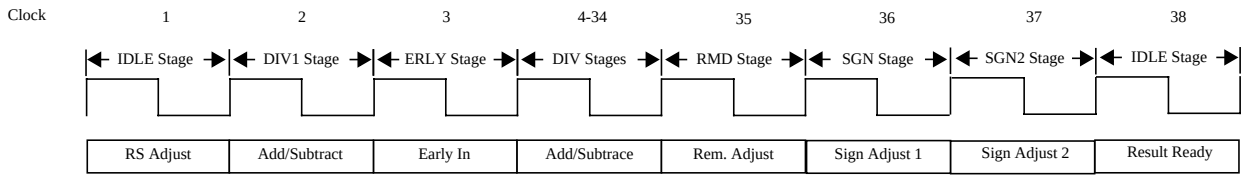


**Figure 2.20 MDU Pipeline Flow During a 32-bit Divide (DIV) Operation**




## 2.5 Skewed ALU

The 34K core has a skewed ALU. This is referring to the fact that the ALU is located in the EX stage instead of the AG stage. This allows the load to use delay to be two cycles, the same as it was in the shorter 4KE pipeline. Software optimized for that pipeline can run without incurring additional stalls. Of course, this does not come for free - an ALU instruction generating the base address for a load or store will have an additional cycle stall. Independent of the ALU location, pointer chasing loads (loads generating the base address for following loads) will see the full 3 cycles of cache access time.

This is shown in Figure 2.21. The earliest an ALU consumer of load data can issue is two cycles after the load. The earliest a load/store consumer can issue is three cycles after the load.

The bypass of data from the ALU is shown in Figure 2.22. For back to back ALU instructions, the result is bypassed from the EX stage to the AG stage. For an ALU bypassing to the base address register of a load or store, the bypassing is from the EX stage to the RF stage and the load cannot issue until two cycles after the ALU instruction. Note that the data register for a store is not used in the AG stage and a dependency there will look like the ALU to ALU bypass.

**Figure 2.21 Load Data Bypass**

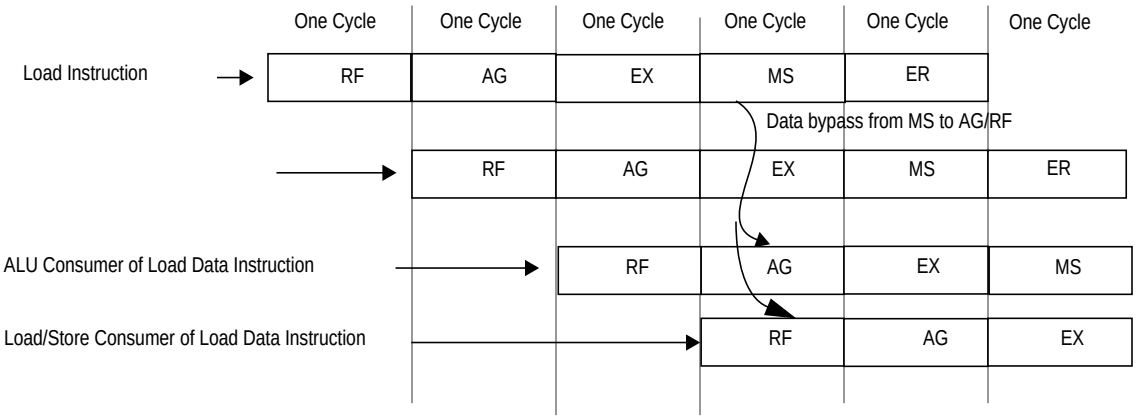
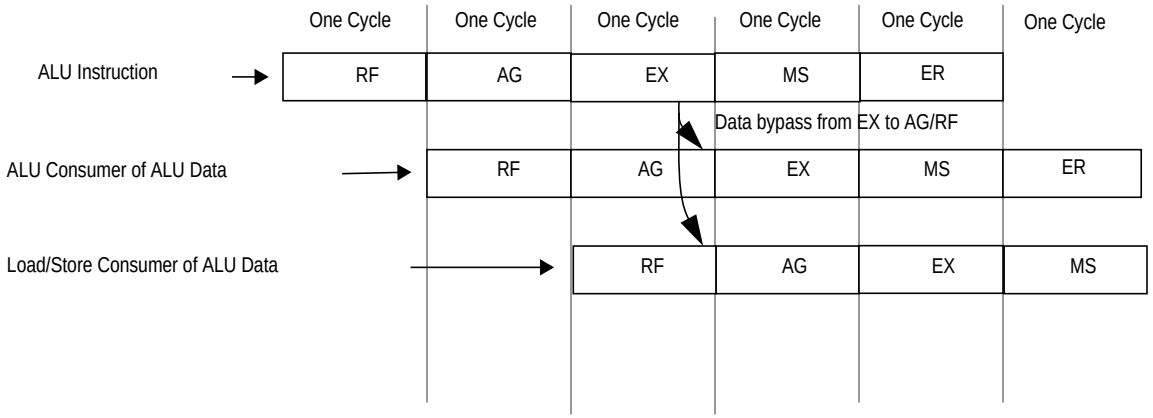


**Figure 2.22 ALU Data Bypass**




## 2.6 Interlock Handling

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

Table 2.5 lists the types of pipeline interlocks for the 34K processor core.

**Table 2.5 Pipeline Interlocks**

| Interlock Type | Sources | Rollback Condition | Slip Stage |
|---|---|---|---|
| GPR dependency - load/store address | Dest. register for any instruction in previous cycle | Dependency caused by a long latency load | AG |
| | Dest. register for loads/MFCx/MDU instns in previous 2 cycles | | |
| MDU busy | Previous MDU operation not completed | MFHI/LO following a DIV | AG |
| GPR dependency | Dest. register for loads/MFCx/MDU instns in previous cycle | Dependency caused by a long latency load | EX |
| LDQ full | Load in pipe while Load Queue is full | None | |
| Blocking load bubble | Blocking load immediately following another blocking load | None | |
| SYNC, I$ CACHE | Previous I$ CACHE not completed | I$ CACHE inst. while previous I$ CACHE not completed | |

**Table 2.5 Pipeline Interlocks (Continued)**

| Interlock Type | Sources | Rollback Condition | Slip Stage |
|---|---|---|---|
| Destination GPR dependency | Outstanding GPR write to same register | Dependency caused by a long latency load | MS |
| WBB full | Store/CACHE instn in pipe while Write-back Buffer is full | None | |
| SPRAM busy | SPRAM load/store in pipe while SPRAM is busy | None | |
| FSB flush | SYNC/CACHE/load/store instn requires Fill Store Buffer to be flushed | None | |
| DTLB miss | Load/Store address miss in microTLB | None | |
| CACHE | CACHE instn needs to re-access data cache | None | |
| L2 CACHE | Previous L2 CACHE not completed | None | |
| Blocking load miss | Load misses with non-blocking loads disabled | None | ER |

In the 34K, some interlocks stall the pipeline and block all TCs until the interlock condition is satisfied. In other cases, the instruction that is waiting for an interlock event is rolled back to allow other TCs to make progress. Table 2.5 identifies the interlocks that can be rolled back.

## 2.7 Instruction Interlocks

Most instructions can be issued at a rate of one per clock cycle. In order to adhere to the sequential programming model, the issue of an instruction must sometimes be delayed. This to ensure that the result of a prior instruction is available. Table 2.6 details the instruction interactions that prevent an instruction from advancing in the processor pipeline.

**Table 2.6 Instruction Interlocks**

| Instruction Interlocks | | | |
|---|---|---|---|
| **First Instruction** | **Second Instruction** | **Issue Delay (in Clock Cycles)** | **Slip Stage** |
| LB/LBU/LH/LHU/LL/LW/LWL/LWR | ALU Consumer of load data | 1 | EX stage |
| | Load/Store consumer for base address register | 2 | AG stage |
| MFC0 | ALU consumer of destination register | 2 | EX stage |
| | Load/store consumer for base address | 3 | AG stage |
| MULTx/MADDx/MSUBx | MFLO/MFHI | 0 | |
| MUL/MFHI/MFLO | ALU Consumer of target data | 4 | EX stage |
| | Load/Store consumer of target data for base address | 5 | AG stage |
| MULTx/MADDx/MSUBx | MULT/MUL/MADD/MSUB MTHI/MTLO/DIV | 0 | EX stage |

**Table 2.6 Instruction Interlocks**

| Instruction Interlocks | | | |
|---|---|---|---|
| **First Instruction** | **Second Instruction** | **Issue Delay (in Clock Cycles)** | **Slip Stage** |
| DIV | MUL/MULTx/MADDx/ MSUBx/MTHI/MTLO/ MFHI/MFLO/DIV | See Figure 2.2 | EX stage |
| TLBWR/TLBWI | Load/Store/PREF/CACHE/ COP0 op | 2 | EX stage |
| TLBR | | 1 | EX stage |

# 2.8 Hazards

In general, the 34K core ensures that instructions are executed following a fully sequential program model. Each instruction in the program sees the results of the previous instruction. There are some deviations to this model. These deviations are referred to as *hazards*.

Prior to Release 2 of the MIPS32® Architecture, hazards (primarily CP0 hazards) were relegated to implementation-dependent cycle-based solutions, primarily based on the SSNOP instruction. This has been an insufficient and error-prone practice that must be addressed with a firm compact between hardware and software. As such, new instructions have been added to Release 2 of the architecture which act as explicit barriers that eliminate hazards. To the extent that it was possible to do so, the new instructions have been added in such a way that they are backward-compatible with existing MIPS processors.

However, on the 34K core there are a number of inter-TC hazards that cannot be resolved even by using the hazard barrier instructions. As such on the 34K core the TCs have no relation to each other and software has to enforce that relation to avoid these hazards.

## 2.8.1 Types of Hazards

With one exception, all hazards were eliminated in Release 1 of the Architecture for unprivileged software. The exception occurs when unprivileged software writes a new instruction sequence and then wishes to jump to it. Such an operation remained a hazard, and is addressed by the capabilities of Release 2.

In privileged software, there are two different types of hazards: *execution hazards* and *instruction hazards*. Both are defined below.

### 2.8.1.1 Execution Hazards

Execution hazards are those created by the execution of one instruction, and seen by the execution of another instruction. These hazards should be resolved by executing an EHB instruction or an instruction hazard barrier (JR.HB, JALR.HB, or ERET) between the two instructions. Table 2.7 lists execution hazards.

**Table 2.7 Execution Hazards**

| Producer | → | Consumer | Hazard On | Spacing (Instructions) |
|---|---|---|---|---|
| TLBWR, TLBWI | → | TLBP, TLBR | TLB entry | 2 |
| | | Load/store using new TLB entry | TLB entry | 3 |

**Table 2.7 Execution Hazards**

| Producer | → | Consumer | Hazard On | Spacing (Instructions) |
|---|---|---|---|---|
| MTC0 | → | Load/store affected by new state | WatchHi<br>WatchLo | 2 |
| MTC0 | → | MFC0 | any cp0 register | 2 |
| MTC0 | → | EI/DI | Status | 2 |
| MTC0 | → | RDHWR $3 | Count | 2 |
| MTC0 | → | ERET | EPC<br>DEPC<br>ErrorEPC | 2 |
| MTC0 | → | ERET | Status | 2 |
| EI, DI | → | Interrupted instruction | $Status_{IE}$ | 2 |
| MTC0 | → | Interrupted instruction | Status | 2 |
| MTC0 | → | User-defined instruction (only for Pro core) | StatusERL StatusEXL | 4 |
| MTC0 | → | Interrupted Instruction | $Cause_{IP}$ | 2 |
| TLBR | → | MFC0 | EntryHi,<br>EntryLo0,<br>EntryLo1, Page-Mask | 2 |
| TLBP | → | MFC0 | Index | 2 |
| MTC0 | → | TLBR<br>TLBWI<br>TLBWR | EntryHi | 2 |
| MTC0 | → | TLBP<br>Load/store affected by new state | $EntryHi_{ASID}$ | 2 |
| MTC0 | → | TLBWI<br>TLBWR | EntryLo0<br>EntryLo1 | 2 |
| MTC0 | → | TLBWI<br>TLBWR | Index | 2 |
| MTC0 | → | RDPGPR<br>WRPGPR | $SRSCtl_{PSS}$ | 1 |
| MTC0 | → | Instruction not seeing a Timer Interrupt | Compare update that clears Timer Interrupt | 4[1] |
| MTC0 | → | Load/Store affected by new state | $EntryHi_{ASID}$ | 3 |
| MTC0 | → | Load/Store affected by new state | $Status_{ERL}$ | 3 |
| MTC0 | Æ | Load/Store affected by new state | $Debug_{LSNM}$ | 3 |
| MTC0 | Æ | Coprocessor instruction affected by new state | $Status_{CU}$ | 4 |
| MTC0 | Æ | Coprocessor instruction affected by new state | $Status_{FR}$ | 4 |
| MTCO | Æ | DSP instruction affected by new state | $Status_{MX}$ | 4 |
| MTCO | Æ | CorExtend instruction affected by new state | $Status_{CEE}$ | 3 |
| MTC0 | Æ | MFTR / MTTR | $VpeControl_{TargTC}$ | 4 |

**Table 2.7 Execution Hazards**

| Producer | → | Consumer | Hazard On | Spacing (Instructions) |
|---|---|---|---|---|
| MTC0 | → | Instruction affected by change | Any other CP0 register | 2 |

1. This is the minimum value. Actual value is system-dependent since it is a function of the sequential logic between the *SI_TimerInt* output and the external logic which feeds *SI_TimerInt* back into one of the *SI_Int* inputs, or a function of the method for handling *SI_TimerInt* in an external interrupt controller.

#### 2.8.1.2 Instruction Hazards

Instruction hazards are those created by the execution of one instruction, and seen by the instruction fetch of another instruction. Table 2.8 lists instruction hazards. Because the fetch unit is decoupled from the execution unit, these hazards are rather large. The use of a hazard barrier instruction is highly recommended for reliable clearing of instruction hazards.

**Table 2.8 Instruction Hazards**

| Producer | → | Consumer | Hazard On | Spacing (Instructions) |
|---|---|---|---|---|
| TLBWR, TLBWI | → | Instruction fetch using new TLB entry | TLB entry | 10 |
| MTC0 | → | Instruction fetch seeing the new value including:<br>1 change to ERL followed by an instruction fetch from the useg segment and<br>2 change to ERL or EXL followed by a Watch exception | Status | 10 |
| MTC0 | → | Instruction fetch seeing the new value | $\text{EntryHi}_{\text{ASID}}$ | 10 |
| MTC0 | → | Instruction fetch seeing the new value | WatchHi<br>WatchLo | 10 |
| Instruction stream write via CACHE | → | Instruction fetch seeing the new instruction stream | Cache entries | 10 |
| Instruction stream write via store | → | Instruction fetch seeing the new instruction stream | Cache entries | System-dependent[1] |

1. This value depends on how long it takes for the store value to propagate through the system.

### 2.8.2 Instruction Listing

Table 2.9 lists the instructions designed to eliminate hazards. See the document titled *MIPS32® Architecture for Programmers Volume II: The MIPS32 Instruction Set* (MD00084) for a more detailed description of these instructions.

**Table 2.9 Hazard Instruction Listing**

| Mnemonic | Function |
|---|---|
| EHB | Clear execution hazard |
| ERET | Clears both execution and instruction hazards |
| JALR.HB | Clears both execution and instruction hazards |
| JR.HB | Clears both execution and instruction hazards |
| SYNCI | Synchronize caches after instruction stream write |

#### 2.8.2.1 Instruction Encoding

The EHB instruction is encoded using a variant of the NOP/SSNOP encoding. This encoding was chosen for compatibility with the Release 1 SSNOP instruction, such that existing software may be modified to be compatible with both Release 1 and Release 2 implementations. See the EHB instruction description for additional information.

The JALR.HB and JR.HB instructions are encoding using bit 10 of the *hint* field of the JALR and JR instructions. These encodings were chosen for compatibility with existing MIPS implementations, including many which pre-date the MIPS32 architecture. Because a pipeline flush clears hazards on most early implementations, the JALR.HB or JR.HB instructions can be included in existing software for backward and forward compatibility. See the JALR.HB and JR.HB instructions for additional information.

The SYNCI instruction is encoded using a new encoding of the REGIMM opcode. This encoding was chosen because it causes a Reserved Instruction exception on all Release 1 implementations. As such, kernel software running on processors that don't implement Release 2 can emulate the function using the CACHE instruction. SYNCI must be used in conjunction with an instruction hazard barrier to ensure that the updated value is seen.

```
SYNCI  offset(base)
SYNC
JR.HB
NOP
```

### 2.8.3 Eliminating Hazards

The Spacing column shown in Table 2.7 and Table 2.8 indicates the number of unrelated instructions (such as NOPs or SSNOPs) that, prior to the capabilities of Release 2, would need to be placed between the producer and consumer of the hazard in order to ensure that the effects of the first instruction are seen by the second instruction. Entries in the table that are listed as 0 are traditional MIPS hazards which are not hazards on the 34K core.

With the hazard elimination instructions available in Release 2, the preferred method to eliminate hazards is to place one of the instructions listed in Table 2.9 between the producer and consumer of the hazard. Execution hazards can be removed by using the EHB, JALR.HB, or JR.HB instructions. Instruction hazards can be removed by using the JALR.HB or JR.HB instructions, in conjunction with the SYNCI instruction.

## 2.9 Instruction Rollback And Its Implications

As described earlier (and listed out in Table 2.5), the 34K core has the capability to rollback certain instructions when a TC is stalled so as to unstall the shared pipeline and allow other TCs to make forward progress. Flushing back a dependent instruction on a load miss is a good example of this. Another example is a I-side cacheop being rolled back if the fetch unit is already processing another cacheop.

However, instruction rollbacks associated with dependency stalls on shared resources (like cacheops) can have some interesting side effects. As an example, if two TCs are executing cacheops (lets say TC0 and TC2) and a TC in the middle (TC1) is executing other instructions and hitting out of the cache, in a pathological case, it is possible that TC0's cahceops will keep getting rolled back and TC2's cacheops will complete. However, the processor is not deadlocked or livelocked and is making forward progress on other TCs. In such situations, it is possible to see temporary starvation of a TC.

*Chapter 3*

# Floating-Point Unit of the 34Kf™ Core

This chapter describes the MIPS64® Floating-Point Unit (FPU) included in the 34Kf core. This chapter contains the following sections:



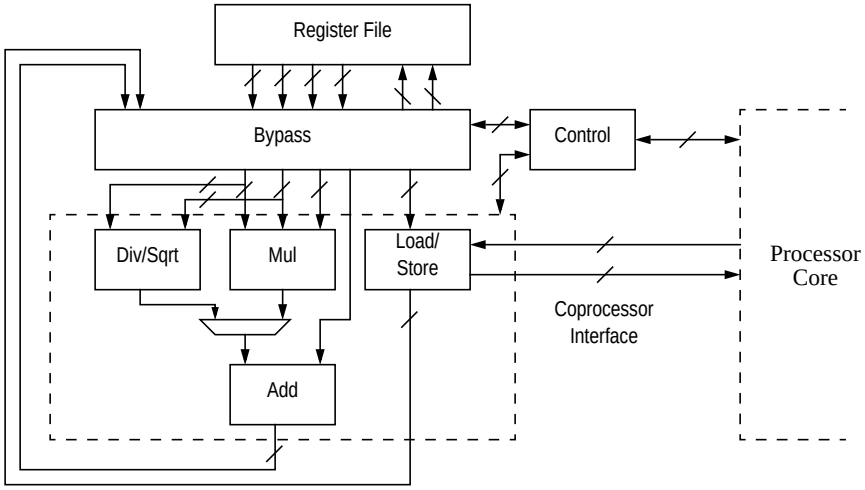
## 3.1 Features Overview

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.
- 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 34K 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

The floating-point unit on the 34K family of cores has only one set of registers, so it may only be used by one TC at any given time, by setting the TCU1 bit in the *TCStatus* register of that TC. It is the operating system's responsibility to make sure that only one TC owns the Floating-Point coprocessor as described above. Any attempt to execute a floating-point instruction by a TC that does not own the Floating-Point Coprocessor, 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 |

**Table 3.1 Parameters of Floating-Point Data Types**

| Parameter | Single | Double |
|---|---|---|
| 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 Figure 3.2 and Figure 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)**

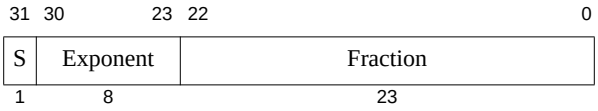
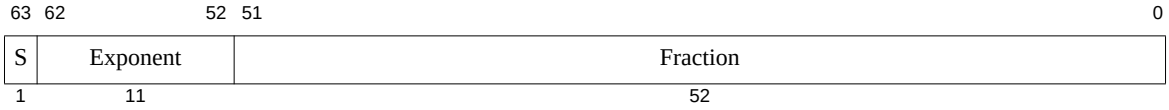



**Figure 3.3 Double-Precision Floating-Point Format (D)**




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[1] | 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 |

1. 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.

**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)**

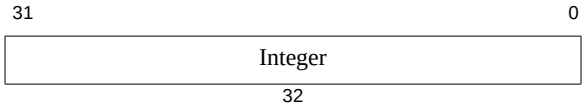
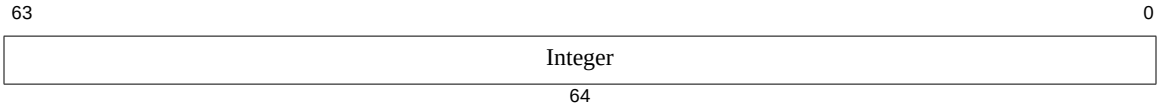



**Figure 3.5 Longword Fixed-Point Format (L)**




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.

# 3.4 Floating-Point General Registers

This section describes the organization and use of the Floating-Point general Registers (FPRs). The FPU is a 64b FPU, but a 32b register mode for backwards compatibility is also supported. The FR bit in the CP0 *Status* register determines which mode is selected:

- When the FR bit is a 1, the 64b 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 32b 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.

Figure 3.6 and Figure 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**

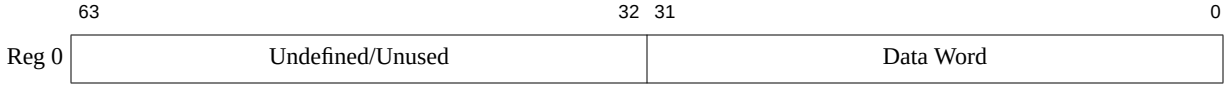
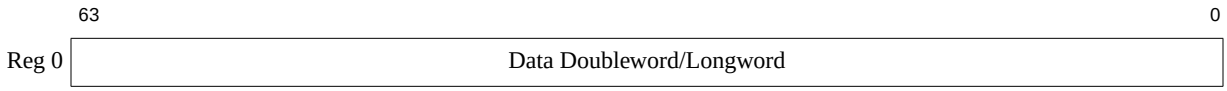



**Figure 3.7 Double Floating-Point or Longword Fixed-Point Operand in an FPR**




## 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.

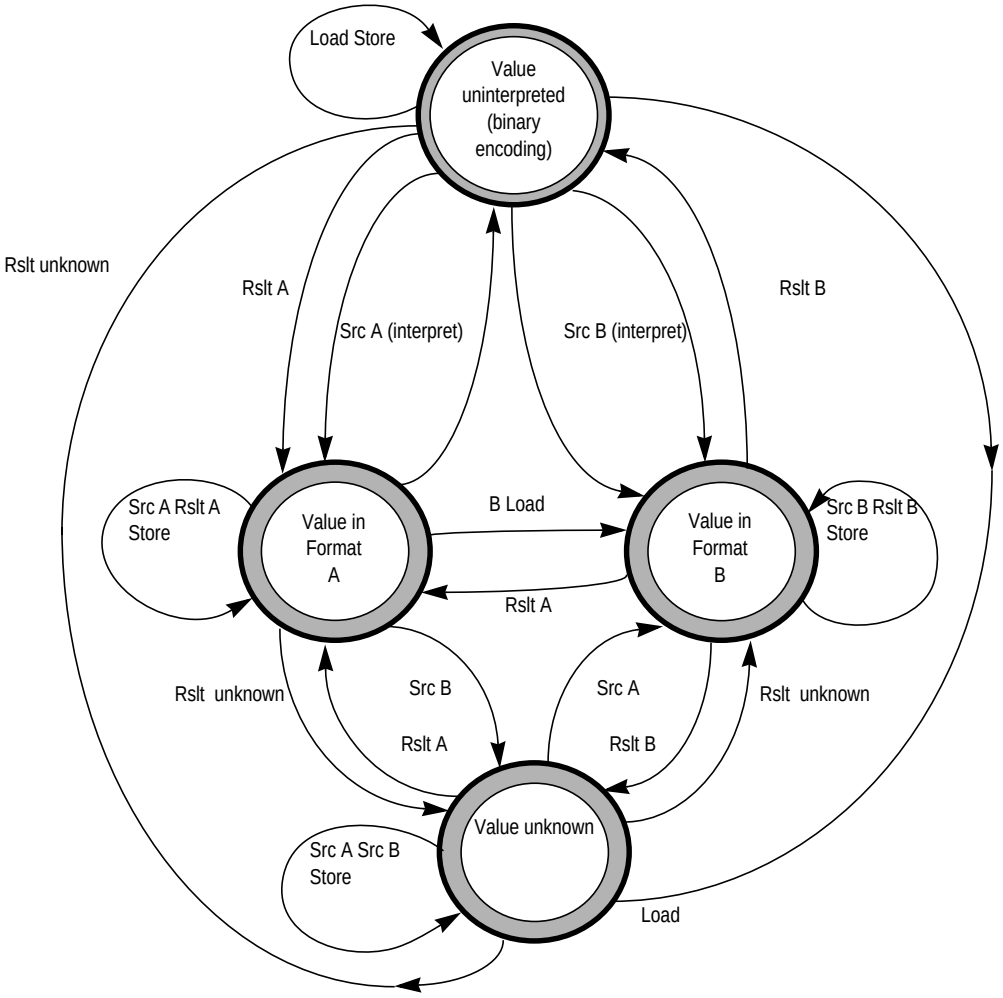
When an FPR with an *uninterpreted* value is used as a source operand by an instruction that requires a value of format *fmt*, the binary contents are interpreted as an encoded value in format *fmt*, and the value in the FPR changes to a value of format *fmt*. The binary contents cannot be reinterpreted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

**Figure 3.8 Effect of FPU Operations on the Format of Values Held in FPRs**




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

### 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**

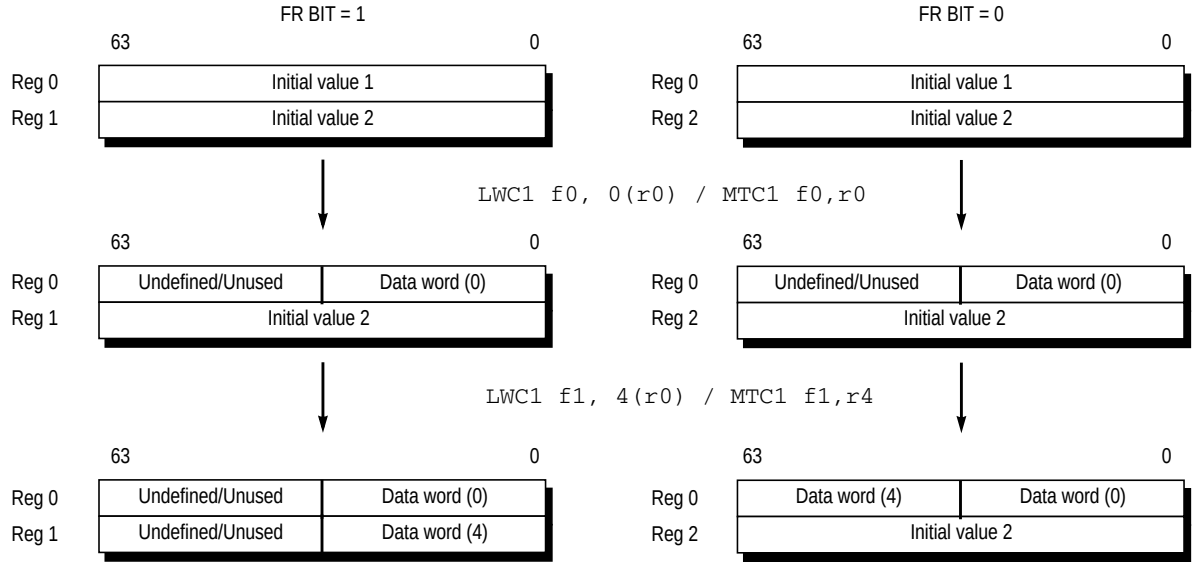
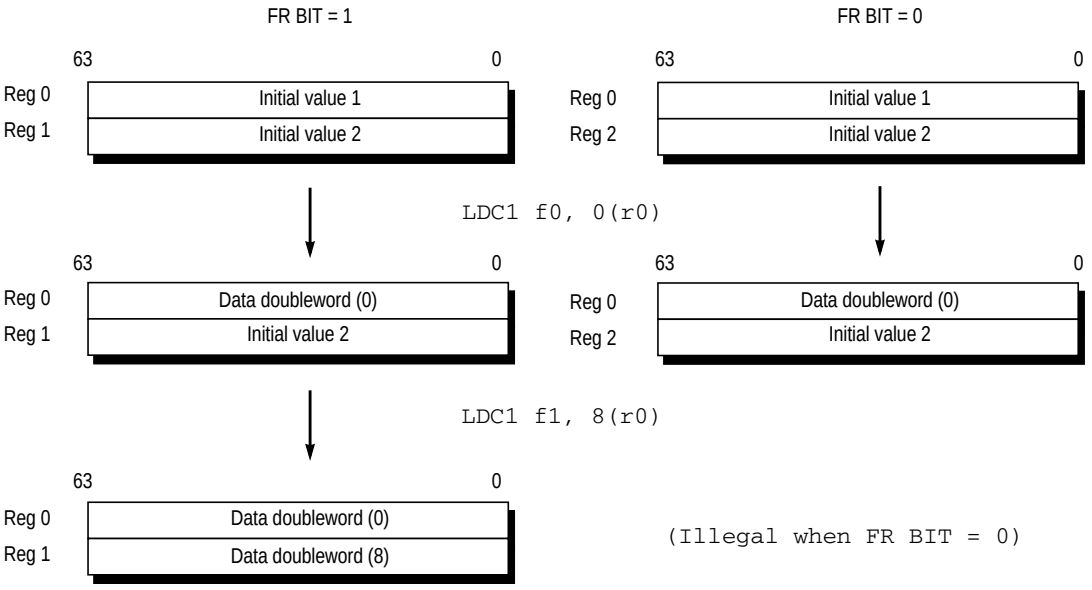



**Figure 3.10 FPU Doubleword Load and Move-to Operations**



## 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**

| Read/Write Notation | Hardware Interpretation | Software Interpretation |
|---|---|---|
| R/W | All bits in this field are readable and writable by software and potentially by hardware.<br>Hardware updates of this field are visible by software reads. Software updates of this field are visible by hardware reads.<br>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. | |
| R | This field is either static or is updated only by hardware.<br>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.<br>If the Reset State of this field is "Undefined", hardware updates this field only under those conditions specified in the description of the field. | A field to which the value written by software is ignored by hardware. Software may write any value to this field without affecting hardware behavior. Software reads of this field return the last value updated by hardware.<br>If the Reset State of this field is "Undefined," software reads of this field result in an UNPREDICTABLE value except after a hardware update done under the conditions specified in the description of the field. |
| 0 | Hardware does not update this field. Hardware can assume a zero value. | 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.<br>If the Reset State of this field is "Undefined," software must write this field with zero before it is guaranteed to read as zero. |

### 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.

**Figure 3.11 FIR Format**

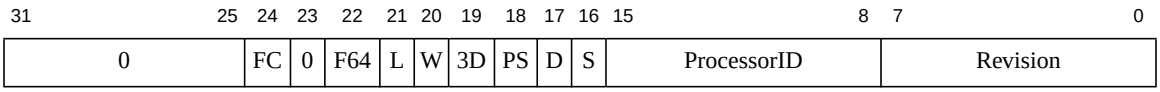



**Table 3.6 FIR Bit Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| FC | 24 | Indicates that full convert ranges are implemented:<br>0: Full convert ranges not implemented<br>1: Full convert ranges implemented<br>This bit is always 1 to indicate that full convert ranges are implemented. This means that all numbers can be converted to another type by the FPU (If FS bit in FCSR is not set Unimplemented Operation exception can still happen on denormal operands though). | R | 1 |
| F64 | 22 | Indicates that this is a 64-bit FPU:<br>0: Not a 64-bit FPU<br>1: A 64-bit FPU.<br>This bit is always 1 to indicate that this is a 64-bit FPU. | R | 1 |
| L | 21 | Indicates that the long fixed point (L) data type and instructions are implemented:<br>0: Long type not implemented<br>1: Long implemented<br>This bit is always 1 to indicate that long fixed point data types are implemented. | R | 1 |
| W | 20 | Indicates that the word fixed point (W) data type and instructions are implemented:<br>0: Word type not implemented<br>1: Word implemented<br>This bit is always 1 to indicate that word fixed point data types are implemented. | R | 1 |
| 3D | 19 | Indicates that the MIPS-3D ASE is implemented:<br>0: MIPS-3D not implemented<br>1: MIPS-3D implemented<br>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:<br>0: PS floating-point not implemented<br>1: PS floating-point implemented<br>This bit is always 0 to indicate that paired-single floating-point data types are not implemented. | R | 0 |

**Table 3.6 FIR Bit Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| D | 17 | Indicates that the double-precision (D) floating-point data type and instructions are implemented:<br>0: D floating-point not implemented<br>1: D floating-point implemented<br>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:<br>0: S floating-point not implemented<br>1: S floating-point implemented<br>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 | 0x93 |
| 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 | 31:25, 23 | 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.

**Figure 3.12 FCCR Format**

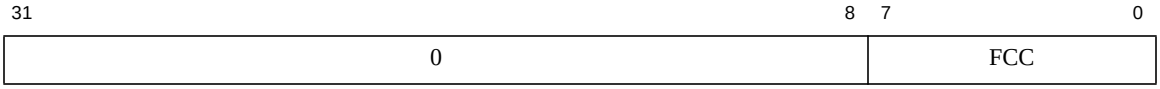



**Table 3.7 FCCR Bit Field Descriptions**

| 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.

**Figure 3.13 FEXR Format**

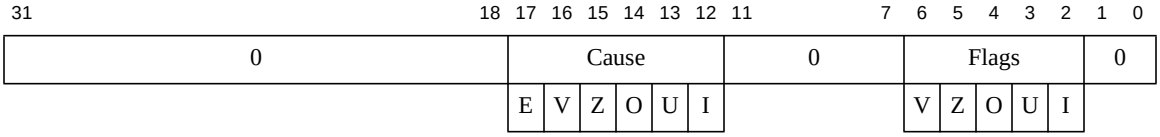



**Table 3.8 FEXR Bit Field Descriptions**

| 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.

**Figure 3.14 FENR Format**

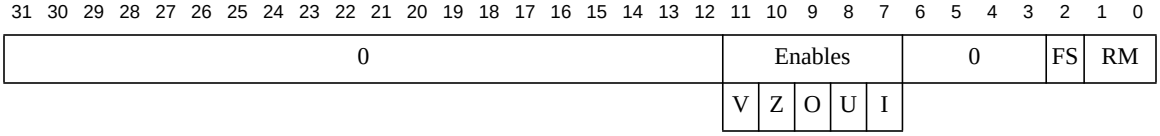



**Table 3.9 FENR Bit Field Descriptions**

| 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 |

**Table 3.9 FENR Bit Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| 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.

**Figure 3.15 FCSR Format**

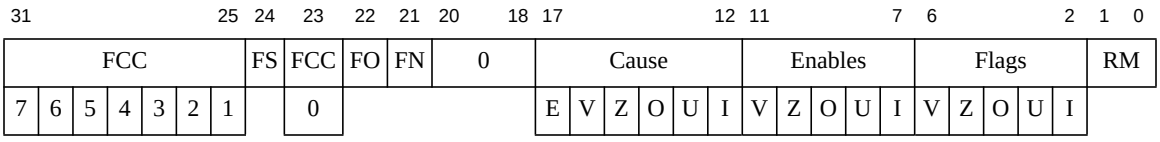


**Table 3.10 FCSR Bit Field Descriptions**

| 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.<br>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.<br>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.<br>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.<br>Hardware never resets this field; software must explicitly reset this field.<br>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).<br>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<br>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<br>Rounds the result to the value closest to but not greater in magnitude than the result. |
| 2 | RP - Round Towards Plus Infinity<br>Rounds the result to the value closest to but not less than the result. |
| 3 | RM - Round Towards Minus Infinity<br>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[1]

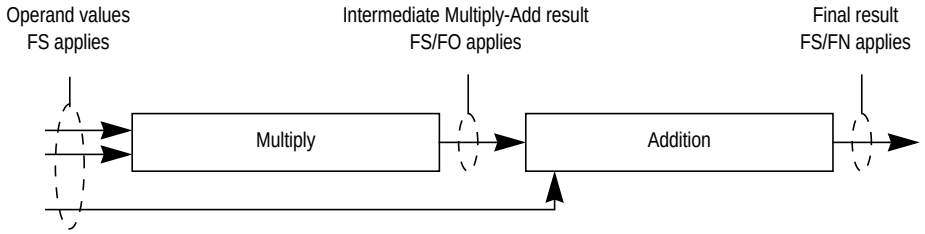
FO bit: MADD, MSUB, NMADD, and NMSUB

1. For ABS, C.cond, and NEG, denormal input operands or tiny results doe not result in Unimplemented exceptions when FS = 0. Flushing to zero nonetheless is implemented when FS = 1 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. |

**Table 3.14 Handling of Denormalized Operand Values and Tiny Results Based on FS Bit Setting**

| FS Bit | Handling of Denormalized Operand Values |
|---|---|
| 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.

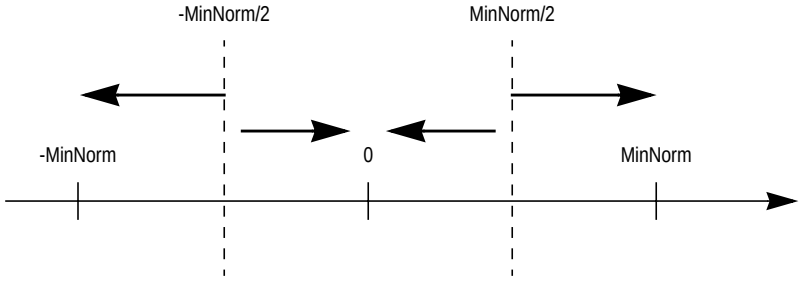
**Table 3.15 Handling of Tiny Intermediate Result Based on the FO and FS Bit Settings**

| 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 3.32) 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. |

**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 | 1 | Final result is forced to the value that would have been delivered for an untrapped underflow (see Table 3.32) 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 embedded applications. High performance on denormal operands and tiny results. |
| 1 | 1 | 1 | Highest accuracy and performance configuration.[1] |

1. 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 15, “34K™ Processor Core Instructions”, 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 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 3.20 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 Move To and From Instructions**

| Mnemonic | Instruction |
|---|---|
| CFC1 | Move Control Word From Floating Point |
| CTC1 | Move Control Word 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 3.21 lists the FPU IEEE compliant arithmetic operations.

**Table 3.21 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 3.22 lists the FPU-approximate arithmetic operations.

**Table 3.22 FPU-Approximate Arithmetic Operations**

| Mnemonic | Instruction |
|---|---|
| RECIP.fmt | Floating-Point Reciprocal Approximation |

**Table 3.22 FPU-Approximate Arithmetic Operations (Continued)**

| Mnemonic | Instruction |
|---|---|
| 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 3.23. 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 3.23 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.

In general, the conversion instructions only 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 3.24 and Table 3.25 list the FPU conversion instructions according to their rounding mode.

**Table 3.24 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 3.25 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 |

**Table 3.25 FPU Conversion Operations Using a Directed Rounding Mode (Continued)**

| Mnemonic | Instruction |
|---|---|
| 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 *MIPS32 Architecture Reference Manual, Volume II*.)

Table 3.26 through Table 3.28 list the formatted operand-value move instructions.

**Table 3.26 FPU Formatted Operand Move Instruction**

| Mnemonic | Instruction |
|---|---|
| MOV.fmt | Floating-Point Move |

**Table 3.27 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 3.28 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 3.29 lists the conditional branch (branch and branch likely) FPU instructions; Table 3.30 lists the deprecated conditional branch likely instructions.

**Table 3.29 FPU Conditional Branch Instructions**

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

**Table 3.30 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 MIPS32 architecture defines various miscellaneous instructions that conditionally move one CPU general register to another, based on an FPU condition code.

Table 3.31 lists these conditional move instructions.

**Table 3.31 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 3.32). 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 3.32 summarizes the default results.

**Table 3.32 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:<br>0 (RN) and 1 (RZ): Supplies a zero with the sign of the exact result.<br>2 (RP): For positive underflow values, supplies $2^{E\_min}$ (MinNorm). For negative underflow values, supplies a positive zero.<br>3 (RM): For positive underflow values, supplies a negative zero. For negative underflow values, supplies a negative $2^{E\_min}$ (MinNorm).<br>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 underflowed result. |
| O | Overflow | Depends on the rounding mode, as shown below:<br>0 (RN): Supplies an infinity with the sign of the exact result.<br>1 (RZ): Supplies the format's largest finite number with the sign of the exact result.<br>2 (RP): For positive overflow values, supplies positive infinity. For negative overflow values, supplies the format's most negative finite number.<br>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 $(+\infty) + (-\infty)$ or $(-\infty) - (-\infty)$.
- Multiplication: $0 \times \infty$, with any signs.
- Division: 0/0 or $\infty/\infty$, 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 $\pm 2^{E\_min}$.

- 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 (but the condition is effectively masked for all operations when FS=1). 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.

## 3.8 Pipeline and Performance

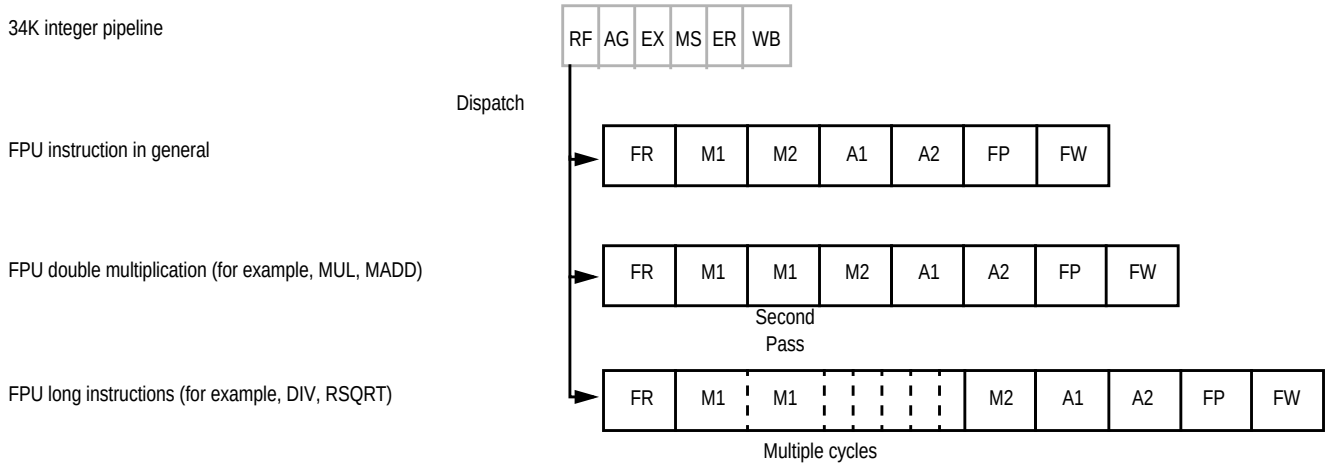
This section describes the structure and operation of the FPU pipeline.

### 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 34K integer pipeline. The FPU can be built to run at either the same frequency as the integer core or at one-half the frequency of the integer core.

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.

**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.

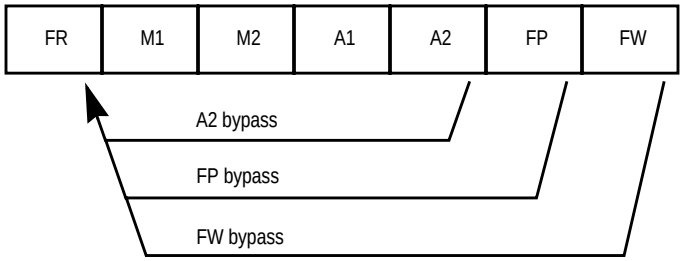
#### 3.8.1.7 FW Stage - Register Write

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 32-bit 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 3.33 shows the repeat rate and latency for the FPU instructions. Note that cycles related to floating point operations are listed in terms of FPU clocks.

**Table 3.33 34Kf Core FPU Latency and Repeat Rate**

| Opcode[1] | 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 |
| 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, MFC1 | 2 | 1 |

1. Format: S = Single, D = Double, W = Word, L = Longword.

*Chapter 4*

# The MIPS® DSP Application-Specific Extension to the MIPS32® Instruction Set

The 34K core includes support for the DSP ASE to accelerate a wide range of signal processing applications. Refer to Volume IV-e of the *MIPS32® Architecture Reference Manual* for a general description of the DSP ASE as well as detailed instruction descriptions. More detailed programming information is contained in the DSP chapter of *Programming the 34K Core Family.*

## 4.1 Additional Register State for the DSP ASE

The DSP ASE adds four additional registers, three accumulator registers and a control register, per TC in 34K line of processors. These registers require the operating system to recognize the presence of the DSP ASE and to include these additional registers in the context save and restore operation.

### 4.1.1 Additional HI-LO Registers

Three additional *HI-LO* registers, which together with the existing one would comprise a total of four accumulator registers. Many common DSP computations are accumulate functions, for example, the filter operation, convolution, etc. The *HI-LO* accumulator in the MIPS architecture would be the destination for such instructions. The instructions that target the accumulators use 2 bits to specify the destination accumulator, with the zero value referring to the original accumulator.

### 4.1.2 DSP Control Register

A control register *DSPControl* used to hold extra state bits needed for efficient support of the DSP instructions. Figure 4.1 illustrates the bits in this register. Table 4.1 describes the use of the various bits and the instructions that refer the fields. Table 4.2 lists the instructions that affect the *ouflag* field.

**Figure 4.1 MIPS32® DSP ASE Control Register (*DSPControl*) Format**

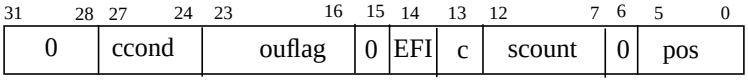



**Table 4.1 MIPS® DSP ASE Control Register (*DSPControl*) Field Descriptions**

| Fields | | Description | Read / Write | Reset State | Compliance |
|---|---|---|---|---|---|
| Name | Bits | | | | |
| 0 | 31:28 | Not used in the MIPS32 architecture, but these are reserved bits since they are used in the MIPS64 architecture. Must be written as zero; returns zero on read. | 0 | 0 | Required |

**Table 4.1 MIPS® DSP ASE Control Register (*DSPControl*) Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State | Compliance |
|---|---|---|---|---|---|
| Name | Bits | | | | |
| ccond | 27:24 | Condition code bits set after a compare instruction. This instruction will set the right-most bits as required by the number of elements in the vector compare. The bits not set by the instruction remain unchanged. | R/W | 0 | Required |
| ouflag | 23:16 | This field is written by hardware when certain instructions overflow or underflow and may have been saturated if the saturate flag was turned on. See Table 4.2 for a full list of which bits are set by what instructions. | R/W | 0 | Required |
| EFI | 14 | Extract Fail Indicator. This bit is set to 1 when an EXTP, EXTPV, EXTPDP, or EXTPDP fails. These instructions fail when there are insufficient bits to extract, that is, the value of pos in DSPControl is less than the value of size specified in the instruction. This bit is not sticky, that is, each invocation of one of those four instructions will reset the bit depending on whether or not the instruction failed. | R/W | 0 | Required |
| c | 13 | Carry bit set and used by a special add instruction used to implement a 64-bit add across two GPRs. Instruction ADDSC sets the bit and instruction ADDWC uses this bit. | R/W | 0 | Required |
| scount | 12:7 | This field is for use by the INSV instruction. The value of this field is used to specify the size of the bit field to be inserted. | R/W | 0 | Required |
| pos | 5:0 | This field is used by the variable insert instructions INSV to specify the insert position.<br>It is also used to indicate the extract position for the EXTP, EXTPV, EXTPDP, and EXTPDPV instructions. The decrement pos (DP) variants of these instructions on completion will have decremented the value of pos (by the size amount).<br>The MTHLIP instruction will increment the pos value by 32 after copying the value of *LO* to *HI*. | R/W | 0 | Required |
| 0 | 15:13 | Must be written as zero; returns zero on read. | 0 | 0 | Reserved |

The bits of the overflow flag ouflag field in the *DSPControl* register are set by a number of instructions. These bits are sticky and can be reset only by an explicit write to these bits in the register (using the WRDSP instruction). The table shows which bits can be set by which instructions and under what conditions.

**Table 4.2 The Instructions that Set the ouflag bits in *DSPControl***

| Bit Number | Which Instruction Sets This Bit |
|---|---|
| 16 | When the destination is accumulator (*HI-LO* pair) zero, and an operation overflow or underflow occurs, then this bit is set. Such instructions are: DPAQ_S, DPSQ_S, MULSAQ_S, MAQ_S. |
| 17 | Instructions as above, when the destination is accumulator (*HI-LO* pair) one. |

**Table 4.2 The Instructions that Set the ouflag bits in *DSPControl***

| Bit Number | Which Instruction Sets This Bit |
|---|---|
| 18 | Instructions as above, when the destination is accumulator (*HI-LO* pair) two. |
| 19 | Instructions as above, when the destination is accumulator (*HI-LO* pair) three. |
| 20 | Instructions that on a overflow/underflow will set this bit are: ADDQ, ADDQ_S, SUBQ, SUBQ_S, ADDU, ADDU_S, SUBU, SUBU_S, ABSQ, ABSQ_S, and ADDWC. |
| 21 | Instructions that on a overflow/underflow will set this bit are: MULQ_RS, MULEQ_S, and MULEU_S. |
| 22 | Instructions that on a overflow/underflow will set this bit are: SHLL, SHLLV, SHLL_S, SHLLV_S, PRECRQU_S. PRECRQ_RS. |
| 23 | Instructions that on a overflow/underflow will set this bit are: EXTR, EXTL, and variants. |

## 4.2 Software Detection of the DSP ASE

In the *Config3* CP0 register, bit 10 (DSPP), “DSP Present” is a static bit used to indicate the presence of the MIPS DSP ASE in the 34K core, as shown in Section 7.2.40 “Config3 Register (CP0 Register 16, Select 3)”. Software may query the DSPP bit to check whether this processor has implemented the MIPS DSP ASE. The DSPP bit is a 1 on the 34K core, since the DSP logic is always present.

Another bit, “DSP ASE Enable” bit 24 (MX) in the CP0 *Status* register, is a read/write bit used to enable access to the extra instructions defined by the DSP ASE as well as the MTLO/HI, MFLO/HI that access accumulators ac1, ac2, and ac3. The *Status* register is described in Section 7.2.29 “Status Register (CP0 Register 12, Select 0)”. Executing a DSP ASE instruction or the flavor of Move instruction described above with this bit set to zero causes a DSP State Disabled Exception. This uses exception code 26 in the CP0 *Cause* register. This allows the OS to do lazy context-switching. Table 7.33 shows the *Cause* Register exception code fields.

*Chapter 5*

# Memory Management of the 34K™ Core

The 34K processor core includes a Memory Management Unit (MMU) that interfaces between the execution unit and the cache controller. The core contains either a Translation Lookaside Buffer (TLB) or a simpler Fixed Mapping (FM) style MMU, specified as a build-time option when the core is implemented.

This chapter contains the following sections:



## 5.1 Introduction

The MMU in a 34K processor core will translate any virtual address to a physical address before a request is sent to the cache controllers for tag comparison or to the bus interface unit for an external memory reference. This translation is a very useful feature for operating systems when trying to manage physical memory to accommodate multiple tasks active in the same memory, possibly on the same virtual address but of course in different locations in physical memory. Other features handled by the MMU are protection of memory areas and defining the cache protocol.

Each VPE in the core features its own MMU. By default, the MMU is TLB based and each VPE will have a 16/32/64 dual-entry fully associative Joint TLB (JTLB). Optionally one or both of the VPEs can have a simple MMU that translates addresses through a Fixed Mapping (FM) mechanism. When at least one of the VPEs uses a TLB based MMU, two micro TLB arrays will also be instantiated to cache the latest translations in a smaller and faster array. The instruction micro TLB (ITLB) has 3 shared entries, plus one private entry per TC. The data micro TLB (DTLB) always contains 8 entries. When an address is translated, the appropriate micro TLB (ITLB or DTLB) is accessed first. If the translation is not found in the micro TLB, the JTLB is accessed. If there is a miss in the JTLB, an exception is taken.

Figure 5.1 shows how the memory management unit interacts with cache accesses with a TLB, while Figure 5.2 shows the equivalent for the FM MMU.

**Figure 5.1 Address Translation During a Cache Access with TLB MMU**

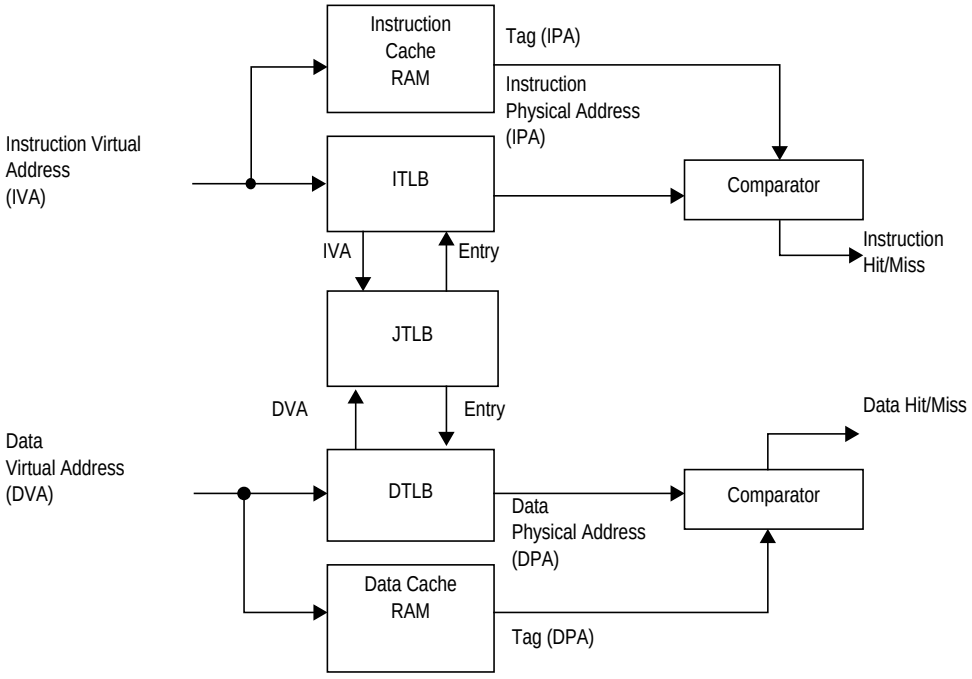
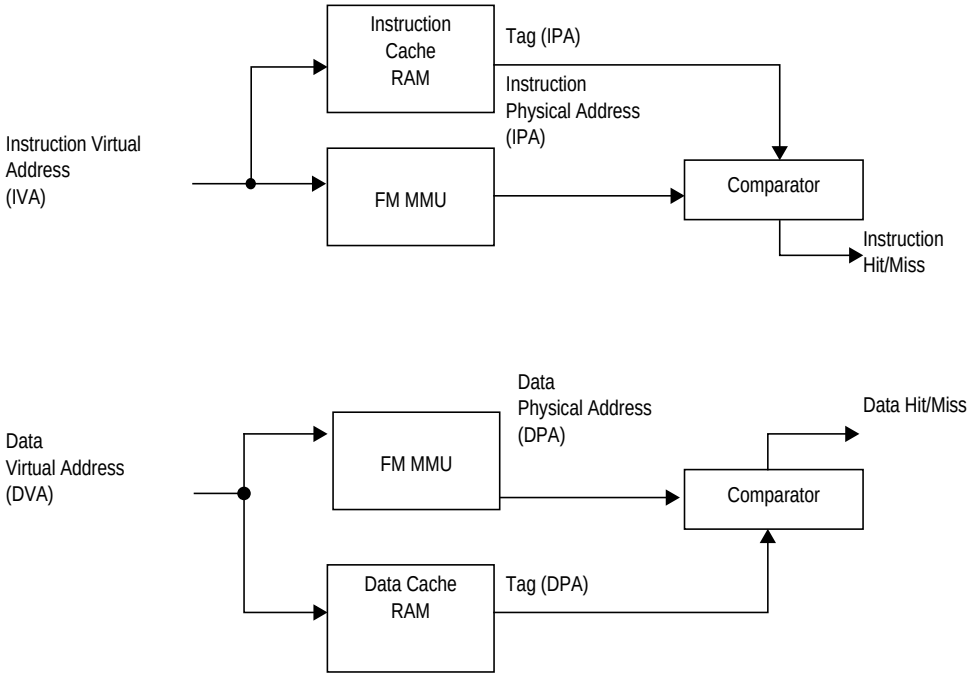



**Figure 5.2 Address Translation During a Cache Access with FM MMU**



## 5.2 Modes of Operation

A 34K processor core supports four modes of operation:

- User mode
- Supervisor mode (only w/ TLB)
- Kernel mode
- Debug mode

User mode is most often used for application programs. Supervisor mode has an intermediate privilege level with access to an additional region of memory and is only supported with the TLB-based MMU. Kernel mode is typically used for handling exceptions and privileged operating system functions, including CP0 management and I/O device accesses. Debug mode is used for software debugging and most likely occurs within a software development tool.

The address translation performed by the MMU depends on the mode in which the processor is operating.

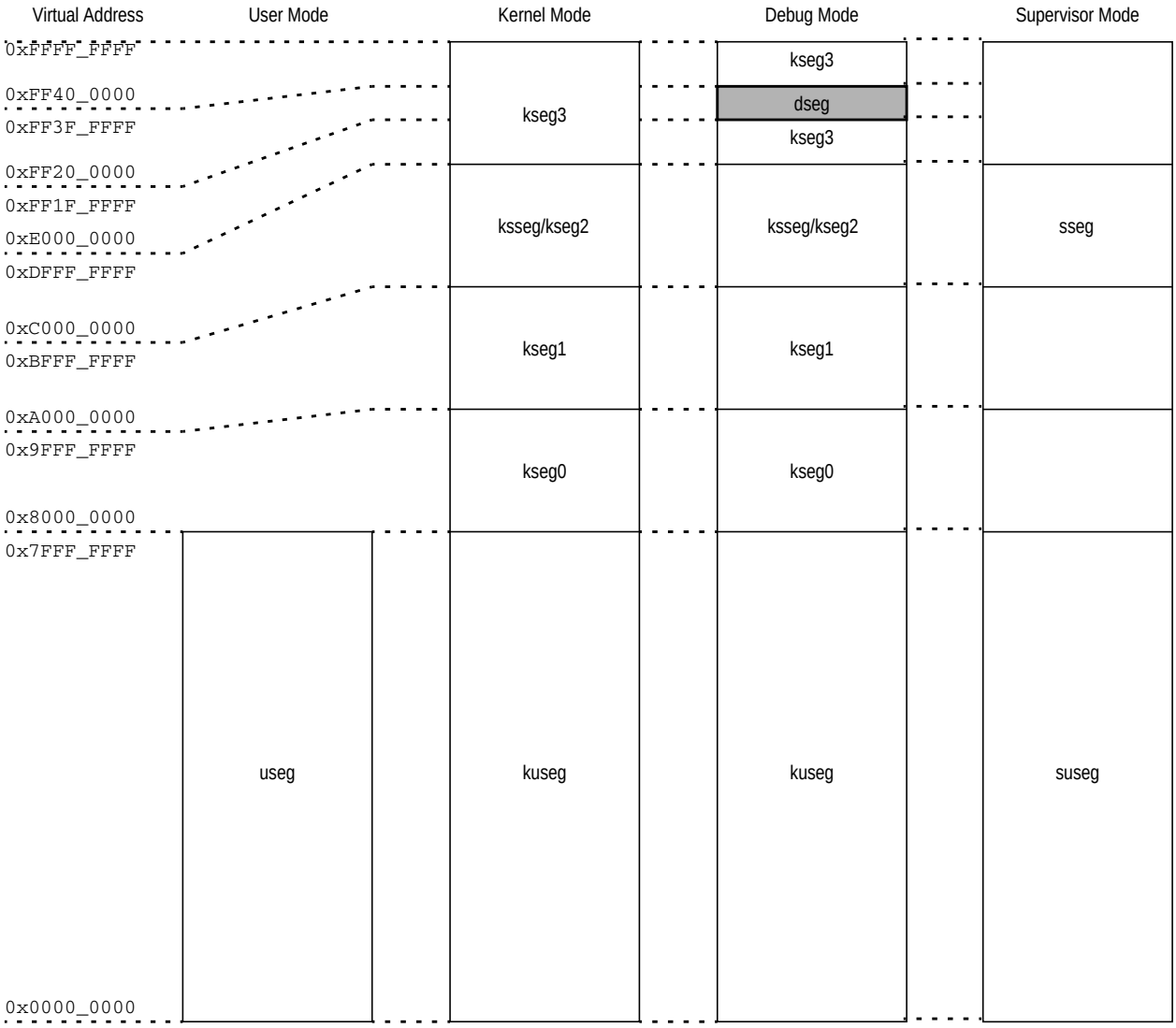
### 5.2.1 Virtual Memory Segments

The Virtual memory segments are different depending on the mode of operation. Figure 5.3 shows the segmentation for the 4 GByte ($2^{32}$ bytes) virtual memory space addressed by a 32-bit virtual address, for the four modes of operation.

The core enters Kernel mode both at reset and when an exception is recognized. While in Kernel mode, software has access to the entire address space, as well as all CP0 registers. User mode accesses are limited to a subset of the virtual address space (0x0000_0000 to 0x7FFF_FFFF) and can be inhibited from accessing CP0 functions. In User mode, virtual addresses 0x8000_0000 to 0xFFFF_FFFF are invalid and cause an exception if accessed. Supervisor mode adds access to sseg (0xC000_0000 to 0xDFFF_FFFF). kseg0, kseg1, and kseg3 will still cause exceptions if they are accessed.

Debug mode is entered on a debug exception. While in Debug mode, the debug software has access to the same address space and CP0 registers as for Kernel mode. In addition, while in Debug mode the core has access to the debug segment dseg. This area overlays part of the kernel segment kseg3. dseg access in Debug mode can be turned on or off, allowing full access to the entire kseg3 in Debug mode, if so desired.

**Figure 5.3 34K™ Processor Core Virtual Memory Map**




Each of the segments shown in Figure 5.3 are either mapped or unmapped. The following two sub-sections explain the distinction. Then sections Section 5.2.2 “User Mode”, Section 5.2.4 “Kernel Mode”, and Section 5.2.5 “Debug Mode” specify which segments are actually mapped and unmapped.

### 5.2.1.1 Unmapped Segments

An unmapped segment does not use the TLB or the FM to translate from virtual-to-physical addresses. Especially after reset, it is important to have unmapped memory segments, because the TLB is not yet programmed to perform the translation.

Unmapped segments have a fixed simple translation from virtual to physical address. This is much like the translations the FM provides for the core, but we will still make the distinction.

Except for kseg0, unmapped segments are always uncached. The cacheability of kseg0 is set in the K0 field of the CP0 register *Config* (see Section 7.2.37 “Config Register (CP0 Register 16, Select 0)”).

#### 5.2.1.2 Mapped Segments

A mapped segment does use the TLB or the FM to translate from virtual-to-physical addresses.

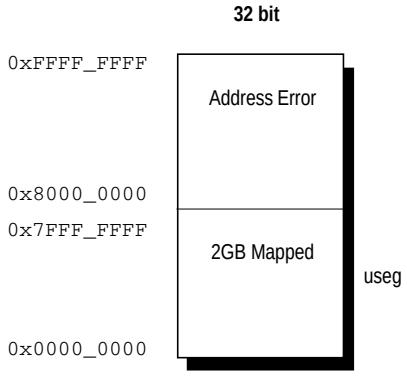
For the core with TLB, the translation of mapped segments is handled on a per-page basis. Included in this translation is information defining whether the page is cacheable or not, and the protection attributes that apply to the page.

For the core with the FM MMU, the mapped segments have a fixed translation from virtual to physical address. The cacheability of the segment is defined in the CP0 register *Config*, fields K23 and KU (see Section 7.2.37 "Config Register (CP0 Register 16, Select 0)"). Write protection of segments is not possible during FM translation.

### 5.2.2 User Mode

In user mode, a single 2 GByte ($2^{31}$ bytes) uniform virtual address space called the user segment (useg) is available. Figure 5.4 shows the location of user mode virtual address space.

**Figure 5.4 User Mode Virtual Address Space**




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

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

- KSU = 2#10
- EXL = 0
- ERL = 0

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

Table 5.1 lists the characteristics of the User mode segment.

**Table 5.1 User Mode Segments**

| Address Bit Value | Status Register Bit Value | | | Segment Name | Address Range | Segment Size |
|---|---|---|---|---|---|---|
| | EXL | ERL | KSU | | | |
| 32-bit<br>A(31) = 0 | 0 | 0 | 2#10 | useg | 0x0000_0000 --><br>0x7FFF_FFFF | 2 GByte<br>($2^{31}$ bytes) |

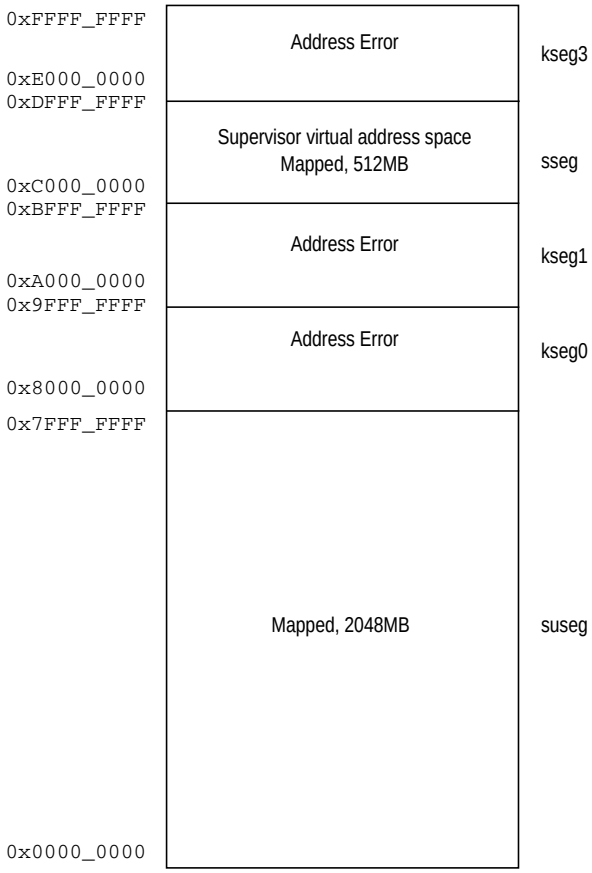
All valid user mode virtual addresses have their most significant bit cleared to 0, indicating that user mode can only access the lower half of the virtual memory map. Any attempt to reference an address with the most significant bit set while in user mode causes an address error exception.

The system maps all references to *useg* through the TLB or FM. For cores with a TLB, the virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address before translation. Also bit settings within the TLB entry for the page determine the cacheability of a reference. For FM MMU cores, the cacheability is set via the KU field of the CP0 Config register.

### 5.2.3 Supervisor Mode

In supervisor mode, two virtual address spaces are available. A 2 GByte ($2^{31}$ bytes) uniform virtual address space called the user segment (useg) as well as the 512MB (ksseg) are available. Figure 5.5 shows the location of supervisor mode virtual address space.

**Figure 5.5 Supervisor Mode Virtual Address Space**




The user segment starts at address 0x0000_0000 and ends at address 0x7FFF_FFFF. The supervisor segment begins at 0xC000_0000 and ends at 0xDFFF_FFFF. Accesses to all other addresses cause an address error exception.

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

- KSU = 2#01
- EXL = 0
- ERL = 0

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

Table 5.1 lists the characteristics of the Supervisor mode segments.

**Table 5.2 Supervisor Mode Segments**

| Address Bit Value | Status Register Bit Value | | | Segment Name | Address Range | Segment Size |
|---|---|---|---|---|---|---|
| | EXL | ERL | KSU | | | |
| 32-bit<br>A(31) = 0 | 0 | 0 | 2#01 | suseg | 0x0000_0000 --><br>0x7FFF_FFFF | 2 GByte<br>($2^{31}$ bytes) |
| 32-bit<br>A(31:29) = $110_2$ | 0 | 0 | 2#01 | sseg | 0xC000_0000 -><br>0xDFFF_FFFF | 512MB<br>($2^{29}$ bytes) |

The system maps all references to *useg and ksseg* through the TLB or FM. For cores with a TLB, the virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address before translation. Also bit settings within the TLB entry for the page determine the cacheability of a reference. For FM MMU cores, the cacheability of useg and ksseg is set via the KU and K23 fields of the CP0 Config register respectively.

### 5.2.4 Kernel Mode

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

- KSU = 2#00
- ERL = 1
- EXL = 1

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

Kernel mode virtual address space is divided into regions differentiated by the high-order bits of the virtual address, as shown in Figure 5.6. Also, Table 5.3 lists the characteristics of the Kernel mode segments.

**Figure 5.6 Kernel Mode Virtual Address Space**

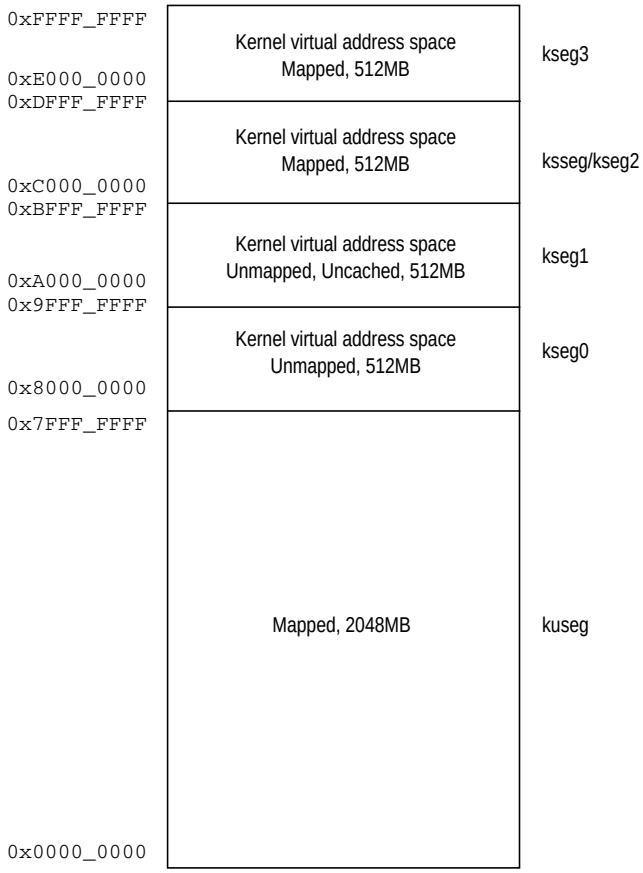


**Table 5.3 Kernel Mode Segments**

| Address Bit Values | Status Register Is One of These Values | | | Segment Name | Address Range | Segment Size |
|---|---|---|---|---|---|---|
| | KSU | EXL | ERL | | | |
| A(31) = 0 | (KSU = $00_2$ or EXL = 1 or ERL = 1) and DM = 0 | | | kuseg | 0x0000_0000 through 0x7FFF_FFFF | 2 GBytes ($2^{31}$ bytes) |
| A(31:29) = $100_2$ | | | | kseg0 | 0x8000_0000 through 0x9FFF_FFFF | 512 MBytes ($2^{29}$ bytes) |
| A(31:29) = $101_2$ | | | | kseg1 | 0xA000_0000 through 0xBFFF_FFFF | 512 MBytes ($2^{29}$ bytes) |
| A(31:29) = $110_2$ | | | | ksseg/kseg2 | 0xC000_0000 through 0xDFFF_FFFF | 512 MBytes ($2^{29}$ bytes) |
| A(31:29) = $111_2$ | | | | kseg3 | 0xE000_0000 through 0xFFFF_FFFF | 512 MBytes ($2^{29}$ bytes) |

#### 5.2.4.1 Kernel Mode, User Space (kuseg)

In Kernel mode, when the most-significant bit of the virtual address (A31) is cleared, the 32-bit kuseg virtual address space is selected and covers the full $2^{31}$ bytes (2 GBytes) of the current user address space mapped to addresses 0x0000_0000 - 0x7FFF_FFFF. For cores with TLBs, the virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

When ERL = 1 in the *Status* register, the user address region becomes a $2^{31}$-byte unmapped and uncached address space. While in this setting, the kuseg virtual address maps directly to the same physical address, and does not include the ASID field.

#### 5.2.4.2 Kernel Mode, Kernel Space 0 (kseg0)

In Kernel mode, when the most-significant three bits of the virtual address are $100_2$, 32-bit kseg0 virtual address space is selected; it is the $2^{29}$-byte (512-MByte) kernel virtual space located at addresses 0x8000_0000 - 0x9FFF_FFFF. References to kseg0 are unmapped; the physical address selected is defined by subtracting 0x8000_0000 from the virtual address. The K0 field of the *Config* register controls cacheability.

#### 5.2.4.3 Kernel Mode, Kernel Space 1 (kseg1)

In Kernel mode, when the most-significant three bits of the 32-bit virtual address are $101_2$, 32-bit kseg1 virtual address space is selected. kseg1 is the $2^{29}$-byte (512-MByte) kernel virtual space located at addresses 0xA000_0000 - 0xBFFF_FFFF. References to kseg1 are unmapped; the physical address selected is defined by subtracting 0xA000_0000 from the virtual address. Caches are disabled for accesses to these addresses, and physical memory (or memory-mapped I/O device registers) are accessed directly.

#### 5.2.4.4 Kernel Mode, Kernel/Supervisor Space 2 (ksseg/kseg2)

In Kernel mode, when KSU= $00_2$, ERL = 1, or EXL = 1 in the *Status* register, and DM = 0 in the *Debug* register, and the most-significant three bits of the 32-bit virtual address are $110_2$, 32-bit kseg2 virtual address space is selected. With the FM MMU, this $2^{29}$-byte (512-MByte) kernel virtual space is located at physical addresses 0xC000_0000 - 0xDFFF_FFFF. Otherwise, this space is mapped through the TLB.

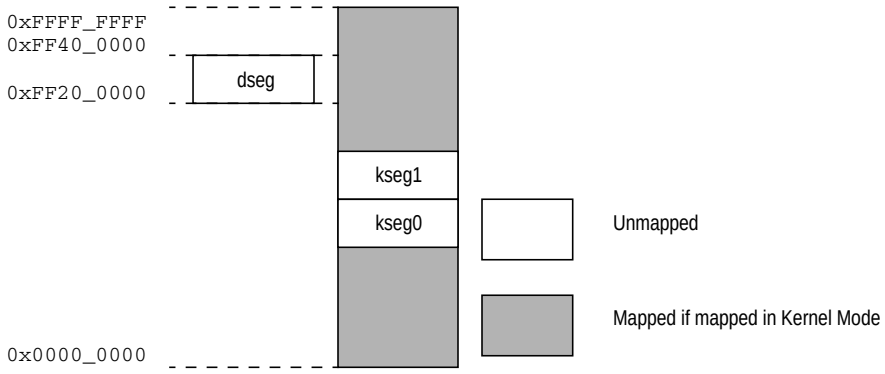
#### 5.2.4.5 Kernel Mode, Kernel Space 3 (kseg3)

In Kernel mode, when the most-significant three bits of the 32-bit virtual address are $111_2$ , the kseg3 virtual address space is selected. With the FM MMU, this $2^{29}$-byte (512-MByte) kernel virtual space is located at physical addresses 0xE000_0000 - 0xFFFF_FFFF. Otherwise, this space is mapped through the TLB.

### 5.2.5 Debug Mode

Debug mode address space is identical to Kernel mode address space with respect to mapped and unmapped areas, except for *kseg3*. In *kseg3*, a debug segment *dseg* co-exists in the virtual address range 0xFF20_0000 to 0xFF3F_FFFF. The layout is shown in Figure 5.7.

**Figure 5.7 Debug Mode Virtual Address Space**




The dseg is sub-divided into the dmseg segment at 0xFF20_0000 to 0xFF2F_FFFF which is used when the probe services the memory segment, and the drseg segment at 0xFF30_0000 to 0xFF3F_FFFF which is used when memory-mapped debug registers are accessed. The subdivision and attributes for the segments are shown in Table 5.4.

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

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

**Table 5.4 Physical Address and Cache Attributes for dseg, dmseg, and drseg Address Spaces**

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

### 5.2.5.1 Conditions and Behavior for Access to drseg, EJTAG Registers

The behavior of CPU access to the drseg address range at 0xFF30_0000 to 0xFF3F_FFFF is determined as shown in Table 5.5

**Table 5.5 CPU Access to drseg Address Range**

| Transaction | LSNM bit in Debug register | Access |
|---|---|---|
| Load / Store | 1 | Kernel mode address space (kseg3) |
| Fetch | Don’t care | drseg, see comments below |
| Load / Store | 0 | |

Debug software is expected to read the debug control register (DCR) to determine which other memory mapped registers exist in drseg. The value returned in response to a read of any unimplemented memory mapped register is unpredictable, and writes are ignored to any unimplemented register in the drseg. Refer to Chapter 11, “EJTAG Debug Support in the 34K™ Core” for more information on the DCR.

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

### 5.2.5.2 Conditions and Behavior for Access to dmseg, EJTAG Memory

The behavior of CPU access to the dmseg address range at 0xFF20_0000 to 0xFF2F_FFFF is determined by the table shown in Table 5.6..

**Table 5.6 CPU Access to dmseg Address Range**

| Transaction | ProbEn bit in DCR register | LSNM bit in Debug register | Access |
|---|---|---|---|
| Load / Store | Don’t care | 1 | Kernel mode address space (kseg3) |
| Fetch | 1 | Don’t care | dmseg |
| Load / Store | 1 | 0 | |
| Fetch | 0 | Don’t care | See comments below |
| Load / Store | 0 | 0 | |

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

# 5.3 Translation Lookaside Buffer

The following subsections discuss the TLB memory management scheme used in the 34Kc processor core. The TLB consists of the joint and micro address translation buffers:

- 16-64 dual-entry fully associative Joint TLB (JTLB) per VPE
- Fully associative Instruction micro TLB (ITLB) with 3 shared entries and 1 private entry per TC
- 8-entry fully associative Data micro TLB (DTLB)

## 5.3.1 Joint TLB

The 16-64 dual-entry, fully associative Joint TLB maps 32-128 virtual pages to their corresponding physical addresses. The purpose of the TLB is to translate virtual addresses and their corresponding ASID into a physical memory address. The translation is performed by comparing the upper bits of the virtual address (along with the ASID bits) against each of the entries in the *tag* portion of the JTLB structure. Because this structure is used to translate both instruction and data virtual addresses, it is referred to as a “joint” TLB.

The JTLB is organized as 16-64 pairs of even and odd entries containing descriptions of pages that range in size from 4-KBytes to 256MBytes into the 4-GByte physical address space.

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

Figure 5.8 shows the contents of one of the dual-entries in the JTLB. The bit range indication in the figure serves to clarify which address bits are (or may be) affected during the translation process.

**Figure 5.8 JTLB Entry (Tag and Data)**

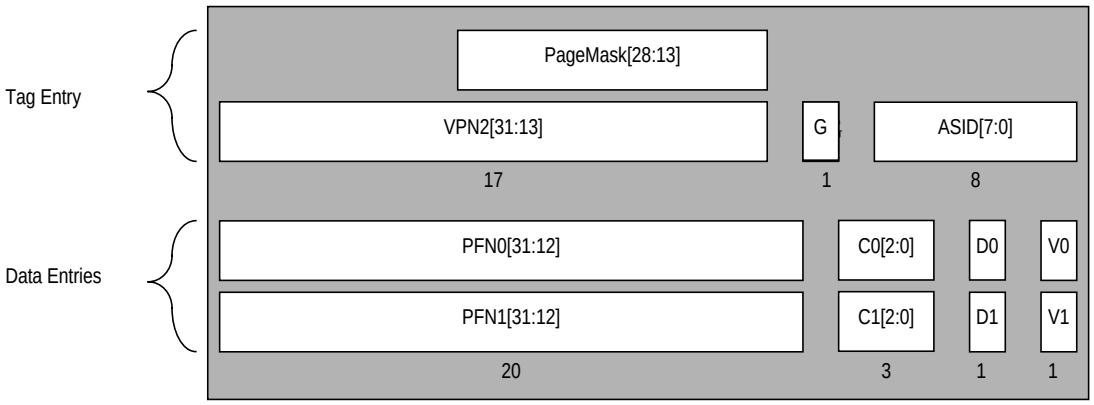


Table 5.7 and Table 5.8 explain each of the fields in a JTLB entry.

**Table 5.7 TLB Tag Entry Fields**

| Field Name | Description |
|---|---|
| PageMask[28:13] | Page Mask Value. The Page Mask defines the page size by masking the appropriate VPN2 bits from being involved in a comparison. It is also used to determine which address bit is used to make the even-odd page (PFN0-PFN1) determination. See the table below.<br><br>**PageMask / Page Size / Even/Odd Bank Select Bit**<br>00_0000_0000_0000_00 — 4KB — VAddr[12]<br>00_0000_0000_0000_11 — 16KB — VAddr[14]<br>00_0000_0000_0011_11 — 64KB — VAddr[16]<br>00_0000_0000_1111_11 — 256KB — VAddr[18]<br>00_0000_0011_1111_11 — 1MB — VAddr[20]<br>00_0000_1111_1111_11 — 4MB — VAddr[22]<br>00_0011_1111_1111_11 — 16MB — VAddr[24]<br>00_1111_1111_1111_11 — 64MB — VAddr[26]<br>11_1111_1111_1111_11 — 256MB — VAddr[28]<br><br>The PageMask column above shows all the legal values for PageMask. Because each pair of bits can only have the same value, the physical entry in the JTLB will only save a compressed version of the PageMask using only 8 bits. This is however transparent to software, which will always work with a 16 bit field |
| VPN2[31:13] | Virtual Page Number divided by 2. This field contains the upper bits of the virtual page number. Because it represents a pair of TLB pages, it is divided by 2. Bits 31:29 are always included in the TLB lookup comparison. Bits 28:13 are included depending on the page size, defined by PageMask |
| G | Global Bit. When set, indicates that this entry is global to all processes and/or threads and thus disables inclusion of the ASID in the comparison. |
| ASID[7:0] | Address Space Identifier. Identifies which process or thread this TLB entry is associated with. |

**Table 5.8 TLB Data Entry Fields**

| Field Name | Description |
|---|---|
| PFN0[31:12], PFN1[31:12] | Physical Frame Number. Defines the upper bits of the physical address. |

**Table 5.8 TLB Data Entry Fields (Continued)**

| Field Name | Description |
|---|---|
| C0[2:0], C1[2:0] | Cacheability. Contains an encoded value of the cacheability attributes and determines whether the page should be placed in the cache or not. The field is encoded as follows: |
| D0, D1 | "Dirty" or Write-enable Bit. Indicates that the page has been written and/or is writable. If this bit is set, stores to the page are permitted. If the bit is cleared, stores to the page cause a TLB Modified exception. |
| V0, V1 | Valid Bit. Indicates that the TLB entry and, thus, the virtual page mapping are valid. If this bit is set, accesses to the page are permitted. If the bit is cleared, accesses to the page cause a TLB Invalid exception. |

| C[2:0] | Coherency Attribute |
|---|---|
| 0 | Cacheable, noncoherent, write-through, no write-allocate |
| 1 | Reserved |
| 2 | Uncached |
| 3 | Cacheable, noncoherent, write-back, write-allocate |
| 4-6 | Reserved |
| 7 | Uncached Accelerated |

In order to fill an entry in the JTLB, software executes a TLBWI or TLBWR instruction (See Section 5.4.3 "TLB Instructions"). Prior to invoking one of these instructions, several CP0 registers must be updated with the information to be written to a TLB entry:

- PageMask is set in the CP0 *PageMask* register.
- VPN2, and ASID are set in the CP0 *EntryHi* register.
- PFN0, C0, D0, V0, and G bits are set in the CP0 *EntryLo0* register.
- PFN1, C1, D1, V1, and G bits are set in the CP0 *EntryLo1* register.

Note that the global bit "G" is part of both *EntryLo0* and *EntryLo1*. The resulting "G" bit in the JTLB entry is the logical AND between the two fields in *EntryLo0* and *EntryLo1*. Please refer to Chapter 7, "CP0 Registers of the 34K™ Core" for further details.

The address space identifier (ASID) helps to reduce the frequency of TLB flushing on a context switch. The existence of the ASID allows multiple processes to exist in both the TLB and instruction caches. The ASID value is stored in the *EntryHi* register and is compared to the ASID value of each entry.

## 5.3.2 Instruction TLB

The ITLB is a small fully associative TLB dedicated to perform translations for the instruction stream. The ITLB only maps 4-Kbyte pages/sub-pages or 1-Mbyte pages/sub-pages.

The ITLB is managed by hardware and is transparent to software. If a fetch address cannot be translated by the ITLB, the JTLB is accessed trying to translate it in the following clock cycles. If successful, the translation information is copied into the ITLB and bypassed to the tag comparators. This results in an ITLB miss penalty of at least 2 cycles. Depending on the JTLB implementation or if it is busy with other operations, it may take additional cycles.

The ITLB array consists of 3 shared entries and 1 private entry per TC. On an ITLB miss, the new translation will be loaded into the least recently used of the 3 shared ITLB entries. For each TC, if the displaced translation was more recently used than its private entry, the displaced translation will be written into the private entry.

### 5.3.3 Data TLB

The DTLB is a small 8-entry, fully associative TLB which provides a faster translation for Load/Store addresses than is possible with the JTLB. The DTLB only maps 4-Kbyte pages/sub-pages or 1-Mbyte pages/sub-pages.

Like the ITLB, the DTLB is managed by hardware and is transparent to software. For simultaneous ITLB and DTLB misses, the DTLB has priority and will access the JTLB first.

## 5.4 Virtual-to-Physical 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 VPN field of the entry, and either:

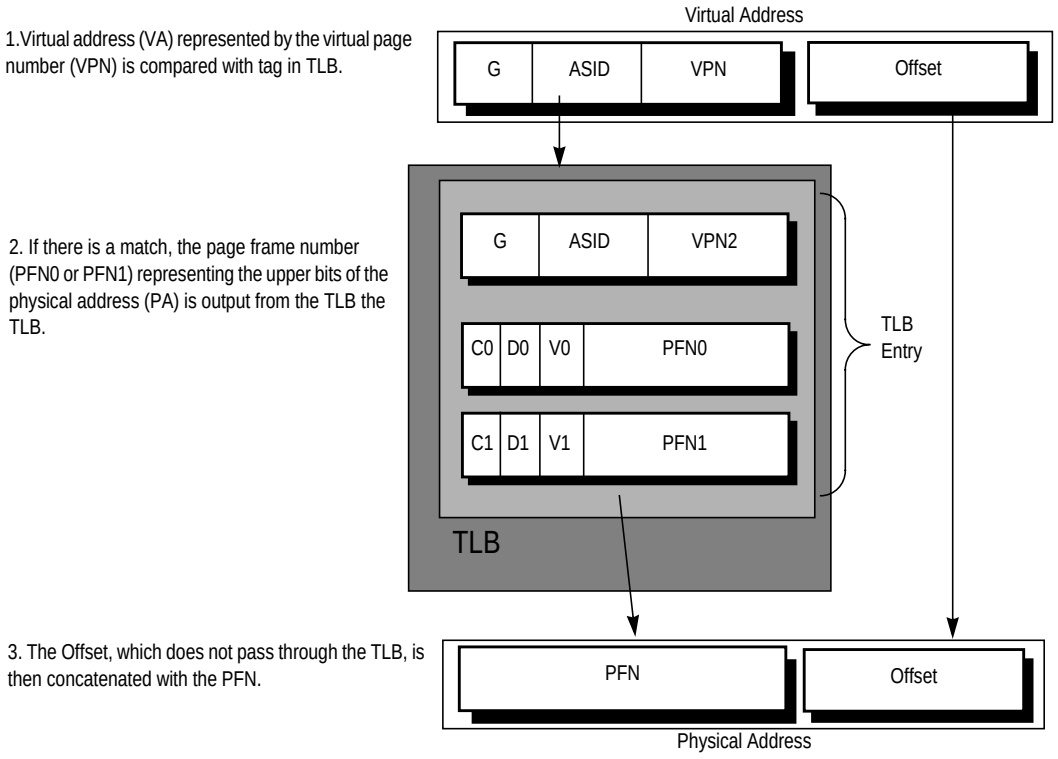
- The Global (G) bit of both the even and odd pages of the TLB entry are set, or
- The ASID field of the virtual address is the same as the ASID field of the TLB entry

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

Figure 5.9 shows the logical translation of a virtual address into a physical address.

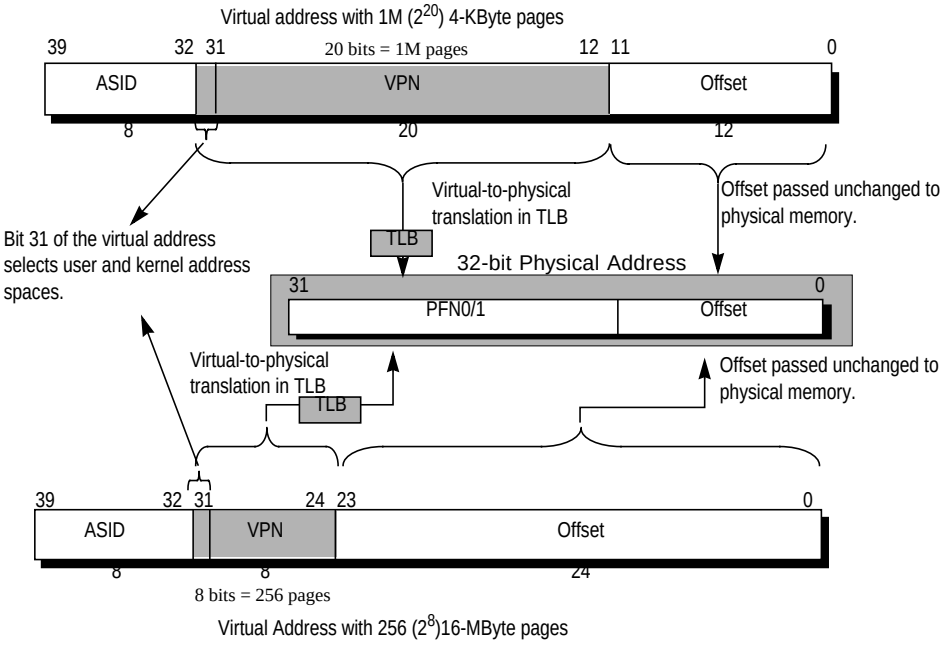
In this figure the virtual address is extended with an 8-bit ASID, which reduces the frequency of TLB flushing during a context switch. This 8-bit ASID contains the number assigned to that process and is stored in the CP0 *EntryHi* register.

**Figure 5.9 Overview of a Virtual-to-Physical Address Translation**




If there is a virtual address match in the TLB, the Physical Frame Number (PFN) is output from the TLB and concatenated with the *Offset*, to form the physical address. The *Offset* represents an address within the page frame space. As shown in Figure 5.9, the *Offset* does not pass through the TLB. Figure 5.10 shows a flow diagram of the address translation process for two page sizes. The top portion of the figure shows a virtual address for a 4 KByte page size. The width of the *Offset* is defined by the page size. The remaining 20 bits of the address represent the virtual page number (VPN). The bottom portion of Figure 5.10 shows the virtual address for a 16 MByte page size. The remaining 8 bits of the address represent the VPN.

**Figure 5.10 32-bit Virtual Address Translation**




## 5.4.1 Hits, Misses, and Multiple Matches

Each JTLB entry contains a tag and two data fields. If a match is found, the upper bits of the virtual address are replaced with the page frame number (PFN) stored in the corresponding entry in the data array of the JTLB. The granularity of JTLB mappings is defined in terms of TLB pages. The JTLB supports pages of different sizes ranging from 4KB to 256 MB in powers of 4. If a match is found, but the entry is invalid (i.e., the V bit in the data field is 0), a TLB Invalid exception is taken.

If no match occurs (TLB miss), an exception is taken and software refills the TLB from the page table resident in memory. Figure 5.11 shows the translation and exception flow of the TLB.

Software can write over a selected TLB entry or use a hardware mechanism to write into a random entry. The *Random* register selects which TLB entry to use on a TLBWR. This register decrements almost every cycle, wrapping to the maximum once its value is equal to the *Wired* register. Thus, TLB entries below the *Wired* value cannot be replaced by a TLBWR allowing important mappings to be preserved. In order to reduce the possibility for a live-lock situation, the *Random* register includes a 10-bit LFSR that introduces a pseudo-random perturbation into the decrement.

The core implements a TLB write-compare mechanism to ensure that multiple TLB matches do not occur. On the TLB write operation, the VPN2 field to be written is compared with all other entries in the TLB. If a match occurs, the write will be dropped.

## 5.4.2 Memory Space

To assist in controlling both the amount of mapped space and the replacement characteristics of various memory regions, the 34K core provides two mechanisms.

#### 5.4.2.1 Page Sizes

First, the page size can be configured, on a per entry basis, to map different page sizes ranging from 4 KByte to 256 MByte, in multiples of 4. The CP0 *PageMask* register is loaded with the desired page size, which is then entered into the TLB when a new entry is written. Thus, operating systems can provide special-purpose maps. For example, a typical frame buffer can be memory mapped with only one TLB entry.

The 34K core implements the following page sizes:

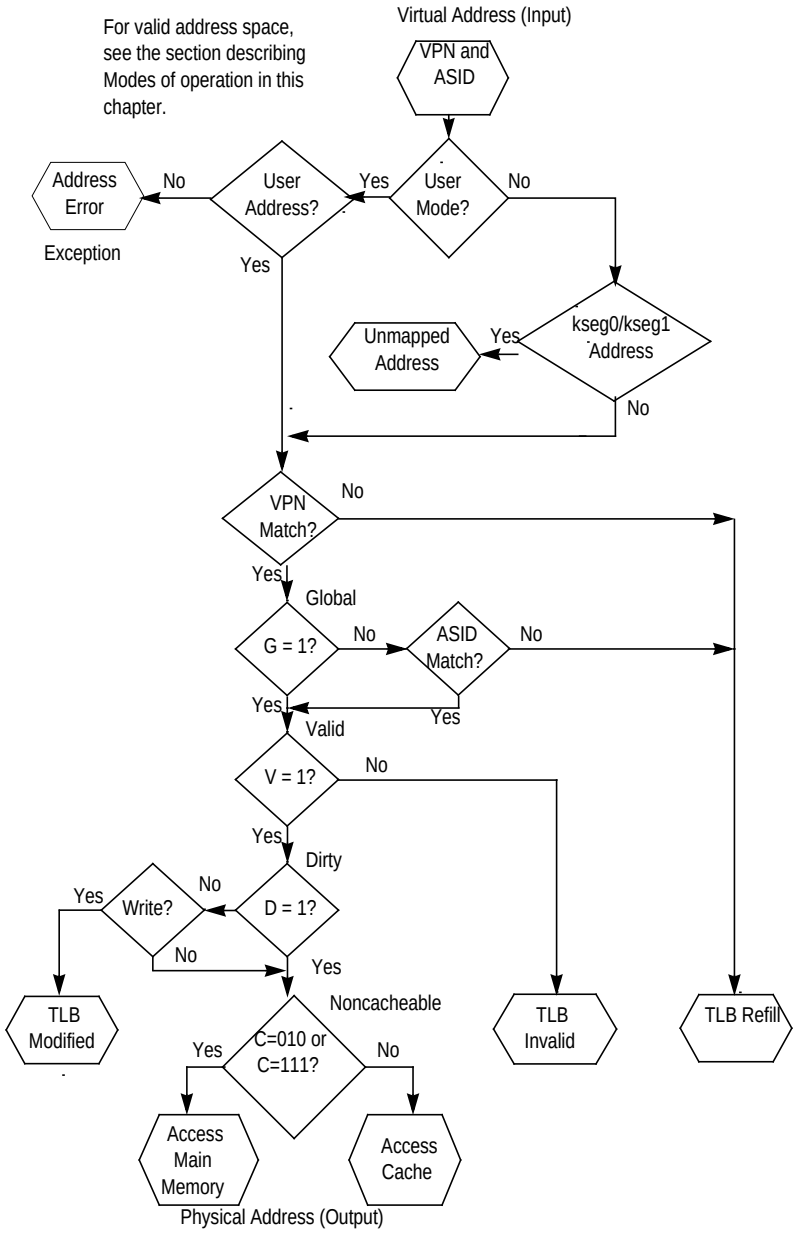
4K, 16K, 64K, 256K, 1M, 4M, 16M, 64M, 256M.

Software may determine which page sizes are supported by writing all ones to the CP0 *PageMask* register, then reading the value back. For additional information, see Section 7.2.20 “PageMask Register (CP0 Register 5, Select 0)”.

#### 5.4.2.2 Replacement Algorithm

The second mechanism controls the replacement algorithm when a TLB miss occurs. To select a TLB entry to be written with a new mapping, the 34K core provides a random replacement algorithm. However, the processor also provides a mechanism whereby a programmable number of mappings can be locked into the TLB via the CP0 Wired register, thus avoiding random replacement. Please refer to Section 7.2.21 “Wired Register (CP0 Register 6, Select 0)” for further details.

**Figure 5.11 TLB Address Translation Flow in the 34K™ Processor Core**




## 5.4.3 TLB Instructions

Table 5.9 lists the TLB-related instructions. Refer to Chapter 15, "34K™ Processor Core Instructions" for more information on these instructions.

**Table 5.9 TLB Instructions**

| Op Code | Description of Instruction |
|---|---|
| TLBP | Translation Lookaside Buffer Probe |
| TLBR | Translation Lookaside Buffer Read |

**Table 5.9 TLB Instructions (Continued)**

| Op Code | Description of Instruction |
|---|---|
| TLBWI | Translation Lookaside Buffer Write Index |
| TLBWR | Translation Lookaside Buffer Write Random |

### 5.4.4 Shared TLB mode

When a TLB based MMU is implemented on both VPEs, it is possible to combine the two arrays into a single array that will be used by both VPEs. This is done by setting $MVPControl_{STLB}$ to 1 - this bit should only be changed by code executing in an unmapped address region and only when there are no valid TLB entries.

- Both VPEs share the same address space (including ASID and VA - VPE # is not part of the TLB tag)
- Software is expected to keep the *Wired* register the same for the two VPEs and the value should allow at least 2 entries for random selection if TLBWR is going to be used.
- Hardware constrains TLBWR from overwriting the entry pointed at by the *Index* register of the opposite VPE (if the P bit is cleared).
  - This is needed to avoid interfering TLB maintenance code running on opposite VPE.
  - The is done by stalling the TLBWR until *Random* does not match *Index*
- $Index_P$ is writable to allow software to “park” the *Index* register when it is done operating on the TLB, allowing all TLB entries to be used by a TLBWR
- Note that the maximum TLB size allowed by the architecture is 64 dual entries. Sharing 2x64 entry JTLBs would result in half of the entries being ignored.
- Multiple TCs could potentially miss on the same address at nearly the same time and process the miss. The hardware will squash the second TLB write to avoid multiple matches. TLBWx to VA/ASID already in JTLB will be dropped. (Note that this can also happen to two TCs in one VPE even without sharing)

## 5.5 Fixed Mapping MMU

The 34K core optionally implements a simple Fixed Mapping (FM) memory management unit that is smaller than the a full translation lookaside buffer (TLB) and more easily synthesized. Like a TLB, the FM performs virtual-to-physical address translation and provides attributes for the different memory segments. Those memory segments which are unmapped in a TLB implementation (kseg0 and kseg1) are translated identically by the FM MMU.

The FM also determines the cacheability of each segment. These attributes are controlled via bits in the *Config* register. Table 5.10 shows the encoding for the K23 (bits 30:28), KU (bits 27:25) and K0 (bits 2:0) of the *Config* register.

**Table 5.10 Cache Coherency Attributes**

| Config Register Fields K23, KU, and K0 | Cache Coherency Attribute |
|---|---|
| 0 | Cacheable, noncoherent, write-through, no write-allocate |
| 1 | Reserved |

**Table 5.10 Cache Coherency Attributes (Continued)**

| Config Register Fields K23, KU, and K0 | Cache Coherency Attribute |
|---|---|
| 2 | Uncached |
| 3 | Cacheable, noncoherent, write-back, write-allocate |
| 4 | Reserved |
| 5 | Reserved |
| 6 | Reserved |
| 7 | Uncached Accelerated |

With the FM MMU, no translation exceptions can be taken, although address errors are still possible.

**Table 5.11 Cacheability of Segments with Fixed Mapping Translation**

| Segment | Virtual Address Range | Cacheability |
|---|---|---|
| useg/kuseg | 0x0000_0000-0x7FFF_FFFF | Controlled by the KU field (bits 27:25) of the *Config* register. Refer to Table 5.10 for the encoding. |
| kseg0 | 0x8000_0000-0x9FFF_FFFF | Controlled by the K0 field (bits 2:0) of the *Config* register. See Table 5.10 for the encoding. |
| kseg1 | 0xA000_0000-0xBFFF_FFFF | Always uncacheable |
| kseg2 | 0xC000_0000-0xDFFF_FFFF | Controlled by the K23 field (bits 30:28) of the *Config* register. Refer to Table 5.10 for the encoding. |
| kseg3 | 0xE000_0000-0xFFFF_FFFF | Controlled by K23 field (bits 30:28) of the *Config* register. Refer to Table 5.10 for the encoding. |

The FM performs a simple translation to map from virtual addresses to physical addresses. This mapping is shown in Figure 5.12. When ERL=1, useg and kuseg become unmapped and uncached just like they do if there is a TLB. The ERL mapping is shown in Figure 5.13.

The ERL bit is usually never asserted by software. It is asserted by hardware after a Reset, NMI, or Cache Error. See Section 6.8 “Exceptions” for further information on exceptions.

**Figure 5.12 FM Memory Map (ERL=0) in the 34K™ Processor Core**

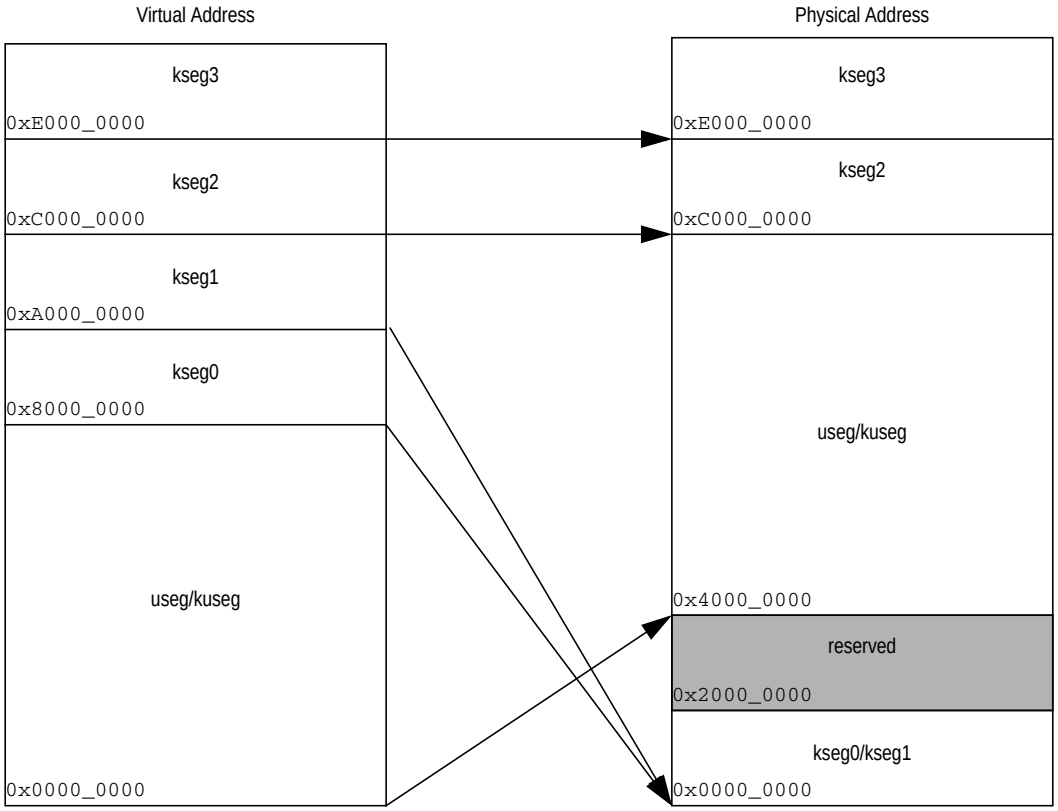
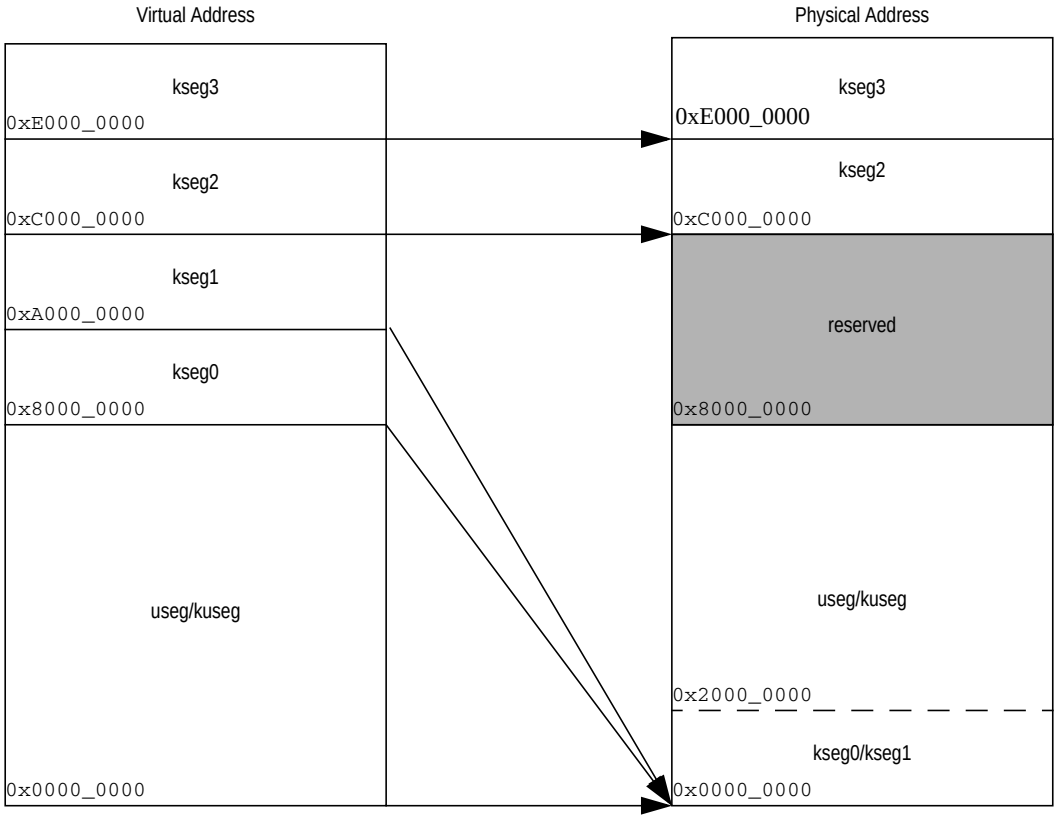


**Figure 5.13 FM Memory Map (ERL=1) in the 34K™ Processor Core**




## 5.6 System Control Coprocessor

The System Control Coprocessor (CP0) is implemented as an integral part of the 34K processor core and supports memory management, address translation, exception handling, and other privileged operations. Certain CP0 registers are used to support memory management. Refer to Chapter 7, “CP0 Registers of the 34K™ Core” for more information on the CP0 register set.

*Chapter 6*

# Exceptions and Interrupts in the 34K™ Core

The 34K processor core receives exceptions from a number of sources, including translation lookaside buffer (TLB) misses, arithmetic overflows, I/O interrupts, and system calls. When the CPU detects one of these exceptions, the normal sequence of instruction execution is suspended and the processor enters kernel mode.

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

When an exception occurs, the core loads the *Exception Program Counter* (*EPC*) register with a location where execution can restart after the exception has been serviced. Most exceptions are *precise*, which mean that *EPC* can be used to identify the instruction that caused the exception. For precise exceptions the restart location in the *EPC* register is the address of the instruction that caused the exception or, if the instruction was executing in a branch delay slot, the address of the branch instruction immediately preceding the delay slot. To distinguish between the two, software must read the BD bit in the CP0 *Cause* register. Bus error exceptions and CP2 exceptions may be imprecise. For imprecise exceptions the instruction that caused the exception can not be identified.

This chapter contains the following sections:



## 6.1 Exception Conditions

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

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

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

A number of exceptions can be taken imprecisely - that is, they are taken after the instruction that caused them has completed and potentially after following instructions have completed.

## 6.2 Exception Priority

Table 6.1 lists all possible exceptions, and the relative priority of each, highest to lowest. Several of these exceptions can happen simultaneously, in that event the exception with the highest priority is the one taken.

**Table 6.1 Priority of Exceptions**

| Exception | Description |
|---|---|
| Reset | Assertion of SI_Reset signal. |
| DSS | EJTAG Debug Single Step. |
| DINT | EJTAG Debug Interrupt. Caused by the assertion of the external *EJ_DINT* input, or by setting the EjtagBrk bit in the *ECR* register. |
| DDBLImpr/DDBSImpr | Debug Data Break Load/Store Imprecise |
| NMI | Asserting edge of *SI_NMI* signal. |
| Interrupt | Assertion of unmasked hardware or software interrupt signal. |
| Deferred Watch | Deferred Watch (unmasked by K\|DM->!(K\|DM) transition). |
| DIB | EJTAG debug hardware instruction break matched. |
| WATCH | A reference to an address in one of the watch registers (fetch). |
| AdEL | Fetch address alignment error.<br>Fetch reference to protected address. |
| TLBL | Fetch TLB miss<br>Fetch TLB hit to page with V=0 |
| ICache Error | Parity error on ICache access |
| IBE | Instruction fetch bus error. |
| DBp | EJTAG Breakpoint (execution of SDBBP instruction). |
| Sys | Execution of SYSCALL instruction. |
| Bp | Execution of BREAK instruction. |
| CpU | Execution of a coprocessor instruction for a coprocessor that is not enabled. |
| CEU | Execution of a CorExtend instruction modifying local state when CorExtend is not enabled. |
| RI | Execution of a Reserved Instruction. |
| FPE | Floating Point exception |

**Table 6.1 Priority of Exceptions (Continued)**

| Exception | Description |
|---|---|
| C2E | Coprocessor2 Exception |
| IS1 | Implementation specific Coprocessor2 exception |
| Ov | Execution of an arithmetic instruction that overflowed. |
| Tr | Execution of a trap (when trap condition is true). |
| MT_ov | A Thread Overflow condition, where a TC allocation request cannot be satisfied. |
| MT_under | A Thread Underflow condition, where the termination and deallocation of a thread leaves no dynamically allocatable TCs activated on a VPE. |
| MT_invalid | An Invalid Qualifier condition, where a YIELD instruction specifies an invalid condition for resuming execution. |
| MT_yield_sched | A YIELD Scheduler exception condition, where a valid YIELD instruction would have caused a rescheduling of a TC, and the YIELD intercept bit is set. |
| DDBL / DDBS | EJTAG Data Address Break (address only) |
| WATCH | A reference to an address in one of the watch registers (data). |
| AdEL | Load address alignment error.<br>Load reference to protected address. |
| AdES | Store address alignment error.<br>Store to protected address. |
| TLBL | Load TLB miss.<br>Load TLB hit to page with V=0 |
| TLBS | Store TLB miss.<br>Store TLB hit to page with V=0 |
| TLB Mod | Store to TLB page with D=0. |
| DCache Error | Cache parity error - imprecise |
| L2 Cache Error | L2 Cache ECC error - imprecise |
| DBE | Load or store bus error - imprecise |
| MT_GSS | A Gating Storage Scheduler exception, where a Gating Storage load or store would have blocked and caused a rescheduling of a TC, and the GS intercept bit is set. |
| MT_GS | A Gating Storage exception condition, where implementation-dependent logic associated with gating or inter-thread communication (ITC) storage requires software intervention. |

## 6.3 Interrupts

Older 32-bit cores available from MIPS that implemented Release 1 of the Architecture included support for two software interrupts, six hardware interrupts, and a special-purpose timer interrupt. The timer interrupt was provided external to the core and typically combined with hardware interrupt 5 in an system-dependent manner. Interrupts were handled either through the general exception vector (offset 16#180) or the special interrupt vector (16#200), based on the value of $Cause_{IV}$. Software was required to prioritize interrupts as a function of the $Cause_{IP}$ bits in the interrupt handler prologue.

Release 2 of the Architecture, implemented by the 34K core, adds an upward-compatible extension to the Release 1 interrupt architecture that supports vectored interrupts. In addition, Release 2 adds a new interrupt mode that supports the use of an external interrupt controller by changing the interrupt architecture.

Additionally, internal performance counters were added to the 34K core. These counters can be set up to count various events within the core. When the MSB of the counter gets set, it can trigger a performance counter interrupt. This is handled like the timer interrupt - it is an output of the core and can be brought back into the core's interrupt pins in a system dependent manner.

### 6.3.1 Interrupt Modes

The 34K core includes support for three interrupt modes, as defined by Release 2 of the Architecture:

- Interrupt compatibility mode, which acts identically to that in an implementation of Release 1 of the Architecture.
- Vectored Interrupt (VI) mode, which adds the ability to prioritize and vector interrupts to a handler dedicated to that interrupt, and to assign a GPR shadow set for use during interrupt processing. The presence of this mode is denoted by the VInt bit in the *Config3* register. This mode is architecturally optional; but it is always present on the 34K core, so the VInt bit will always read as a 1 for the 34K core.
- External Interrupt Controller (EIC) mode, which redefines the way in which interrupts are handled to provide full support for an external interrupt controller handling prioritization and vectoring of interrupts. This presence of this mode denoted by the VEIC bit in the *Config3* register. Again, this mode is architecturally optional. On the 34K core, the VEIC bit is set externally by the static input, *SI_EICPresent*, to allow system logic to indicate the presence of an external interrupt controller.

The reset state of the processor is to interrupt compatibility mode such that a processor supporting Release 2 of the Architecture, like the 34K core, is fully compatible with implementations of Release 1 of the Architecture.

Table 6.2 shows the current interrupt mode of the processor as a function of the coprocessor 0 register fields that can affect the mode.

**Table 6.2 Interrupt Modes**

| $Status_{BEV}$ | $Cause_{IV}$ | $IntCtl_{VS}$ | $Config3_{VINT}$ | $Config3_{VEIC}$ | Interrupt Mode |
|---|---|---|---|---|---|
| 1 | x | x | x | x | Compatibility |
| x | 0 | x | x | x | Compatibility |
| x | x | =0 | x | x | Compatibility |
| 0 | 1 | ≠0 | 1 | 0 | Vectored Interrupt |
| 0 | 1 | ≠0 | x | 1 | External Interrupt Controller |
| 0 | 1 | ≠0 | 0 | 0 | Can't happen - $IntCtl_{VS}$ can not be non-zero if neither Vectored Interrupt nor External Interrupt Controller mode is implemented. |
| "x" denotes don't care | | | | | |

#### 6.3.1.1 Interrupt Compatibility Mode

This is the default interrupt mode for the processor and is entered when a Reset exception occurs. In this mode, interrupts are non-vectored and dispatched though exception vector offset 16#180 (if $Cause_{IV}$ = 0) or vector offset 16#200 (if $Cause_{IV}$ = 1). This mode is in effect if any of the following conditions are true:

- $Cause_{IV}$ = 0
- $Status_{BEV}$ = 1
- $IntCtl_{VS}$ = 0, which would be the case if vectored interrupts are not implemented, or have been disabled.

A typical software handler for interrupt compatibility mode might look as follows:

```
/*
 * Assumptions:
 *  - Cause_IV = 1 (if it were zero, the interrupt exception would have to
 *                  be isolated from the general exception vector before getting
 *                  here)
 *  - GPRs k0 and k1 are available (no shadow register switches invoked in
 *                                  compatibility mode)
 *  - The software priority is IP7..IP0 (HW5..HW0, SW1..SW0)
 *
 * Location: Offset 0x200 from exception base
 */

IVexception:
   mfc0   k0, C0_Cause        /* Read Cause register for IP bits */
   mfc0   k1, C0_Status       /* and Status register for IM bits */
   andi   k0, k0, M_CauseIM   /* Keep only IP bits from Cause */
   and    k0, k0, k1          /* and mask with IM bits */
   beq    k0, zero, Dismiss   /* no bits set - spurious interrupt */
   clz    k0, k0              /* Find first bit set, IP7..IP0; k0 = 16..23 */
   xori   k0, k0, 0x17        /* 16..23 => 7..0 */
   sll    k0, k0, VS          /* Shift to emulate software IntCtl_VS */
   la     k1, VectorBase      /* Get base of 8 interrupt vectors */
   addu   k0, k0, k1          /* Compute target from base and offset */
   jr     k0                  /* Jump to specific exception routine */
   nop

/*
 * Each interrupt processing routine processes a specific interrupt, analogous
 * to those reached in VI or EIC interrupt mode. Since each processing routine
 * is dedicated to a particular interrupt line, it has the context to know
 * which line was asserted.  Each processing routine may need to look further
 * to determine the actual source of the interrupt if multiple interrupt requests
 * are ORed together on a single IP line. Once that task is performed, the
 * interrupt may be processed in one of two ways:
 *
 * - Completely at interrupt level (e.g., a simply UART interrupt). The
 *   SimpleInterrupt routine below is an example of this type.
 * - By saving sufficient state and re-enabling other interrupts. In this
 *   case the software model determines which interrupts are disabled during
 *   the processing of this interrupt. Typically, this is either the single
 *   StatusIM bit that corresponds to the interrupt being processed, or some
 *   collection of other Status_IM bits so that "lower" priority interrupts are
 *    also disabled. The NestedInterrupt routine below is an example of this type.
 */

SimpleInterrupt:
/*
 * Process the device interrupt here and clear the interupt request
 * at the device. In order to do this, some registers may need to be
```

```
 * saved and restored. The coprocessor 0 state is such that an ERET
 * will simple return to the interrupted code.
 */
   eret                       /* Return to interrupted code */

NestedException:
/*
 * Nested exceptions typically require saving the EPC and Status registers,
 * any GPRs that may be modified by the nested exception routine, disabling
 * the appropriate IM bits in Status to prevent an interrupt loop, putting
 * the processor in kernel mode, and re-enabling interrupts. The sample code
 * below can not cover all nuances of this processing and is intended only
 * to demonstrate the concepts.
 */

   /* Save GPRs here, and setup software context */
   mfc0   k0, C0_EPC          /* Get restart address */
   sw     k0, EPCSave         /* Save in memory */
   mfc0   k0, C0_Status       /* Get Status value */
   sw     k0, StatusSave      /* Save in memory */
   li     k1, ~IMbitsToClear  /* Get Im bits to clear for this interrupt */
                              /*   this must include at least the IM bit */
                              /*   for the current interrupt, and may include */
                              /*   others */
   and    k0, k0, k1             /* Clear bits in copy of Status */
   ins    k0, zero, S_StatusEXL, (W_StatusKSU+W_StatusERL+W_StatusEXL)
                                 /* Clear KSU, ERL, EXL bits in k0 */
   mtc0   k0, C0_Status          /* Modify mask, switch to kernel mode, */
                                 /*   re-enable interrupts */

   /*
    * Process interrupt here, including clearing device interrupt.
    * In some environments this may be done with a thread running in
    * kernel or user mode. Such an environment is well beyond the scope of
    * this example.
    */

/*
 * To complete interrupt processing, the saved values must be restored
 * and the original interrupted code restarted.
 */

   di                         /* Disable interrupts - may not be required */
   lw     k0, StatusSave      /* Get saved Status (including EXL set) */
   lw     k1, EPCSave         /*   and EPC */
   mtc0   k0, C0_Status       /* Restore the original value */
   mtc0   k1, C0_EPC          /*  and EPC */
   /* Restore GPRs and software state */
   eret                       /* Dismiss the interrupt */
```

#### 6.3.1.2 Vectored Interrupt Mode

Vectored Interrupt mode builds on the interrupt compatibility mode by adding a priority encoder to prioritize pending interrupts and to generate a vector with which each interrupt can be directed to a dedicated handler routine. This mode also allows each interrupt to be mapped to a GPR shadow set for use by the interrupt handler. Vectored Interrupt mode is in effect if all of the following conditions are true:

- $Config3_{VInt} = 1$
- $Config3_{VEIC} = 0$
- $IntCtl_{VS} \neq 0$
- $Cause_{IV} = 1$
- $Status_{BEV} = 0$

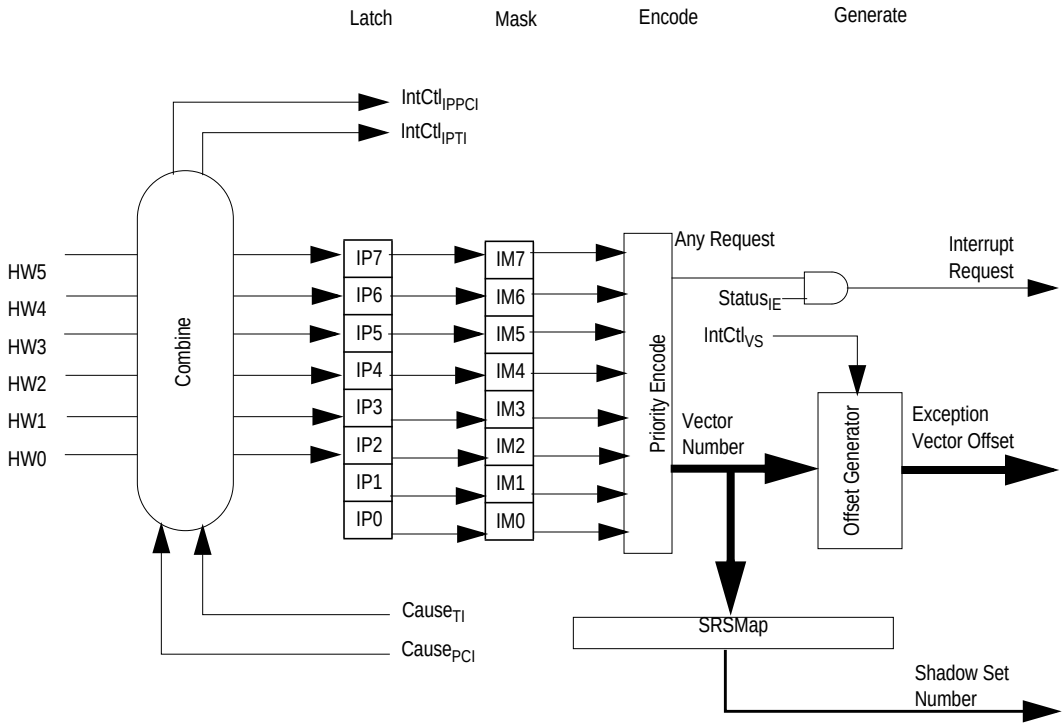
In VI interrupt mode, the six hardware interrupts are interpreted as individual hardware interrupt requests. The timer and performance counter interrupts are combined in a system-dependent way (external to the core) with the hardware interrupts (the interrupt with which they are combined is indicated by the $IntCtl_{IPTI/IPCI}$ fields) to provide the appropriate relative priority of the those interrupts with that of the hardware interrupts. The processor interrupt logic ANDs each of the $Cause_{IP}$ bits with the corresponding $Status_{IM}$ bits. If any of these values is 1, and if interrupts are enabled ($Status_{IE} = 1$, $Status_{EXL} = 0$, and $Status_{ERL} = 0$), an interrupt is signaled and a priority encoder scans the values in the order shown in Table 6.3.

**Table 6.3 Relative Interrupt Priority for Vectored Interrupt Mode**

| Relative Priority | Interrupt Type | Interrupt Source | Interrupt Request Calculated From | Vector Number Generated by Priority Encoder |
|---|---|---|---|---|
| Highest Priority | Hardware | HW5 | IP7 and IM7 | 7 |
| | | HW4 | IP6 and IM6 | 6 |
| | | HW3 | IP5 and IM5 | 5 |
| | | HW2 | IP4 and IM4 | 4 |
| | | HW1 | IP3 and IM3 | 3 |
| | | HW0 | IP2 and IM2 | 2 |
| | Software | SW1 | IP1 and IM1 | 1 |
| Lowest Priority | | SW0 | IP0 and IM0 | 0 |

The priority order places a relative priority on each hardware interrupt and places the software interrupts at a priority lower than all hardware interrupts. When the priority encoder finds the highest priority pending interrupt, it outputs an encoded vector number that is used in the calculation of the handler for that interrupt, as described below. This is shown pictorially in Figure 6.1.

**Figure 6.1 Interrupt Generation for Vectored Interrupt Mode**




A typical software handler for vectored interrupt mode bypasses the entire sequence of code following the IVexception label shown for the compatibility mode handler above. Instead, the hardware performs the prioritization, dispatching directly to the interrupt processing routine. Unlike the compatibility mode examples, a vectored interrupt handler may take advantage of a dedicated GPR shadow set to avoid saving any registers. As such, the SimpleInterrupt code shown above need not save the GPRs.

A nested interrupt is similar to that shown for compatibility mode, but may also take advantage of running the nested exception routine in the GPR shadow set dedicated to the interrupt or in another shadow set. Such a routine might look as follows:

```
NestedException:
/*
 * Nested exceptions typically require saving the EPC, Status and SRSCtl registers,
 * setting up the appropriate GPR shadow set for the routine, disabling
 * the appropriate IM bits in Status to prevent an interrupt loop, putting
 * the processor in kernel mode, and re-enabling interrupts. The sample code
 * below can not cover all nuances of this processing and is intended only
 * to demonstrate the concepts.
 */

   /* Use the current GPR shadow set, and setup software context */
   mfc0   k0, C0_EPC          /* Get restart address */
   sw     k0, EPCSave         /* Save in memory */
   mfc0   k0, C0_Status       /* Get Status value */
   sw     k0, StatusSave      /* Save in memory */
   mfc0   k0, C0_SRSCtl       /* Save SRSCtl if changing shadow sets */
   sw     k0, SRSCtlSave
   li     k1, ~IMbitsToClear  /* Get Im bits to clear for this interrupt */
                              /*   this must include at least the IM bit */
```

```
                                /*   for the current interrupt, and may include */
                                /*   others */
    and    k0, k0, k1              /* Clear bits in copy of Status */
    /* If switching shadow sets, write new value to SRSCtl_PSS here */
    ins    k0, zero, S_StatusEXL, (W_StatusKSU+W_StatusERL+W_StatusEXL)
                                /* Clear KSU, ERL, EXL bits in k0 */
    mtc0   k0, C0_Status        /* Modify mask, switch to kernel mode, */
                                /*   re-enable interrupts */
    /*
     * If switching shadow sets, clear only KSU above, write target
     * address to EPC, and do execute an eret to clear EXL, switch
     * shadow sets, and jump to routine
     */

    /* Process interrupt here, including clearing device interrupt */

/*
 * To complete interrupt processing, the saved values must be restored
 * and the original interrupted code restarted.
 */

    di                          /* Disable interrupts - may not be required */
    lw     k0, StatusSave       /* Get saved Status (including EXL set) */
    lw     k1, EPCSave          /*   and EPC */
    mtc0   k0, C0_Status        /* Restore the original value */
    lw     k0, SRSCtlSave       /* Get saved SRSCtl */
    mtc0   k1, C0_EPC           /*  and EPC */
    mtc0   k0, C0_SRSCtl        /* Restore shadow sets */
    ehb                         /* Clear hazard */
    eret                        /* Dismiss the interrupt */
```

#### 6.3.1.3 External Interrupt Controller Mode

External Interrupt Controller Mode redefines the way that the processor interrupt logic is configured to provide support for an external interrupt controller. The interrupt controller is responsible for prioritizing all interrupts, including hardware, software, timer, and performance counter interrupts, and directly supplying to the processor the vector number of the highest priority interrupt. EIC interrupt mode is in effect if all of the following conditions are true:

- $Config3_{VEIC} = 1$
- $IntCtl_{VS} \neq 0$
- $Cause_{IV} = 1$
- $Status_{BEV} = 0$

In EIC interrupt mode, the processor sends the state of the software interrupt requests ($Cause_{IP1..IP0}$) and the timer and performance counter interrupt requests ($Cause_{TI/PCI}$) to the external interrupt controller, where it prioritizes these interrupts in a system-dependent way with other hardware interrupts. The interrupt controller can be a hard-wired logic block, or it can be configurable based on control and status registers. This allows the interrupt controller to be more specific or more general as a function of the system environment and needs.

The external interrupt controller prioritizes its interrupt requests and produces the vector number of the highest priority interrupt to be serviced. The vector number, called the Requested Interrupt Priority Level (RIPL), is a 6-bit encoded value in the range 0..63, inclusive. A value of 0 indicates that no interrupt requests are pending. The values

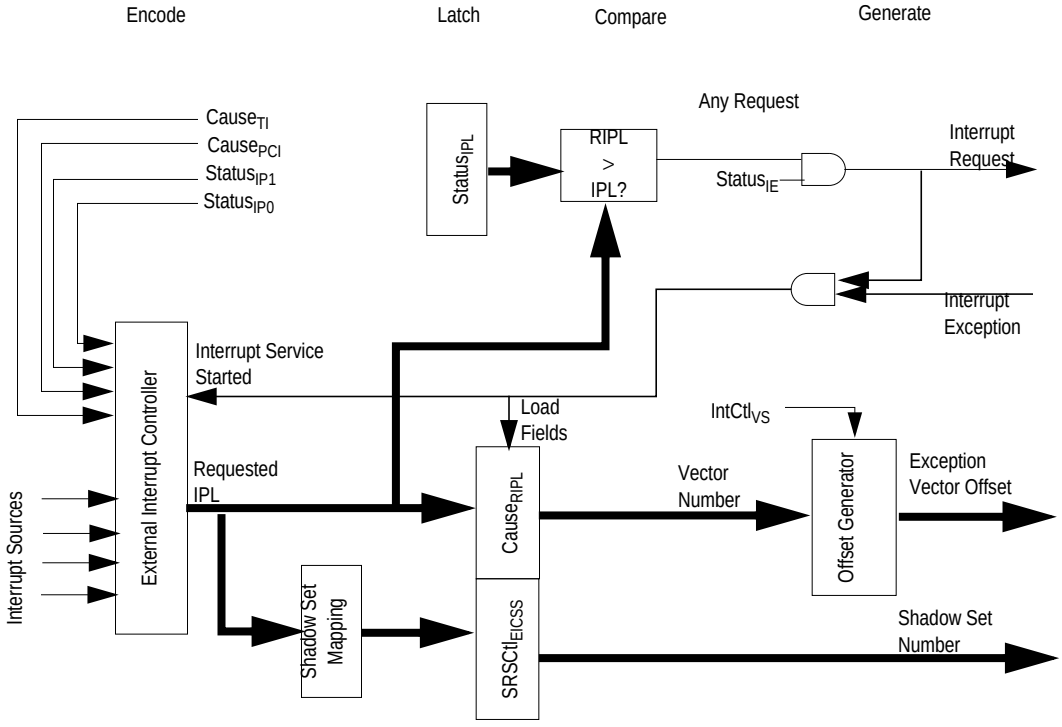
1..63 represent the lowest (1) to highest (63) RIPL for the interrupt to be serviced. The interrupt controller passes this value on the 6 hardware interrupt line, which are treated as an encoded value in EIC interrupt mode.

*Status*$_{IPL}$ (which overlays *Status*$_{IM7..IM2}$) is interpreted as the Interrupt Priority Level (IPL) at which the processor is currently operating (with a value of zero indicating that no interrupt is currently being serviced). When the interrupt controller requests service for an interrupt, the processor compares RIPL with *Status*$_{IPL}$ to determine if the requested interrupt has higher priority than the current IPL. If RIPL is strictly greater than *Status*$_{IPL}$, and interrupts are enabled (*Status*$_{IE}$ = 1, *Status*$_{EXL}$ = 0, and *Status*$_{ERL}$ = 0) an interrupt request is signaled to the pipeline. When the processor starts the interrupt exception, it loads RIPL into *Cause*$_{RIPL}$ (which overlays *Cause*$_{IP7..IP2}$) and signals the external interrupt controller to notify it that the request is being serviced. The interrupt exception uses the value of *Cause*$_{RIPL}$ as the vector number. Because *Cause*$_{RIPL}$ is only loaded by the processor when an interrupt exception is signaled, it is available to software during interrupt processing.

In EIC interrupt mode, the external interrupt controller is also responsible for supplying the GPR shadow set number to use when servicing the interrupt. As such, the *SRSMap* register is not used in this mode, and the mapping of the vectored interrupt to a GPR shadow set is done by programming (or designing) the interrupt controller to provide the correct GPR shadow set number when an interrupt is requested. When the processor loads an interrupt request into Cause$_{RIPL}$, it also loads the GPR shadow set number into *SRSCtl*$_{EICSS}$, which is copied to *SRSCtl*$_{CSS}$ when the interrupt is serviced.

The operation of EIC interrupt mode is shown pictorially in Figure 6.2.

**Figure 6.2 Interrupt Generation for External Interrupt Controller Interrupt Mode**




A typical software handler for EIC interrupt mode bypasses the entire sequence of code following the IVexception label shown for the compatibility mode handler above. Instead, the hardware performs the prioritization, dispatching directly to the interrupt processing routine. Unlike the compatibility mode examples, an EIC interrupt handler may

take advantage of a dedicated GPR shadow set to avoid saving any registers. As such, the SimpleInterrupt code shown above need not save the GPRs.

A nested interrupt is similar to that shown for compatibility mode, but may also take advantage of running the nested exception routine in the GPR shadow set dedicated to the interrupt or in another shadow set. It also need only copy $Cause_{RIPL}$ to $Status_{IPL}$ to prevent lower priority interrupts from interrupting the handler. Such a routine might look as follows:

```
NestedException:
/*
 * Nested exceptions typically require saving the EPC, Status,and SRSCtl registers,
 * setting up the appropriate GPR shadow set for the routine, disabling
 * the appropriate IM bits in Status to prevent an interrupt loop, putting
 * the processor in kernel mode, and re-enabling interrupts. The sample code
 * below can not cover all nuances of this processing and is intended only
 * to demonstrate the concepts.
 */

   /* Use the current GPR shadow set, and setup software context */
   mfc0   k1, C0_Cause        /* Read Cause to get RIPL value */
   mfc0   k0, C0_EPC          /* Get restart address */
   srl    k1, k1, S_CauseRIPL /* Right justify RIPL field */
   sw     k0, EPCSave         /* Save in memory */
   mfc0   k0, C0_Status       /* Get Status value */
   sw     k0, StatusSave      /* Save in memory */
   ins    k0, k1, S_StatusIPL, 6 /* Set IPL to RIPL in copy of Status */
   mfc0   k1, C0_SRSCtl       /* Save SRSCtl if changing shadow sets */
   sw     k1, SRSCtlSave
   /* If switching shadow sets, write new value to SRSCtl_PSS here */
   ins    k0, zero, S_StatusEXL, (W_StatusKSU+W_StatusERL+W_StatusEXL)
                                /* Clear KSU, ERL, EXL bits in k0 */
   mtc0   k0, C0_Status         /* Modify IPL, switch to kernel mode, */
                                /*   re-enable interrupts */
   /*
    * If switching shadow sets, clear only KSU above, write target
    * address to EPC, and do execute an eret to clear EXL, switch
    * shadow sets, and jump to routine
    */

   /* Process interrupt here, including clearing device interrupt */

/*
 * The interrupt completion code is identical to that shown for VI mode above.
 */
```

## 6.3.2 Generation of Exception Vector Offsets for Vectored Interrupts

For vectored interrupts (in either VI or EIC interrupt mode), a vector number is produced by the interrupt control logic. This number is combined with $IntCtl_{VS}$ to create the interrupt offset, which is added to 16#200 to create the exception vector offset. For VI interrupt mode, the vector number is in the range 0..7, inclusive. For EIC interrupt mode, the vector number is in the range 1..63, inclusive (0 being the encoding for “no interrupt”). The $IntCtl_{VS}$ field specifies the spacing between vector locations. If this value is zero (the default reset state), the vector spacing is zero and the processor reverts to Interrupt Compatibility Mode. A non-zero value enables vectored interrupts, and Table

6.4 shows the exception vector offset for a representative subset of the vector numbers and values of the *IntCtl*$_{VS}$ field.

**Table 6.4 Exception Vector Offsets for Vectored Interrupts**

| Vector Number | Value of IntCtl$_{VS}$ Field | | | | |
|---|---|---|---|---|---|
| | 2#00001 | 2#00010 | 2#00100 | 2#01000 | 2#10000 |
| 0 | 16#0200 | 16#0200 | 16#0200 | 16#0200 | 16#0200 |
| 1 | 16#0220 | 16#0240 | 16#0280 | 16#0300 | 16#0400 |
| 2 | 16#0240 | 16#0280 | 16#0300 | 16#0400 | 16#0600 |
| 3 | 16#0260 | 16#02C0 | 16#0380 | 16#0500 | 16#0800 |
| 4 | 16#0280 | 16#0300 | 16#0400 | 16#0600 | 16#0A00 |
| 5 | 16#02A0 | 16#0340 | 16#0480 | 16#0700 | 16#0C00 |
| 6 | 16#02C0 | 16#0380 | 16#0500 | 16#0800 | 16#0E00 |
| 7 | 16#02E0 | 16#03C0 | 16#0580 | 16#0900 | 16#1000 |
| | • • • | | | | |
| 61 | 16#09A0 | 16#1140 | 16#2080 | 16#3F00 | 16#7C00 |
| 62 | 16#09C0 | 16#1180 | 16#2100 | 16#4000 | 16#7E00 |
| 63 | 16#09E0 | 16#11C0 | 16#2180 | 16#4100 | 16#8000 |

The general equation for the exception vector offset for a vectored interrupt is:

```
vectorOffset ← 16#200 + (vectorNumber × (IntCtl_VS ‖ 2#00000))
```

## 6.4 GPR Shadow Registers

Release 2 of the Architecture optionally removes the need to save and restore GPRs on entry to high priority interrupts or exceptions, and to provide specified processor modes with the same capability. This is done by introducing multiple copies of the GPRs, called *shadow sets*, and allowing privileged software to associate a shadow set with entry to kernel mode via an interrupt vector or exception. The normal GPRs are logically considered shadow set zero.

The number of GPR shadow sets is a build-time option on the 34K core. Although Release 2 of the Architecture defines a maximum of 16 shadow sets, the core allows one (the normal GPRs), two, or four shadow sets. The highest number actually implemented is indicated by the *SRSCtl*$_{HSS}$ field. If this field is zero, only the normal GPRs are implemented.

Shadow sets are new copies of the GPRs that can be substituted for the normal GPRs on entry to kernel mode via an interrupt or exception. Once a shadow set is bound to a kernel mode entry condition, reference to GPRs work exactly as one would expect, but they are redirected to registers that are dedicated to that condition. Privileged software may need to reference all GPRs in the register file, even specific shadow registers that are not visible in the current mode. The RDPGPR and WRPGPR instructions are used for this purpose. The CSS field of the *SRSCtl* register provides the number of the current shadow register set, and the PSS field of the *SRSCtl* register provides the number of the previous shadow register set (that which was current before the last exception or interrupt occurred).

If the processor is operating in VI interrupt mode, binding of a vectored interrupt to a shadow set is done by writing to the *SRSMap* register. If the processor is operating in EIC interrupt mode, the binding of the interrupt to a specific

shadow set is provided by the external interrupt controller, and is configured in an implementation-dependent way. Binding of an exception or non-vectored interrupt to a shadow set is done by writing to the ESS field of the *SRSCtl* register. When an exception or interrupt occurs, the value of $SRSCtl_{CSS}$ is copied to $SRSCtl_{PSS}$, and $SRSCtl_{CSS}$ is set to the value taken from the appropriate source. On an ERET, the value of $SRSCtl_{PSS}$ is copied back into $SRSCtl_{CSS}$ to restore the shadow set of the mode to which control returns. More precisely, the rules for updating the fields in the *SRSCtl* register on an interrupt or exception are as follows:

1. No field in the *SRSCtl* register is updated if any of the following conditions is true. In this case, steps 2 and 3 are skipped.
   - The exception is one that sets $Status_{ERL}$: Reset or NMI.
   - The exception causes entry into EJTAG Debug Mode
   - $Status_{BEV} = 1$
   - $Status_{EXL} = 1$
2. $SRSCtl_{CSS}$ is copied to $SRSCtl_{PSS}$
3. $SRSCtl_{CSS}$ is updated from one of the following sources:
   - The appropriate field of the *SRSMap* register, based on IPL, if the exception is an interrupt, $Cause_{IV} = 1$, $Config3_{VEIC} = 0$, and $Config3_{VInt} = 1$. These are the conditions for a vectored interrupt.
   - The EICSS field of the *SRSCtl* register if the exception is an interrupt, $Cause_{IV} = 1$, and $Config3_{VEIC} = 1$. These are the conditions for a vectored EIC interrupt.
   - The ESS field of the *SRSCtl* register in any other case. This is the condition for a non-interrupt exception, or a non-vectored interrupt.

Similarly, the rules for updating the fields in the *SRSCtl* register at the end of an exception or interrupt are as follows:

1. No field in the *SRSCtl* register is updated if any of the following conditions is true. In this case, step 2 is skipped.
   - A DERET is executed
   - An ERET is executed with $Status_{ERL} = 1$
2. $SRSCtl_{PSS}$ is copied to $SRSCtl_{CSS}$

These rules have the effect of preserving the *SRSCtl* register in any case of a nested exception or one which occurs before the processor has been fully initialized ($Status_{BEV} = 1$).

Privileged software may switch the current shadow set by writing a new value into $SRSCtl_{PSS}$, loading EPC with a target address, and doing an ERET.

## 6.5 Exception Vector Locations

The Reset, Soft Reset, NMI and EJTAG Debug exceptions are vectored to a specific location as shown in Table 6.5 and Table 6.6. Addresses for all other exceptions are a combination of a vector offset and a vector base address. In Release 1 of the architecture, the vector base address was fixed. In Release 2 of the architecture, software is allowed to specify the vector base address via the *EBase* register for exceptions that occur when $Status_{BEV}$ equals 0. Another degree of flexibility in the selection of the vector base address, for use when $Status_{BEV}$ equals 1, is provided via a set of input pins, *SI_UseExceptionBase* and *SI_ExceptionBase[29:12]*. Table 6.5 gives the vector base address when *SI_UseExceptionBase* equals 0, as a function of the exception and whether the BEV bit is set in the *Status* register. Table 6.6 gives the vector base addresses when *SI_UseExceptionBase* equals 1. As can be seen in Table 6.6, when *SI_UseExceptionBase equals 1*, the exception vectors for cases where *StatusBEV* equals 0 are not affected.

Table 6.7 gives the offsets from the vector base address as a function of the exception. Note that the IV bit in the *Cause* register causes Interrupts to use a dedicated exception vector offset, rather than the general exception vector. For implementations of Release 2 of the Architecture, Table 6.4 gives the offset from the base address in the case where $Status_{BEV}$ = 0 and $Cause_{IV}$ = 1. For implementations of Release 1 of the architecture in which $Cause_{IV}$ = 1, the vector offset is as if $IntCtl_{VS}$ were 0. Table 6.8 combines these two tables into one that contains all possible vector addresses as a function of the state that can affect the vector selection. To avoid complexity in the table, the vector address value assumes that the *EBase* register, as implemented in Release 2 devices, is not changed from its reset state and that $IntCtl_{VS}$ is 0.

**Table 6.5 Exception Vector Base Addresses when *SI_UseExceptionBase* equals 0**

| Exception | $Status_{BEV}$ | |
|---|---|---|
| | 0 | 1 |
| Reset, NMI | 16#BFC0.0000 | |
| EJTAG Debug (with ProbEn = 0 in the EJTAG_Control_register) | 16#BFC0.0480 | |
| EJTAG Debug (with ProbEn = 1 in the EJTAG_Control_register) | 16#FF20.0200 | |
| Cache Error | *For Release 1 of the architecture:* 16#A000.0000 *For Release 2 of the architecture:* $EBase_{31..30} \| 1 \| EBase_{28..12} \| 16\#000$ Note that $EBase_{31..30}$ have the fixed value 2#10 | 16#BFC0.0300 |
| Other | *For Release 1 of the architecture:* 16#8000.0000 *For Release 2 of the architecture:* $EBase_{31..12} \| 16\#000$ Note that $EBase_{31..30}$ have the fixed value 2#10 | 16#BFC0.0200 |
| '\|\|' denotes bit string concatenation | | |

**Table 6.6 Exception Vector Base Addresses when *SI_UseExceptionBase* equals 1**

| Exception | $Status_{BEV}$ | |
|---|---|---|
| | 0 | 1 |
| Reset, NMI | 2#10 \|\| SI_ExceptionBase[29:12] \|\| 16#000 | |
| EJTAG Debug (with ProbEn = 0 in the EJTAG_Control_register) | 2#10 \|\|SI_ExceptionBase[29:12] \|\| 16#480 | |
| EJTAG Debug (with ProbEn = 1 in the EJTAG_Control_register) | 16#FF20.0200 | |
| Cache Error | *For Release 1 of the architecture:* 16#A000.0000 *For Release 2 of the architecture:* $EBase_{31..30} \| 1 \| EBase_{28..12}$ \|\| 16#000 Note that $EBase_{31..30}$ have the fixed value 2#10 | 2#101 \|\| SI_ExceptionBase[28:12] \|\| 16#300 |
| Other | *For Release 1 of the architecture:* 16#8000.0000 *For Release 2 of the architecture:* $EBase_{31..12}$ \|\| 16#000 Note that $EBase_{31..30}$ have the fixed value 2#10 | 2#10 \|\| SI_ExceptionBase[29:12] \|\| 16#200 |
| '\|\|' denotes bit string concatenation | | |

**Table 6.7 Exception Vector Offsets**

| Exception | Vector Offset |
|---|---|
| TLB Refill, EXL = 0 | 16#000 |
| General Exception | 16#180 |
| Interrupt, $Cause_{IV}$ = 1 | 16#200 (In Release 2 implementations, this is the base of the vectored interrupt table when $Status_{BEV}$ = 0) |
| Reset, NMI | None (Uses Reset Base Address) |

**Table 6.8 Exception Vectors**

| Exception | SI_UseExceptionBase | $Status_{BEV}$ | $Status_{EXL}$ | $Cause_{IV}$ | EJTAG ProbEn | Vector<br>For Release 2 Implementations, assumes that EBase retains its reset state and that $IntCtl_{VS}$ = 0 |
|---|---|---|---|---|---|---|
| Reset, NMI | 0 | x | x | x | x | 16#BFC0.0000 |
| Reset, NMI | 1 | x | x | x | x | 2#10 \|\| SI_ExceptionBase[29:12] \|\| 16#000 |

**Table 6.8 Exception Vectors (Continued)**

| Exception | SI_UseExceptionBase | $Status_{BEV}$ | $Status_{EXL}$ | $Cause_{IV}$ | EJTAG ProbEn | Vector<br>For Release 2 Implementations, assumes that EBase retains its reset state and that $IntCtl_{VS}$ = 0 |
|---|---|---|---|---|---|---|
| EJTAG Debug | 0 | x | x | x | 0 | 16#BFC0.0480 |
| EJTAG Debug | 1 | x | x | x | 0 | 2#10 \|\| SI_ExceptionBase[29:12] \|\| 16#480 |
| EJTAG Debug | x | x | x | x | 1 | 16#FF20.0200 |
| TLB Refill | 0 | 0 | 1 | x | x | 16#8000.0180 |
| TLB Refill | 0 | 1 | 0 | x | x | 16#BFC0.0200 |
| TLB Refill | 1 | 1 | 0 | x | x | 2#10 \|\| SI_ExceptionBase[29:12] \|\| 16#200 |
| TLB Refill | 0 | 1 | 1 | x | x | 16#BFC0.0380 |
| TLB Refill | 1 | 1 | 1 | x | x | 2#10 \|\| SI_ExceptionBase[29:12] \|\| 16#380 |
| Cache Error | 0 | 0 | x | x | x | 16#A000.0100 |
| Cache Error | 0 | 1 | x | x | x | 16#BFC0.0300 |
| Cache Error | 1 | 1 | x | x | x | 2#101 \|\| SI_ExceptionBase[28:12] \|\| 16#300 |
| Interrupt | x | 0 | 0 | 0 | x | 16#8000.0180 |
| Interrupt | x | 0 | 0 | 1 | x | 16#8000.0200 |
| Interrupt | 0 | 1 | 0 | 0 | x | 16#BFC0.0380 |
| Interrupt | 1 | 1 | 0 | 0 | x | 2#10 \|\| SI_ExceptionBase[29:12] \|\| 16#380 |
| Interrupt | 0 | 1 | 0 | 1 | x | 16#BFC0.0400 |
| Interrupt | 1 | 1 | 0 | 1 | x | 2#10 \|\| SI_ExceptionBase[29:12] \|\| 16#400 |
| All others | 0 | 0 | x | x | x | 16#8000.0180 |
| All others | 0 | 1 | x | x | x | 16#BFC0.0380 |
| All others | 1 | 1 | x | x | x | 2#10 \|\| SI_ExceptionBase[29:12] \|\| 16#380 |
| ‘x’ denotes don’t care, ‘\|\|’ denotes bit string concatenation | | | | | | |

## 6.6 General Exception Processing

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

- If the EXL bit in the *Status* register is zero, the *EPC* register is loaded with the PC at which execution will be restarted and the BD bit is set appropriately in the *Cause* register (see Table 7.33). The value loaded into the *EPC* register is dependent on whether the processor implements the MIPS16 ASE, and whether the instruction is in the delay slot of a branch or jump which has delay slots. Table 6.9 shows the value stored in each of the CP0 PC registers, including *EPC*. For implementations of Release 2 of the Architecture if $Status_{BEV}$ = 0, the CSS field in the *SRSCtl* register is copied to the PSS field, and the CSS value is loaded from the appropriate source.

If the EXL bit in the *Status* register is set, the *EPC* register is not loaded and the BD bit is not changed in the *Cause* register. For implementations of Release 2 of the Architecture, the *SRSCtl* register is not changed.

**Table 6.9 Value Stored in EPC, ErrorEPC, or DEPC on an Exception**

| MIPS16 Implemented? | In Branch/Jump Delay Slot? | Value stored in EPC/ErrorEPC/DEPC |
|---|---|---|
| No | No | Address of the instruction |
| No | Yes | Address of the branch or jump instruction (PC-4) |
| Yes | No | Upper 31 bits of the address of the instruction, combined with the *ISA Mode* bit |
| Yes | Yes | Upper 31 bits of the branch or jump instruction (PC-2 in the MIPS16 ISA Mode and PC-4 in the 32-bit ISA Mode), combined with the *ISA Mode* bit |

- The CE, and ExcCode fields of the *Cause* registers are loaded with the values appropriate to the exception. The CE field is loaded, but not defined, for any exception type other than a coprocessor unusable exception.
- The EXL bit is set in the *Status* register.
- The processor is started at the exception vector.

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

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

**Operation:**

```
/* If Status_EXL is 1, all exceptions go through the general exception vector */
/* and neither EPC nor Cause_BD nor SRSCtl are modified */
if Status_EXL = 1 then
   vectorOffset ← 16#180
else
   /* For implementations that include the MIPS16e ASE, calculate potential */
   /* PC adjustment for exceptions in the delay slot */
   if Config1_CA = 0 then
      restartPC ← PC
      branchAdjust ← 4       /* Possible adjustment for delay slot */
   else
      restartPC ← PC_31..1 || ISAMode
      if (ISAMode = 0) or ExtendedMIPS16Instruction
         branchAdjust ← 4    /* Possible adjustment for 32-bit MIPS delay slot */
      else
         branchAdjust ← 2    /* Possible adjustment for MIPS16 delay slot */
      endif
   endif
   if InstructionInBranchDelaySlot then
      EPC ← restartPC - branchAdjust/* PC of branch/jump */
      Cause_BD ← 1
   else
      EPC ← restartPC              /* PC of instruction */
      Cause_BD ← 0
```

```
    endif

    /* Compute vector offsets as a function of the type of exception */
    NewShadowSet ← SRSCtl_ESS          /* Assume exception, Release 2 only */
    if ExceptionType = TLBRefill then
       vectorOffset ← 16#000
    elseif (ExceptionType = Interrupt) then
       if (Cause_IV = 0) then
          vectorOffset ← 16#180
       else
          if (Status_BEV = 1) or (IntCtl_VS = 0) then
             vectorOffset ← 16#200
          else
             if Config3_VEIC = 1 then
                VecNum ← Cause_RIPL
                NewShadowSet ← SRSCtl_EICSS
             else
                VecNum ← VIntPriorityEncoder()
                NewShadowSet ← SRSMap_IPL×4+3..IPL×4
             endif
             vectorOffset ← 16#200 + (VecNum × (IntCtl_VS || 2#00000))
          endif /* if (Status_BEV = 1) or (IntCtl_VS = 0) then */
       endif /* if (Cause_IV = 0) then */
    endif /* elseif (ExceptionType = Interrupt) then */

    /* Update the shadow set information for an implementation of */
    /* Release 2 of the architecture */
    if ((ArchitectureRevision ≥ 2) and (SRSCtl_HSS > 0) and (Status_BEV = 0) and
       (Status_ERL = 0)) then
       SRSCtl_PSS ← SRSCtl_CSS
       SRSCtl_CSS ← NewShadowSet
    endif
endif /* if Status_EXL = 1 then */

Cause_CE ← FaultingCoprocessorNumber
Cause_ExcCode ← ExceptionType
Status_EXL ← 1

if Config1_CA = 1 then
   ISAMode ← 0
endif

/* Calculate the vector base address */
if Status_BEV = 1 then
   vectorBase ← 16#BFC0.0200
else
   if ArchitectureRevision ≥ 2 then
      /* The fixed value of EBase_31..30 forces the base to be in kseg0 or kseg1 */
      vectorBase ← EBase_31..12 || 16#000
   else
      vectorBase ← 16#8000.0000
   endif
endif

/* Exception PC is the sum of vectorBase and vectorOffset */
PC ← vectorBase_31..30 || (vectorBase_29..0 + vectorOffset_29..0)
                          /* No carry between bits 29 and 30 */
```

## 6.7 Debug Exception Processing

All debug exceptions have the same basic processing flow:

- The *DEPC* register is loaded with the program counter (PC) value at which execution will be restarted and the DBD bit is set appropriately in the *Debug* register. The value loaded into the *DEPC* register is the current PC if the instruction is not in the delay slot of a branch, or the PC-4 of the branch if the instruction is in the delay slot of a branch.
- The DSS, DBp, DDBL, DDBS, DIB and DINT bits (D* bits at [5:0]) in the *Debug* register are updated appropriately depending on the debug exception type.
- Halt and Doze bits in the *Debug* register are updated appropriately.
- DM bit in the *Debug* register is set to 1.
- The processor is started at the debug exception vector.

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

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

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

**Operation:**

```
if InstructionInBranchDelaySlot then
    DEPC ← PC-4
    Debug_DBD ← 1
else
    DEPC ← PC
    Debug_DBD ← 0
endif
Debug_D* bits at at [5:0] ← DebugExceptionType
Debug_Halt ← HaltStatusAtDebugException
Debug_Doze ← DozeStatusAtDebugException
Debug_DM ← 1
if EJTAGControlRegister_ProbTrap = 1 then
    PC ← 0xFF20_0200
else
    PC ← 0xBFC0_0480
endif
```

The same debug exception vector location is used for all debug exceptions. The location is determined by the ProbTrap bit in the EJTAG Control register (ECR), as shown in Table 6.10.

**Table 6.10 Debug Exception Vector Addresses**

| ProbTrap bit in ECR Register | Debug Exception Vector Address |
|---|---|
| 0 | 0xBFC0_0480 |

**Table 6.10 Debug Exception Vector Addresses (Continued)**

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

# 6.8 Exceptions

The following subsections describe each of the exceptions listed in the same sequence as shown in Table 6.1.

## 6.8.1 Reset Exception

A reset exception occurs when the *SI_Reset* signal is asserted to the processor. This exception is not maskable. When a Reset exception occurs, the processor performs a full reset initialization, including aborting state machines, establishing critical state, and generally placing the processor in a state in which it can execute instructions from uncached, unmapped address space. On a Reset exception, the state of the processor 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* register is initialized with its boot state.
- The RP, BEV, TS, SR, NMI, and ERL fields of the *Status* register are initialized to a specified state.
- The I, R, and W fields of the *WatchLo* register are initialized to 0.
- The *ErrorEPC* register is loaded with PC-4 if the state of the processor indicates that it was executing an instruction in the delay slot of a branch. Otherwise, the *ErrorEPC* register is loaded with PC. Note that this value may or may not be predictable.
- PC is loaded with 0xBFC0_0000.

***Cause* Register ExcCode Value:**

None

**Additional State Saved:**

None

**Entry Vector Used:**

Reset (0xBFC0_0000)

**Operation:**

```
Random ← TLBEntries - 1
Wired ← 0
Config ← ConfigurationState
Status_RP ← 0
Status_BEV ← 1
Status_TS ← 0
Status_SR ← 0
```

```
Status_NMI ← 0
Status_ERL ← 1
WatchLo_I ← 0
WatchLo_R ← 0
WatchLo_W ← 0
if InstructionInBranchDelaySlot then
   ErrorEPC ← PC - 4
else
   ErrorEPC ← PC
endif
PC ← 0xBFC0_0000
```

## 6.8.2 Debug Single Step Exception

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

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

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

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

***Debug* Register Debug Status Bit Set**

DSS

**Additional State Saved**

None

**Entry Vector Used**

Debug exception vector

## 6.8.3 Debug Interrupt Exception

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

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

***Debug* Register Debug Status Bit Set**

DINT

**Additional State Saved**

None

**Entry Vector Used**

Debug exception vector

### 6.8.4 Non-Maskable Interrupt (NMI) Exception

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

- The BEV, TS, SR, NMI, and ERL fields of the *Status* register are initialized to a specified state.
- The *ErrorEPC* register is loaded with PC-4 if the state of the processor indicates that it was executing an instruction in the delay slot of a branch. Otherwise, the *ErrorEPC* register is loaded with PC.
- PC is loaded with 0xBFC0_0000.

***Cause* Register ExcCode Value:**

None

**Additional State Saved:**

None

**Entry Vector Used:**

Reset (0xBFC0_0000)

**Operation:**

```
Status_BEV ← 1
Status_TS ← 0
Status_SR ← 0
Status_NMI ← 1
Status_ERL ← 1
if InstructionInBranchDelaySlot then
   ErrorEPC ← PC - 4
else
   ErrorEPC ← PC
endif
PC ← 0xBFC0_0000
```

### 6.8.5 Interrupt Exception

The interrupt exception occurs when one or more of the six hardware, two software, or timer interrupt requests is enabled by the *Status* register and the interrupt input is asserted. See Section 6.3 "Interrupts" for more details about the processing of interrupts.

**Register ExcCode Value:**

Int

**Additional State Saved:**

**Table 6.11 Register States an Interrupt Exception**

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

**Entry Vector Used:**

See Section 6.3.2 "Generation of Exception Vector Offsets for Vectored Interrupts" for the entry vector used, depending on the interrupt mode the processor is operating in.

### 6.8.6 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 to match. This exception can only occur if instruction hardware breakpoints are implemented.

***Debug* Register Debug Status Bit Set:**

DIB

**Additional State Saved:**

None

**Entry Vector Used:**

Debug exception vector

### 6.8.7 Watch Exception — Instruction Fetch or Data Access

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

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

**Register ExcCode Value:**

WATCH

**Additional State Saved:**

**Table 6.12 Register States on a Watch Exception**

| Register State | Value |
|---|---|
| $Cause_{WP}$ | Indicates that the watch exception was deferred until after $Status_{EXL}$, $Status_{ERL}$, and $Debug_{DM}$ were zero. This bit directly causes a watch exception, so software must clear this bit as part of the exception handler to prevent a watch exception loop at the end of the current handler execution. |
| $WatchHi_{I,R,W}$ | Set for the watch channel that matched, and indicates which type of match there was. |

**Entry Vector Used:**

General exception vector (offset 0x180)

## 6.8.8 Address Error Exception — Instruction Fetch/Data Access

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

- Fetch an instruction, load a word, or store a word that is not aligned on a word boundary
- Load or store a halfword that is not aligned on a halfword boundary
- Reference the kernel address space from user mode

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

***Cause* Register ExcCode Value:**

ADEL: Reference was a load or an instruction fetch

ADES: Reference was a store

**Additional State Saved:**

**Table 6.13 CP0 Register States on an Address Exception Error**

| Register State | Value |
|---|---|
| *BadVAddr* | failing address |
| $Context_{VPN2}$ | UNPREDICTABLE |
| $EntryHi_{VPN2}$ | UNPREDICTABLE |
| *EntryLo0* | UNPREDICTABLE |
| *EntryLo1* | UNPREDICTABLE |

**Entry Vector Used:**

General exception vector (offset 0x180)

## 6.8.9 TLB Refill Exception — Instruction Fetch or Data Access

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

***Cause* Register ExcCode Value:**

TLBL: Reference was a load or an instruction fetch

TLBS: Reference was a store

**Additional State Saved:**

**Table 6.14 CP0 Register States on a TLB Refill Exception**

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

**Entry Vector Used:**

TLB refill vector (offset 0x000) if $Status_{EXL}$ = 0 at the time of exception;

general exception vector (offset 0x180) if $Status_{EXL}$ = 1 at the time of exception

## 6.8.10 TLB Invalid Exception — Instruction Fetch or Data Access

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

- No TLB entry matches a reference to a mapped address space; and the EXL bit is 1 in the *Status* register.
- A TLB entry matches a reference to a mapped address space, but the matched entry has the valid bit off.

***Cause* Register ExcCode Value:**

TLBL: Reference was a load or an instruction fetch

TLBS: Reference was a store

**Additional State Saved:**

**Table 6.15 CP0 Register States on a TLB Invalid Exception**

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

**Entry Vector Used:**

General exception vector (offset 0x180)

## 6.8.11 Cache Error Exception

A cache error exception occurs when an instruction or data reference detects a cache tag or data error. This exception is not maskable. Because the error was in a cache, the exception vector is to an unmapped, uncached address. This exception can be imprecise and the ErrorEPC may not point to the instruction that saw the errorIn the 34K, the cache memory may be shared between multiple VPEs on a virtual multiprocessor. In the event of a cache parity or other data integrity error, all VPEs sharing the cache may be affected, and all must take a Cache Error exception. It is the responsibility of software to coordinate any diagnostics or re-initialization of the shared cache, communicating by means other than cached storage.

L2 cache errors are considered to be imprecise. An L2 cache error on a data load operation can potentially corrupt the target GPR.

***Cause* Register ExcCode Value**

N/A

**Additional State Saved**

**Table 6.16 CP0 Register States on a Cache Error Exception**

| Register State | Value |
|---|---|
| *CacheErr* | Error state |
| *ErrorEPC* | Restart PC |

**Entry Vector Used**

Cache error vector (offset 16#100)

## 6.8.12 Bus Error Exception — Instruction Fetch or Data Access

A bus error exception occurs when an instruction or data access makes a bus request (due to a cache miss or an uncacheable reference) and that request terminates in an error. The bus error exception can occur on either an instruc-

tion fetch or a data read. Bus error exceptions cannot be generated on data writes. Bus error exceptions that occur on an instruction fetch have a higher priority than bus error exceptions that occur on a data access.

Bus Error exceptions on instruction fetch (IBE) are precise. Bus errors on data load operations (DBE) are considered to be imprecise. These errors are taken when the ERR code is returned on the *OC_SResp* input. Bus errors on data load operations can potentially corrupt the target GPR.

In the 34K, a DBE is delivered to the VPE where the operation was issued. A DBE exception may thus be taken by a TC other than the one which issued the failing operation. A per-TC TBE bit is defined to allow exception handlers to determine which TC(s) were associated with the failed bus transaction. If a DBE results from an operation that was combined across VPEs, a DBE exception must be delivered to all VPEs affected. Where the origin of the failure cannot be determined, all VPEs in a processor must take a DBE exception.

***Cause* Register ExcCode Value:**

IBE: Error on an instruction reference

DBE: Error on a data reference

**Additional State Saved:**

None

**Entry Vector Used:**

General exception vector (offset 0x180)

### 6.8.13 Debug Software Breakpoint Exception

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

**Debug Register Debug Status Bit Set:**

DBp

**Additional State Saved:**

None

**Entry Vector Used:**

Debug exception vector

### 6.8.14 Execution Exception — System Call

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

***Cause* Register ExcCode Value:**

Sys

**Additional State Saved:**

None

**Entry Vector Used:**

General exception vector (offset 0x180)

### 6.8.15 Execution Exception — Breakpoint

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

***Cause*** **Register ExcCode Value:**

Bp

**Additional State Saved:**

None

**Entry Vector Used:**

General exception vector (offset 0x180)

### 6.8.16 Execution Exception — Reserved Instruction

The reserved instruction exception is one of the execution exceptions. All of these exceptions have the same priority. A reserved instruction exception occurs when a reserved or undefined major opcode or function field is executed. This includes Coprocessor 2 instructions which are decoded reserved in the Coprocessor 2.

***Cause*** **Register ExcCode Value:**

RI

**Additional State Saved:**

None

**Entry Vector Used:**

General exception vector (offset 0x180)

### 6.8.17 Execution Exception — Coprocessor Unusable

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

- a corresponding coprocessor unit that has not been marked usable by setting its CU bit in the *Status* register
- CP0 instructions, when the unit has not been marked usable, and the processor is executing in user mode

***Cause*** **Register ExcCode Value:**

CpU

**Additional State Saved:**

**Table 6.17 Register States on a Coprocessor Unusable Exception**

| Register State | Value |
|---|---|
| $Cause_{CE}$ | unit number of the coprocessor being referenced |

**Entry Vector Used:**

General exception vector (offset 0x180)

## 6.8.18 Execution Exception — CorExtend block Unusable

The CorExtend block unusable exception is one of the execution exceptions. All of these exceptions have the same priority. A CEU exception occurs when an attempt is made to execute a CorExtend instruction when the CEE bit in the *Status* register is not set. It is dependent on the implementation of the CorExtend block, but this exception should be taken on any CorExtend instruction that modifies local state within the CorExtend block and can optionally be taken on other CorExtend instructions.

***Cause* Register ExcCode Value:**

CEU

**Additional State Saved:**

None

**Entry Vector Used:**

General exception vector (offset 0x180)

## 6.8.19 Execution Exception — Floating Point Exception

A floating point exception is initiated by the floating point coprocessor.

***Cause* Register ExcCode Value:**

FPE

**Additional State Saved:**

**Table 6.18 Register States on a Floating Point Exception**

| Register State | Value |
|---|---|
| *FCSR* | Indicates the cause of the floating point exception |

**Entry Vector Used:**

General exception vector (offset 0x180)

## 6.8.20 Execution Exception — Integer Overflow

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

***Cause*** **Register ExcCode Value:**

Ov

**Additional State Saved:**

None

**Entry Vector Used:**

General exception vector (offset 0x180)

### 6.8.21 Execution Exception — Trap

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

***Cause*** **Register ExcCode Value:**

Tr

**Additional State Saved:**

None

**Entry Vector Used:**

General exception vector (offset 0x180)

### 6.8.22 Execution Exception — C2E

A C2E exception is signalled from the optional coprocessor2 block on a coprocessor instruction.

***Cause*** **Register ExcCode Value:**

C2E

**Additional State Saved:**

None

**Entry Vector Used:**

General exception vector (offset 0x180)

### 6.8.23 Execution Exception — IS1

An IS1 exception is signalled from the optional coprocessor2 block on a coprocessor instruction.

***Cause*** **Register ExcCode Value:**

IS1

**Additional State Saved:**

None

**Entry Vector Used:**

General exception vector (offset 0x180)

## 6.8.24 Execution Exceptions — MT_ov, MT_under, MT_invalid, MT_yield_sched

- **MT_ov** - A Thread Overflow condition on a FORK, where a TC allocation request cannot be satisfied.
- **MT_under** - A Thread Underflow condition on a YIELD, where the termination and deallocation of a thread leaves no dynamically allocatable TCs activated on a VPE.
- **MT_invalid** - An Invalid qualifier condition, where a YIELD instruction specifies an invalid condition for resuming execution.
- **MY_yield_sched** - A YIELD scheduler exception, where a valid YIELD instruction would have caused a rescheduling of a TC, and the YIELD Intercept bit is set.

***Cause* Register ExcCode Value:**

Thread.

**Additional State Saved:**

There is a sub-cause filed in *VPEControl*[EXCPT], which indicates the type of Thread exception. Table 6.19 shows the different Thread Exception Codes.

**Table 6.19 Thread exception codes in VPEControl[EXCPT]**

| Value | Exception |
|---|---|
| 0 | MT_ov |
| 1 | MT_under |
| 2 | MT_invalid |
| 3 | MT_gs |
| 4 | MT_yield_sched |
| 5 | MT_gss |

**Entry Vector Used:**

General exception vector (offset 0x180)

## 6.8.25 Thread Exceptions — MT_gs, MT_gss

- **MT_gs** - A Gating Storage exception condition, where implementation dependent logic associated with gating or inter-thread communication (ITC) storage requires software intervention.
- **MT_gss** - A Gating Storage Scheduler exception, where a Gating Storage load or store would have blocked and caused a rescheduling of a TC, and the GS Intercept bit is set.

***Cause* Register ExcCode Value:**

Thread.

**Additional State Saved:**

There is a sub-cause filed in *VPEControl*[EXCPT], which indicates the type of Thread exception. Table 6.19 shows the different Thread Exception Codes.

**Entry Vector Used:**

General exception vector (offset 0x180)

## 6.8.26 Debug Data Break Exception

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

***Debug* Register Debug Status Bit Set:**

DDBL for a load instruction or DDBS for a store instruction

**Additional State Saved:**

None

**Entry Vector Used:**

Debug exception vector

## 6.8.27 TLB Modified Exception — Data Access

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

- The matching TLB entry is valid, but not dirty.

***Cause* Register ExcCode Value:**

Mod

**Additional State Saved:**

**Table 6.20 Register States on a TLB Modified Exception**

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

**Entry Vector Used:**

General exception vector (offset 0x180)

## 6.9 Exception Handling and Servicing Flowcharts

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

- General exceptions and their exception handler
- TLB miss exception and their exception handler
- Reset and NMI exceptions, and a guideline to their handler.
- Debug exceptions

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

## Figure 6.3 General Exception Handler (HW)

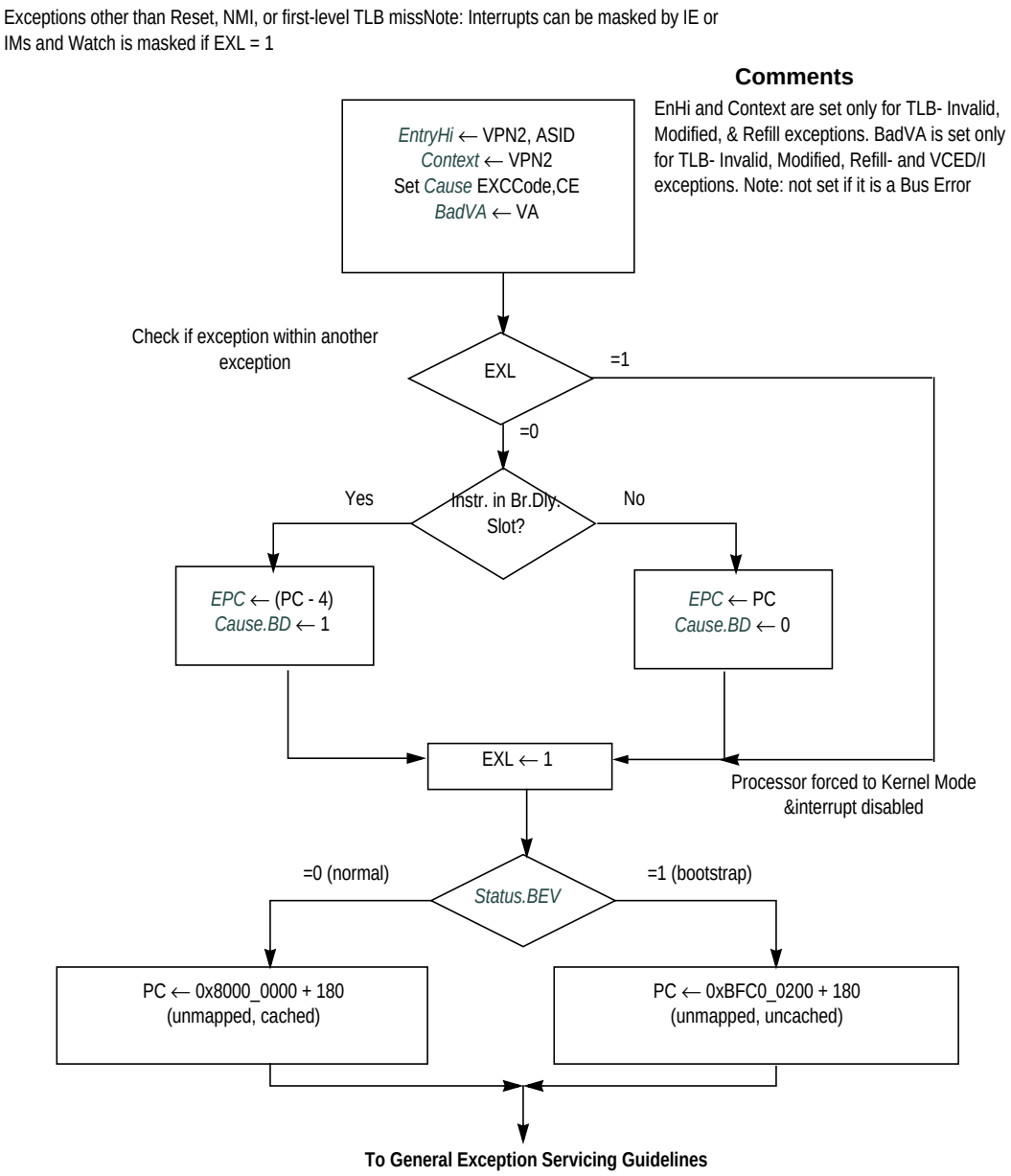


**Figure 6.4 General Exception Servicing Guidelines (SW)**

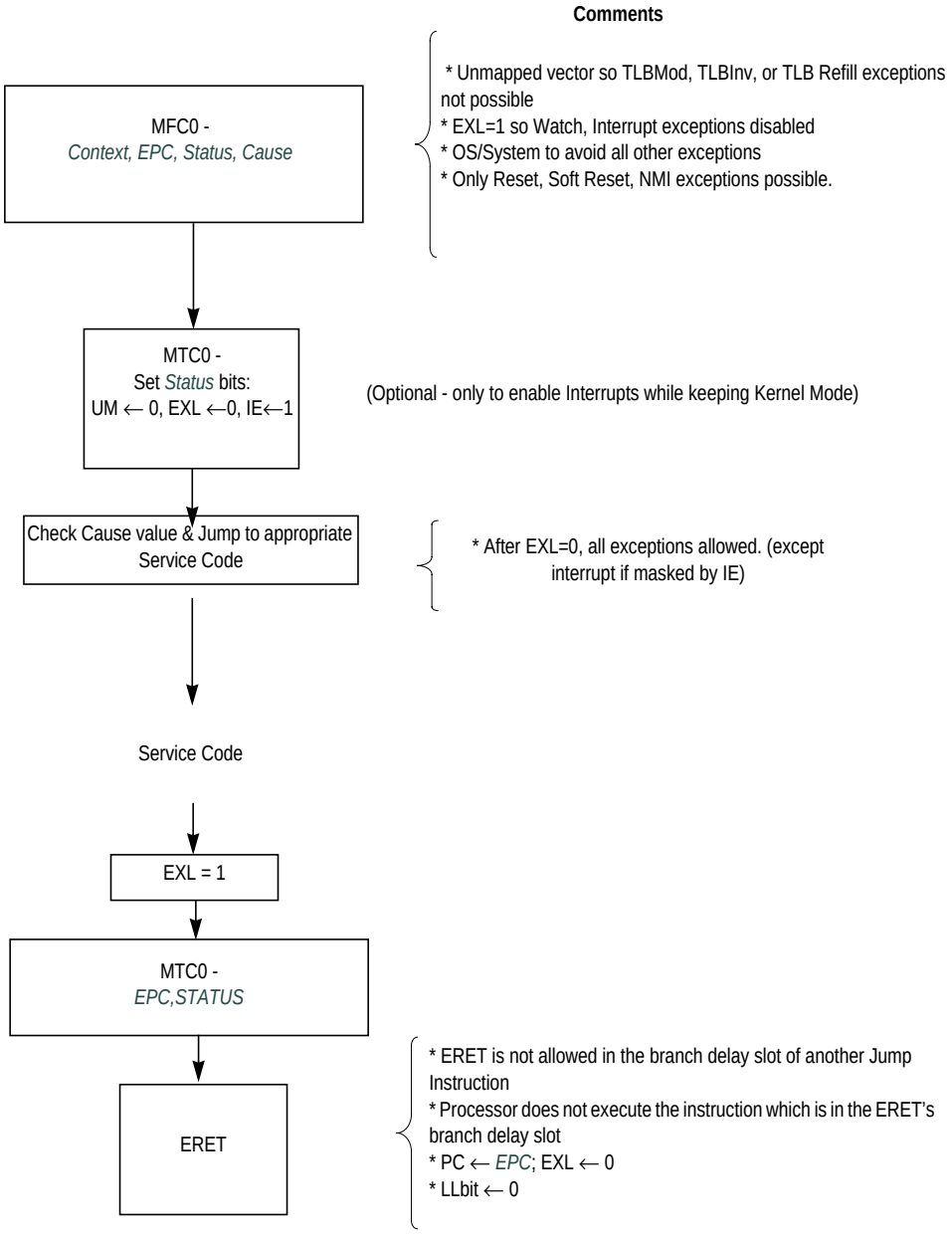


**Figure 6.5 TLB Miss Exception Handler (HW)**

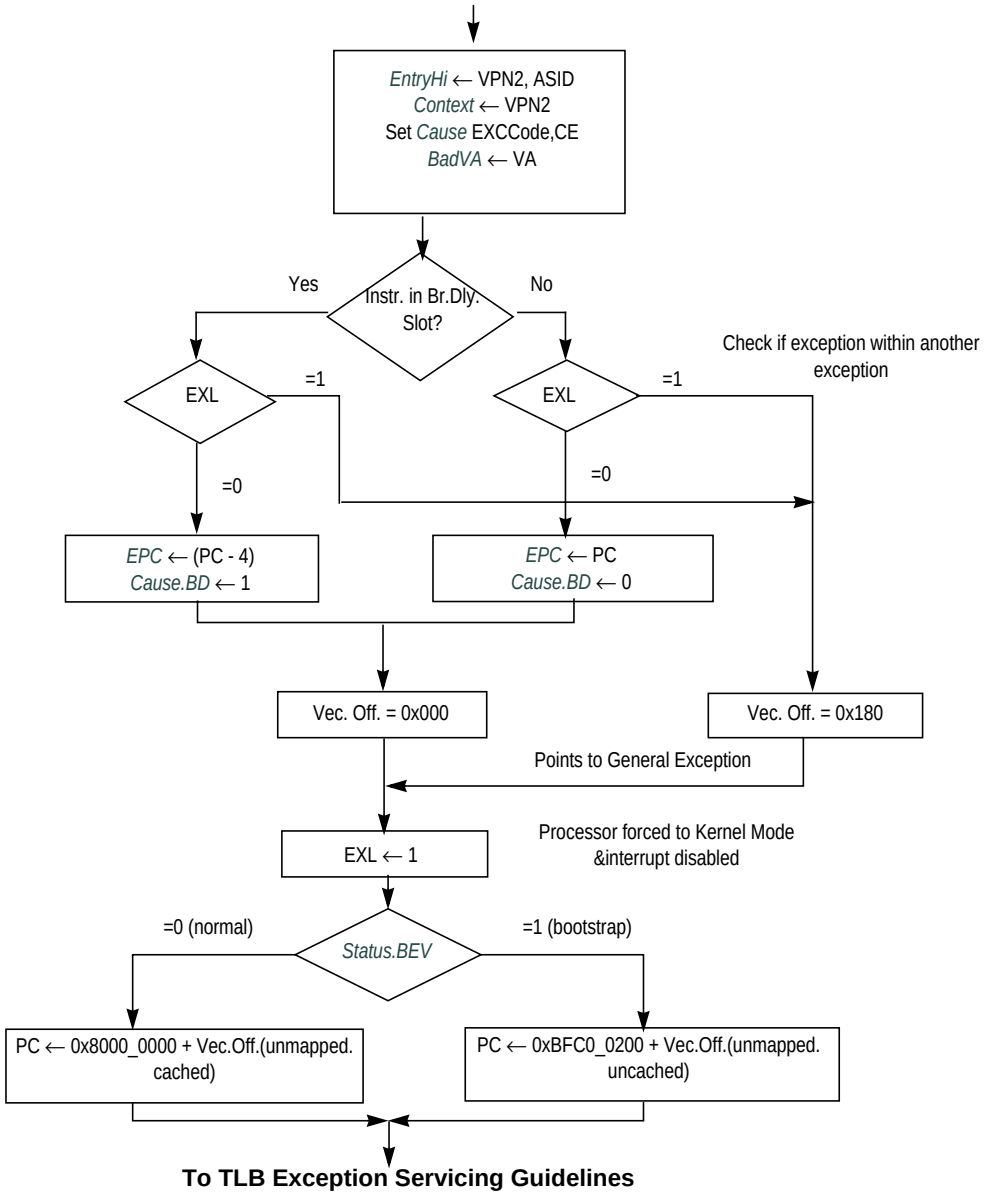


**Figure 6.6 TLB Exception Servicing Guidelines (SW)**

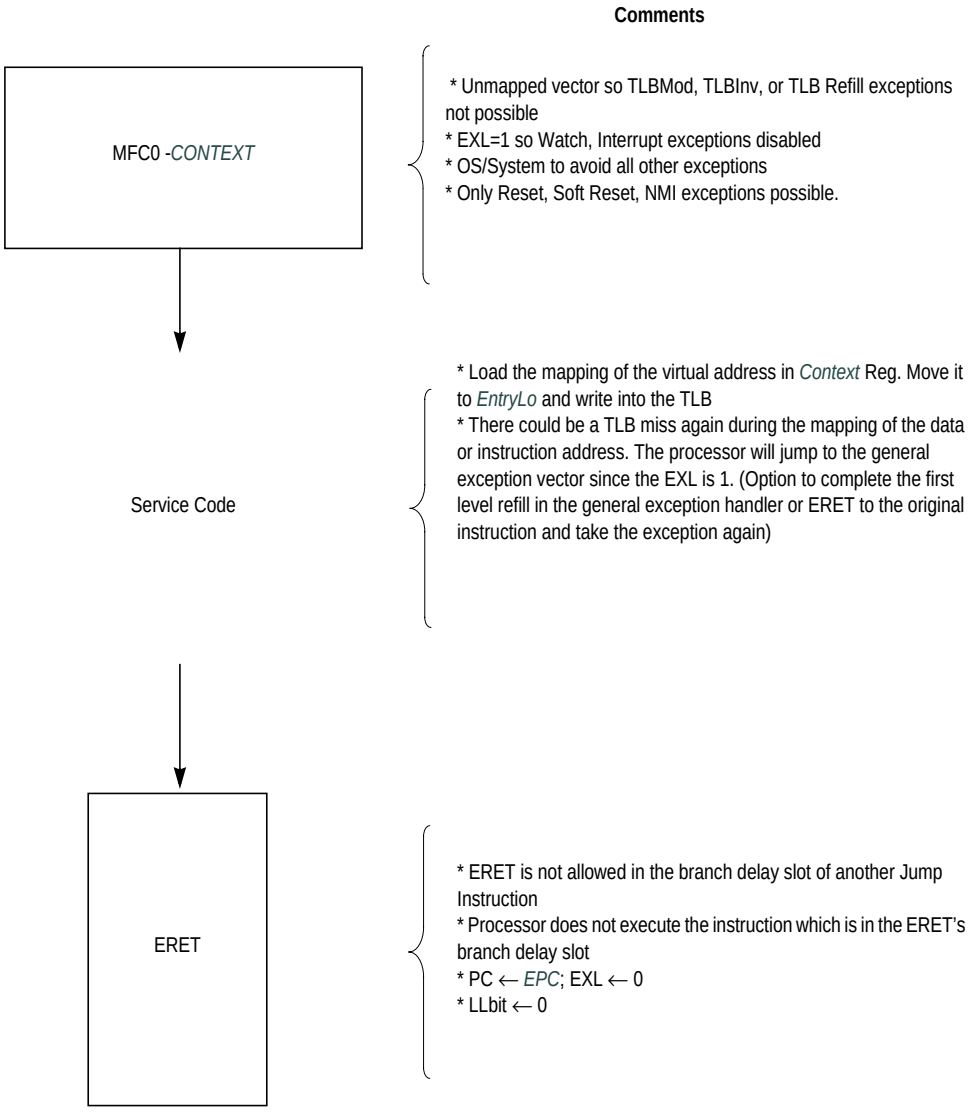
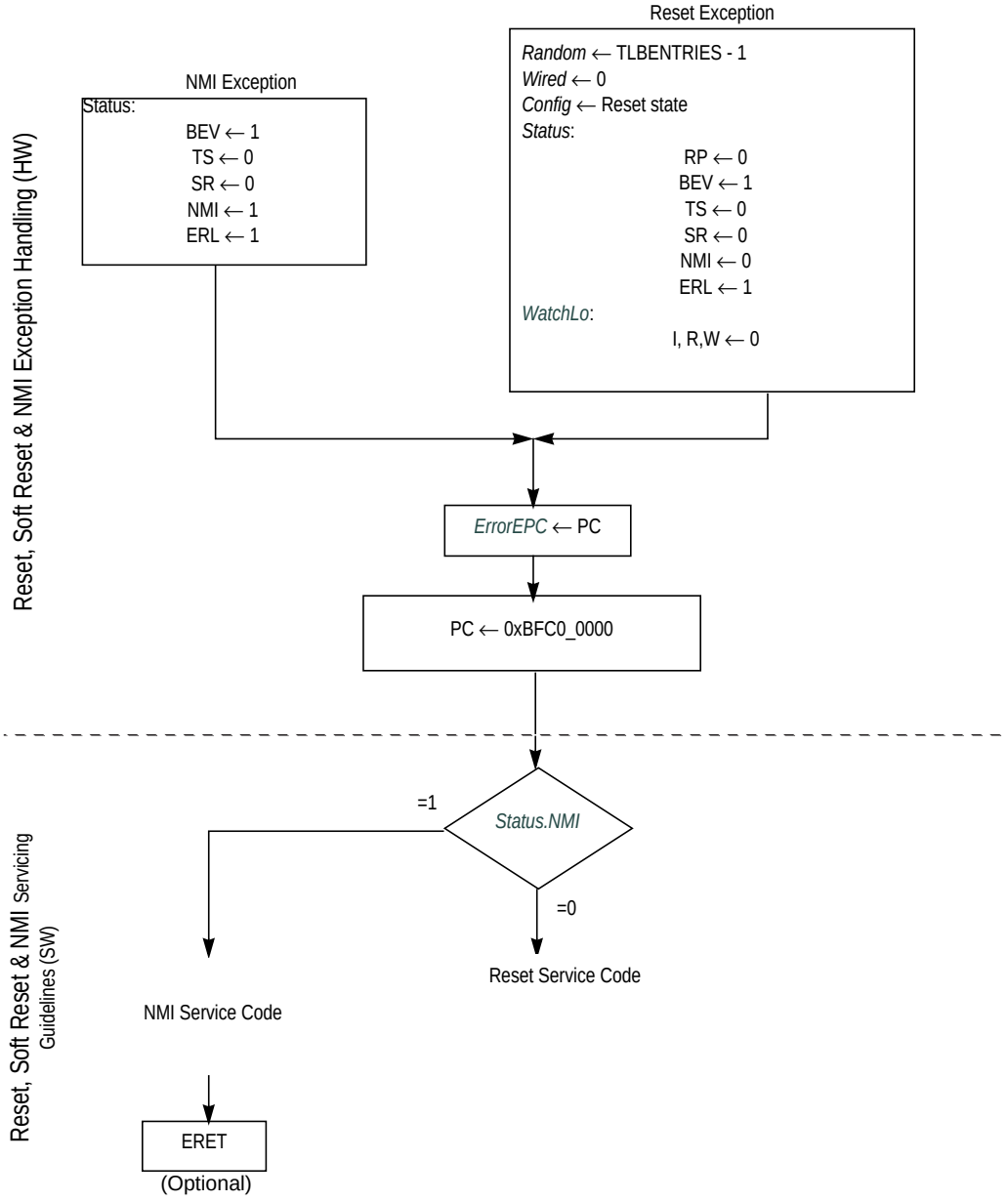


**Figure 6.7 Reset and NMI Exception Handling and Servicing Guidelines**



*Chapter 7*

# CP0 Registers of the 34K™ Core

The System Control Coprocessor (CP0) provides the register interface to the 34K processor core and supports memory management, address translation, exception handling, and other privileged operations. Each CP0 register has a unique number that identifies it; this number is referred to as the *register number*. For instance, the *PageMask* register is register number 5. A register may also have a select After updating a CP0 register there is a hazard period of zero or more instructions from the update instruction (MTC0) and until the effect of the update has taken place in the core.

This chapter contains the following sections:



## 7.1 CP0 Register Summary

Table 7.1 lists the CP0 registers in numerical order. The individual registers are described throughout this chapter.

**Table 7.1 CP0 Registers**

| Register | | | Function | Per | | |
|---|---|---|---|---|---|---|
| Number | Select | Name | | VPE | TC | Proc |
| 0 | 0 | Index[1] | Index into the TLB array. This register is reserved if the TLB is not implemented. | X | | |
| 0 | 1 | MVPControl | CPU-wide multithreading control. | | | X |
| 0 | 2-3 | MVPConf0-1 | CPU's multithreading resources | | | X |
| 1 | 0 | Random[1] | Randomly generated index into the TLB array. This register is reserved if the TLB is not implemented. | X | | |
| 1 | 1 | VPEControl | VPE control and status | X | | |
| 1 | 2-3 | VPEConf0-1 | Initializable per-VPE resource lists | X | | |
| 1 | 4 | YQMask | Defines valid inputs for yield instruction | X | | |
| 1 | 5 | VPESchedule | Per-VPE thread policy hints | X | | |
| 1 | 6 | VPEScheFBack | Per-VPE information from policy manager | X | | |
| 1 | 7 | VPEOpt | Per-VPE cache-way inhibition | X | | |
| 2 | 0 | EntryLo0[1] | Low-order portion of the TLB entry for even-numbered virtual pages. This register is reserved if the TLB is not implemented. | | | |
| 2 | 1 | TCStatus | Status and control for each TC | | X | |
| 2 | 2 | TCBind | VPE affiliation and own TC number of this TC | | X | |
| 2 | 3 | TCRestart | Where this TC will next fetch code from | | X | |

**Table 7.1 CP0 Registers (Continued)**

| Register | | | Function | Per | | |
|---|---|---|---|---|---|---|
| Number | Select | Name | | VPE | TC | Proc |
| 2 | 4 | TCHalt | Set 1 to freeze the TC for inspection/modification | | X | |
| 2 | 5 | TCContext | Read/write scratch register for OS to maintain thread ID | | X | |
| 2 | 6 | TCSchedule | Per-TC thread scheduling hints | | X | |
| 2 | 7 | TCScheFBack | Per-TC information from policy manager | | X | |
| 3 | 0 | EntryLo1[1] | Low-order portion of the TLB entry for odd-numbered virtual pages. This register is reserved if the TLB is not implemented. | X | | |
| 4 | 0 | Context[2] | Pointer to page table entry in memory. This register is reserved if the TLB is not implemented. | X | | |
| 5 | 0 | PageMask | PageMask controls the variable page sizes in TLB entries. This register is reserved if the TLB is not implemented. | X | | |
| 6 | 0 | Wired[1] | Controls the number of fixed (“wired”) TLB entries. This register is reserved if the TLB is not implemented. | X | | |
| 6 | 1-5 | SRSConf0-4 | Write these to use TCs as shadow registers | X | | |
| 7 | 0 | HWREna | Enables access via the RDHWR instruction to selected hardware registers in non-privileged mode. | X | | |
| 8 | 0 | BadVAddr[2] | Reports the address for the most recent address-related exception. | X | | |
| 9 | 0 | Count[2] | Processor cycle count. | X | | |
| 10 | 0 | EntryHi[1] | High-order portion of the TLB entry. This register is reserved if the TLB is not implemented. | X | X[3] | |
| 11 | 0 | Compare[2] | Timer interrupt control. | X | | |
| 12 | 0 | Status[2] | Processor status and control. | X | X[4] | |
| 12 | 1 | IntCtl[2] | Setup for interrupt vector and interrupt priority features. | X | | |
| 12 | 2 | SRSCtl[2] | Shadow register set selectors | X | | |
| 12 | 3 | SRSMap[2] | In vectored interrupt mode, determines which shadow set is used for each interrupt source. | X | | |
| 13 | 0 | Cause[2] | Cause of last exception. | X | | |
| 14 | 0 | EPC[2] | Program counter at last exception. | X | | |
| 15 | 0 | PRId | Processor identification and revision. | X | | |
| 15 | 1 | EBase | Exception base address. | X | | |
| 16 | 0 | Config | Configuration register. | X | | |
| 16 | 1-2 | Config1-2 | Configuration for MMU, caches etc. | X | | |
| 16 | 3 | Config3 | Interrupt and ASE capabilities | X | | |
| 16 | 7 | Config7 | 34K family-specific configuration register. | X | | |
| 17 | 0 | LLAddr | Address associated with last LL instruction of a "load-linked/store-conditional" instruction pair. | X | | |
| 18 | 0-1 | WatchLo0-1[2] | Low-order watchpoint address associated with instruction watchpoints. | X | | |
| 18 | 2-3 | WatchLo2-3[2] | Low-order watchpoint address associated with data watchpoints. | X | | |

**Table 7.1 CP0 Registers (Continued)**

| Register | | | Function | Per | | |
|---|---|---|---|---|---|---|
| Number | Select | Name | | VPE | TC | Proc |
| 19 | 0-1 | WatchHi0-1[2] | High-order watchpoint address used for instruction watchpoints. | X | | |
| 19 | 2-3 | WatchHi2-3[2] | High-order watchpoint address used for data watchpoints. | X | | |
| 23 | 0 | Debug[5] | EJTAG Debug register. | X | | |
| 24 | 0 | DEPC[5] | Restart address from last EJTAG debug exception. | X | | |
| 25 | 0 | PerfCtl0 | Performance counter 0 control. | | | X |
| 25 | 1 | PerfCnt0 | Performance counter 0. | | | X |
| 25 | 2 | PerfCtl1 | Performance counter 1 control. | | | X |
| 25 | 3 | PerfCnt1 | Performance counter 1. | | | X |
| 25 | 4 | PerfCtl2 | Performance counter 2 control. | | | X |
| 25 | 5 | PerfCnt2 | Performance counter 2. | | | X |
| 25 | 6 | PerfCtl3 | Performance counter 3 control. | | | X |
| 25 | 7 | PerfCnt3 | Performance counter 3. | | | X |
| 26 | 0 | ErrCtl | Software test enable of way-select and Data RAM arrays for I-Cache and D-Cache. | X | | |
| 27 | 0 | CacheErr | Records information about cache parity errors | X | | |
| 28 | 0 | TagLo0 | Cache tag read/write interface for I-cache. | X | | |
| 28 | 1 | DataLo0 | Low-order data read/write interface for I-cache. | X | | |
| 28 | 2 | TagLo1 | Cache tag read/write interface for D-cache. | X | | |
| 28 | 3 | DataLo1 | Low-order data read/write interface for D-cache. | X | | |
| 28 | 4 | TagLo2 | Cache tag read/write interface for L2-cache. | X | | |
| 28 | 5 | DataLo2 | Low-order data read/write interface for L2-cache. | X | | |
| 29 | 0 | DataHi0 | Upper bits for I-cache interface. This is only accessible in 64-bit units. | X | | |
| 29 | 4 | DataHi2 | Upper bits for L2-cache | X | | |
| 30 | 3 | ErrorEPC[2] | Program counter at last error. | X | | |
| 31 | 0 | DeSAVE[5] | Debug handler scratchpad register. | X | | |

1. Registers used in memory management.
2. Registers used in exception processing.
3. *ASID* per-TC. See Section 7.2.27 “EntryHi Register (CP0 Register 10, Select 0)”.
4. *KSU* and *CU0-3* per-TC. See Section 7.2.29 “Status Register (CP0 Register 12, Select 0)”.
5. Registers used in debug.

## 7.2 CP0 Register Descriptions

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

For each register described below, field descriptions include the read/write properties of the field, and the reset state of the field. For single bit fields, the name is truncated to a single character which is then shown outside brackets in

the Fields|Name column; for example, *(TLB)**S*** for the TLB Sharable bit in the *MVPConf0* register. For the read/write properties of the field, the following notation is used:

**Table 7.2 CP0 Register Field Types**

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

## 7.2.1 Index Register (CP0 Register 0, Select 0)

The *Index* register is a 32-bit read/write register that contains the index used to access the TLB for TLBP, TLBR, and TLBWI instructions. The width of the index field is implementation-dependent as a function of the number of TLB entries that are implemented. The minimum value for TLB-based MMUs is $Ceiling(Log_2(TLBEntries))$.

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

This register is only valid with the TLB. It is reserved if the FM is implemented.

**Figure 7.1 Index Register Format**

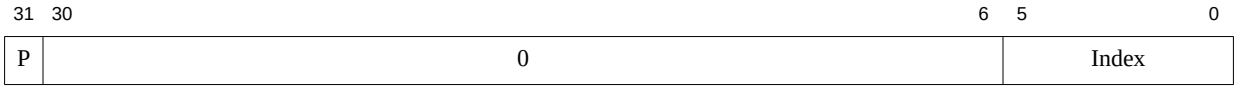


**Table 7.3 Index Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| P | 31 | Probe Failure. Set to 1 when the previous TLBProbe (TLBP) instruction failed to find a match in the TLB. | R/W | Undefined |
| 0 | 30:6 | Must be written as zeros; returns zeros on reads. | 0 | 0 |
| Index | 5:0 | Index to the TLB entry affected by the TLBRead and TLBWrite instructions.<br>For 16 or 32 entry TLBs, behavior is undefined if index points to a non-existent entry. | R/W | Undefined |

## 7.2.2 MVPControl Register (CP0 Register 0, Select 1)

The *MVPControl* register is a read/write per-CPU control/status register. This register is writable only when $VPEConf0_{MVP}$ is set.

**Figure 7.2 MVPControl Register Format**

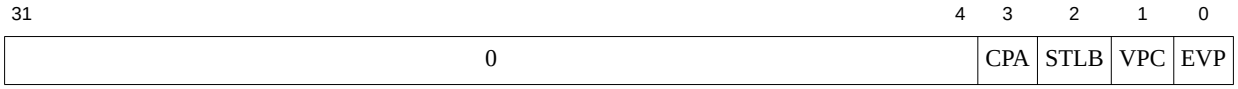



**Table 7.4 MVPControl Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| 0 | 31:4 | Must be written as zeros; returns zeros on reads. | 0 | 0 |
| CPA | 3 | Cache Partitioning Active: If set, the *IWX* and *DWX* fields of the *VPEOpt* register control the allocation of cache lines. | R/W | 0 |
| STLB | 2 | Set to enable TLB sharing between VPEs. Unset to disable TLB sharing. | R/W | 0 |
| VPC | 1 | Set to write to configuration register fields which are otherwise read-only on conventional MIPS32® CPUs. Unset to make the fields in the Config*NN* registers read-only. | R/W | 0 |
| EVP | 0 | Set to execute instructions for all threads. Unset to execute instructions only for the thread which is running when this bit is cleared. Set and cleared by the EVPE/DVPE instructions. | R/W | 0 |

## 7.2.3 MVPConf0-1 Registers (CP0 Register 0, Select 2-3)

The *MVPConf0-1* registers provide read-only multithreading-specific configuration information.

**Figure 7.3 MVPConf0 Register Format**

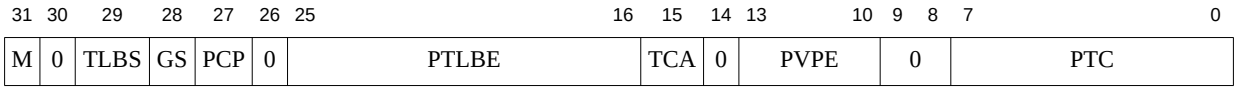


**Table 7.5 MVPConf0 Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| M | 31 | This bit reads 1 if the *MVPConf1* register is present, otherwise it reads 0. | R | 1 |
| TLBS | 29 | TLB Sharable: Indicates that TLB sharing amongst all VPEs is possible. TLB sharing is enabled by the *STLB* bit of the *MVPControl* register. | R | 1 if both VPEs have TLB |
| GS | 28 | Gating Storage Present. Indicates that the processor is configured to support gating storage operations. Externally set on reset based on *IT_num_entries*. If *IT_num_entries* is greater than zero, this bit is set to 1. | R | Preset |
| PCP | 27 | Programmable Cache Partitioning: If set, indicates that the allocation behavior of the “ways” of the primary instruction and data caches can be controlled via the *VPEOpt* register’s *IWX* and *DWX* fields. | R | 1 if multiple VPEs |
| PTLBE | 25:16 | Total processor compliment of allocatable TLB entry pairs.TLB configuration is fixed, so PTLBE is zero. | R | 0 |
| TCA | 15 | TCs Allocatable: If set, TCs may be assigned to VPEs by writing the *CurVPE* field of the *TCBind* register of each TC while the *VPC* bit of *MVPControl* is set. | R | 1 |
| PVPE | 13:10 | Total processor compliment of VPE contexts - 1. | R | Preset: 0 or 1 |
| PTC | 7:0 | Total processor compliment of TCs - 1. | R | Preset: 0 to 8 |
| 0 | 30, 26, 14, 9:8 | Must be written as zeros; returns zeros on reads. | 0 | 0 |

**Figure 7.4 MVPConf1 Register Format**

| 31 | 30 | 29 28 27 | 20 19 18 17 | 10 9 8 7 | 0 |
|---|---|---|---|---|---|
| C1M | C1F | 0 | PCX | 0 | PCP2 | 0 | PCP1 |

**Table 7.6 MVPConf1 Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| C1M | 31 | If set, floating point unit (co-processor 1) implements the MDMX™ extension to the instruction set. | R | Preset |
| C1F | 30 | If set, floating point unit (co-processor 1) implements 64-bit instructions | R | Preset |
| PCX | 27:20 | Number of register set contexts available for CorExtend™. | R | Preset |
| PCP2 | 17:10 | Number of register set contexts available for co-processor 2. | R | Preset |
| PCP1 | 7:0 | Number of register set contexts available for co-processor 1. | R | Preset |
| 0 | 29:28, 19:18, 9:8 | Must be written as zeros; returns zeros on reads. | 0 | 0 |

## 7.2.4 Random Register (CP0 Register 1, Select 0)

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

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

- A lower bound is set by the number of TLB entries reserved for exclusive use by the operating system (the contents of the *Wired* register). The entry indexed by the *Wired* register is the first entry available to be written by a TLB Write Random operation.
- An upper bound is set by the total number of TLB entries minus 1.

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

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

This register is only valid with the TLB. It is reserved if the FM is implemented.

**Figure 7.5 Random Register Format**

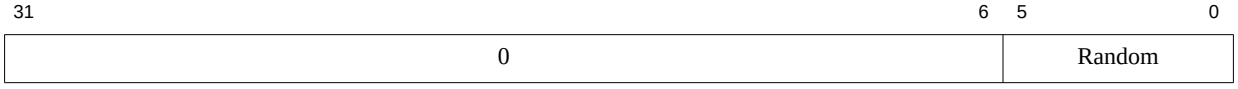



**Table 7.7 Random Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| 0 | 31:6 | Must be written as zero; returns zero on reads. | 0 | 0 |
| Random | 5:0 | TLB Random Index | R | TLB Entries - 1 |

## 7.2.5 VPEControl Register (CP0 Register 1, Select 1)

The *VPEControl* register contains per-VPE status and control fields for exception and MFTR/MTTR instruction support.

**Figure 7.6 VPEControl Register Format**

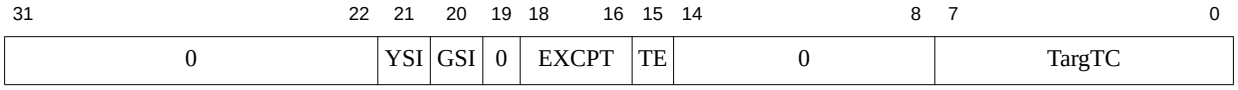



**Table 7.8 VPEControl Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| 0 | 31:22, 19, 14:8 | Must be written as zeros; returns zeros on reads. | 0 | 0 |

**Table 7.8 VPEControl Register Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| YSI | 21 | If set, any thread that would be blocked by a YIELD instruction can be detected (by exception processing). | R/W | 0 |
| GSI | 20 | If set, any thread that would be blocked by a gating storage access can be detected (by exception processing). | R/W | 0 |
| EXCPT | 18:16 | Encodes the cause of the last thread exception:<br>Encoding / Description<br>0 — Thread underflow on YIELD.<br>1 — Thread overflow on FORK.<br>2 — Bad qualifier fed to YIELD.<br>3 — Exception on gating storage operation<br>4 — YIELD which would have blocked run while *VPEControl*$_{YSI}$ is set to 1.<br>5 — Gating storage access which would have blocked attempted while *VPEControl*$_{GSI}$ is set to 1. | R/W | Undefined |
| TE | 15 | If set, multi-threading is enabled. If unset, only one TC is allowed to issue instructions. Normally set/unset using the DMT/EMT instructions. | R/W | 0 |
| TargTC | 7:0 | Selects the TC number of the "other thread context" in the MTTR/MFTR instructions. | R/W | Undefined |

## 7.2.6 VPEConf0-1 Register (CP0 Register 1, Select 2,3)

The *VPEConf0-1* registers define the available resources on the VPE.

**Figure 7.7 VPEConf0 Register Format**

| 31 | 30 | 29 28 | 21 | 20 | 19 | 18 | 17 | 16 | 15 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| M | 0 | XTC | | 0 | TCS | SCS | DCS | ICS | 0 | MVP | VPA |

**Table 7.9 VPEConf0 Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| M | 31 | This bit reads 1 if the *VPEConf1* register is present, otherwise it reads 0. | R | 1 |
| XTC | 28:21 | When only one TC in a VPE is runnable because the VPE is in exception mode or *VPEControl*$_{TE}$ is clear, this field identifies that one runnable TC. | R/W | 0 |
| TCS | 19 | Tertiary Cache Shared: If set, the tertiary cache is shared with other VPEs. | R | Preset |
| SCS | 18 | Secondary Cache Shared: If set, the secondary cache is shared with other VPEs. | R | Preset |

**Table 7.9 VPEConf0 Register Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| DCS | 17 | Data Cache Shared: If set, the L1 data cache is shared with other VPEs. It is reset to 1 if two VPEs are present. | R | 1 |
| ICS | 16 | Instruction Cache Shared: If set, the L1 instruction cache is shared with other VPEs. It is reset to 1 if two VPEs are present. | R | 1 |
| MVP | 1 | Master Virtual Processor: If set, registers in different VPEs (or in the TCs of different VPE affiliation) are made writable. It also controls read/write access to *MVPControl* (see Section 7.2.2 "MVPControl Register (CP0 Register 0, Select 1)"). It is reset to 1 for VPE0, 0 for others. | R/W | 1/0 |
| VPA | 0 | Virtual Processor Activated: If zero, no TCs bound to this VPE will run. It is reset to 1 for VPE0, 0 for others. | R/W | 1/0 |
| 0 | 30:29, 20, 15:2 | Must be written as zeros; returns zeros on reads. | 0 | 0 |

**Figure 7.8 VPEConf1 Register Format**

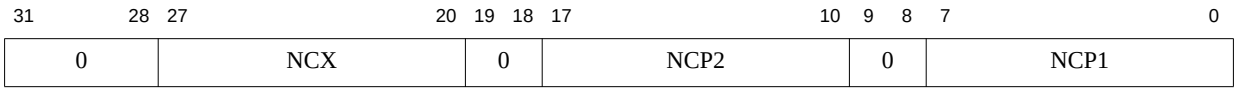



**Table 7.10 VPEConf1 Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| 0 | 31:28, 19:19, 9:8 | Must be written as zeros; returns zeros on reads. | 0 | 0 |
| NCX | 27:20 | Number of UDI contexts available to this VPE. | R | External[1] |
| NCP2 | 17:10 | Number of Coprocessor 2 contexts available to this VPE. | R | External[1] |
| NCP1 | 7:0 | Number of Floating Point Unit (FPU) / Coprocessor 1 contexts available to this VPE. | R | External[1] |

1. Externally set on reset based on *SI_VpeCX*, *SI_VpeCP2*, and *SI_VpeCP1*. When only one CorExtend/Coproccessor1/2 context is available, it is assigned on reset to the VPE indicated by *SI_VpeCX*, *SI_VpeCP2*, or *SI_VpeCP1*. These fields are writable when $MVPControl_{VPC}$ is set.

## 7.2.7 YQMask Register (CP0 Register 1, Select 4)

The *YQMask* register is instantiated per-VPE. The 34K core only supports 16 mask bits.

**Figure 7.9 YQMask Register Format**

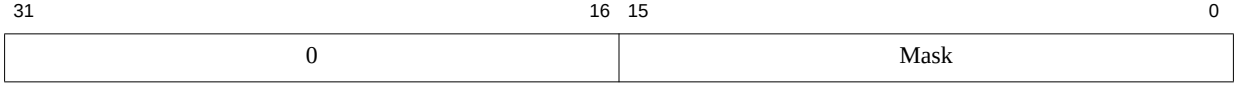


**Table 7.11 YQMask Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| 0 | 31:16 | Must be written as zero; return zero on read. | 0 | 0 |
| Mask | 15:0 | Bit vector which determines which values may be used as external state qualifiers by YIELD instructions. | R/W | 0 |

## 7.2.8 VPESchedule Register (CP0 Register 1, Select 5)

The *VPESchedule* is a per-VPE value whose interpretation is scheduler implementation-dependent. For example, it could encode a description of the overall requested issue bandwidth for the associated VPE, or it could encode a priority level.

A *VPESchedule* register value of zero is the default, and should result in a well-behaved default scheduling of the associated VPE.

## 7.2.9 VPEScheFBack Register (CP0 Register 1, Select 6)

The Scheduler Feedback is a per-VPE feedback value from scheduler hardware to software. The interpretation is scheduler implementation-dependent. For example, it might encode the total number of instructions retired in the instruction streams on the associated VPE since the last time the value was cleared by software.

## 7.2.10 VPEOpt Register (CP0 Register 1, Select 7)

The optional *VPEOpt* register is instantiated per-VPE. If way exclusion is enabled via the $MVPControl_{CPA}$ bit, the fields in this register will control which ways should be excluded from the replacement scheme for this VPE.

**Figure 7.10 VPEOpt Register Format**

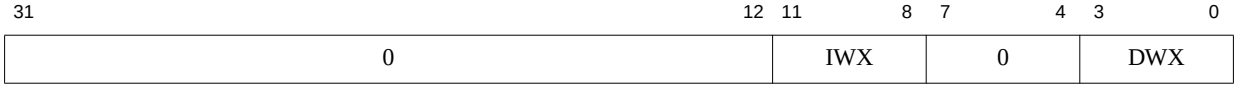



**Table 7.12 VPEOpt Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| 0 | 31:12, 7:4 | Must be written as zero; return zero on read. | R | 0 |

**Table 7.12 VPEOpt Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| **Name** | **Bits** | | | |
| IWX3<br>..<br>IWX0 | 11:8 | Instruction cache way exclusion mask. If programmable cache allocation is enabled via the CPA bit in the *MVPControl* register, a VPE can exclde an arbitrary subset of the ways of the primary instruction cache from allocation by the cache controller on behalf of the VPE<br><br>**Bit / Name / Meaning**<br>11 / IWX3 / If set, I-cache way 3 will not be allocated for the VPE<br>10 / IWX2 / If set, I-cache way 2 will not be allocated for the VPE<br>9 / IWX1 / If set, I-cache way 1 will not be allocated for the VPE<br>8 / IWX0 / If set, I-cache way 0 will not be allocated for the VPE | R/W | 0 |
| DWX3<br>..<br>DWX0 | 3:0 | Data cache way exclusion mask. If programmable cache allocation is enabled via the CPA bit in the *MVPControl* register, a VPE can exclde an arbitrary subset of the ways of the primary data cache from allocation by the cache controller on behalf of the VPE<br><br>**Bit / Name / Meaning**<br>3 / DWX3 / If set, D-cache way 3 will not be allocated for the VPE<br>2 / DWX2 / If set, D-cache way 2 will not be allocated for the VPE<br>1 / DWX1 / If set, D-cache way 1 will not be allocated for the VPE<br>0 / DWX0 / If set, D-cache way 1 will not be allocated for the VPE | R/W | 0 |

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

The pair of *EntryLo* registers act as the interface between the TLB and the TLBR, TLBWI, and TLBWR instructions. For a TLB-based MMU, *EntryLo0* holds the entries for even pages and *EntryLo1* holds the entries for odd pages. The contents of the *EntryLo0* and *EntryLo1* registers are undefined after an address error, TLB invalid, TLB modified, or TLB refill exception. These registers are only valid when the TLB-based memory management unit is present. They are reserved if the FM-style MMU is present.

**Figure 7.11 EntryLo0, EntryLo1 Register Format**

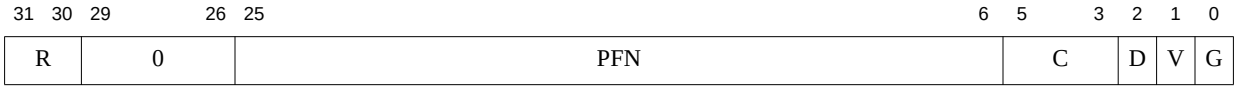



**Table 7.13 EntryLo0, EntryLo1 Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| **Name** | **Bit(s)** | | | |
| R | 31:30 | Reserved. Should be ignored on writes; returns zero on reads. | R | 0 |

**Table 7.13 EntryLo0, EntryLo1 Register Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| 0 | 29:26 | These 4 bits are normally part of the PFN, however, since the core supports only 32 bits of physical address, the PFN is only 20 bits wide; therefore, bits 29:26 of this register must be written with zeros. | R | 0 |
| PFN | 25:6 | Page Frame Number: Contributes to the definition of the high-order bits of the physical address. The PFN field corresponds to bits 31..12 of the physical address. | R/W | Undefined |
| C | 5:3 | Coherency attribute of the page. See Table 7.14. | R/W | Undefined |
| D | 2 | "Dirty" or write-enable bit: Indicates that the page has been written, and/or is writable. If this bit is a one, then stores to the page are permitted. If this bit is a zero, then 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 are valid. If this bit is a one, then accesses to the page are permitted. If this bit is a zero, then accesses to the page cause a *TLB Invalid* exception | R/W | Undefined |
| G | 0 | Global bit: On a TLB write, the logical AND of the G bits in both the *EntryLo0* and *EntryLo1* registers become the G bit in the TLB entry. If the TLB entry G bit is a one, then the ASID comparisons are ignored during TLB matches. On a read from a TLB entry, the G bits of both *EntryLo0* and *EntryLo1* reflect the state of the TLB G bit. | R/W | Undefined |

Table 7.14 lists the encoding of the *C* field of the *EntryLo0* and *EntryLo1* registers and the *K0* field of the *Config* register.

**Table 7.14 Cache Coherency Attributes**

| C[5:3] Value | Cache Coherency Attribute |
|---|---|
| 0 | Cacheable, noncoherent, write-through, no write allocate |
| 1 | Reserved |
| 2 | Uncached |
| 3 | Cacheable, noncoherent, write-back, write allocate |
| 4,5,6 | Reserved |
| 7 | Uncached Accelerated |

## 7.2.12 TCStatus Register (CP0 Register 2, Select 1)

The *TCStatus* register is instantiated per-TC as part of the system coprocessor. It defines status and control for each TC.

**Figure 7.12 TCStatus Register Format**

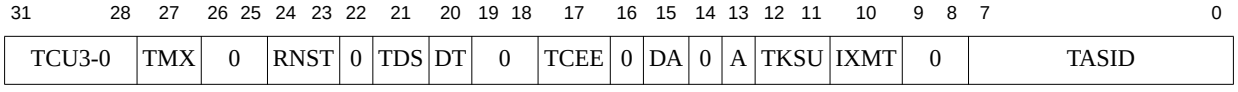


**Table 7.15 TCStatus Register Field Descriptions**

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>TCU3-0</td><td>31:28</td><td>Enable this TC to run instructions for the corresponding co-processor:<br><table><tr><th>Bit</th><th>Name</th><th>Meaning</th></tr><tr><td>31</td><td>TCU3</td><td>If set, this TC can run on co-processor 3.</td></tr><tr><td>30</td><td>TCU2</td><td>If set, this TC can run on co-processor 2.</td></tr><tr><td>29</td><td>TCU1</td><td>If set, this TC can run on co-processor 1.</td></tr><tr><td>28</td><td>TCU0</td><td>If set, this TC can run on co-processor 0.</td></tr></table>There are only two feasible co-processors: Cop1 is the floating point unit and Cop2 is available for custom use. Cop3 is not available on cores in the 34K family; TCStatus$_{TCU3}$ always reads zero. These are per-TC views of the Status$_{CU3-0}$ per-VPE fields.</td><td>R/W</td><td>0</td></tr>
<tr><td>TMX</td><td>27</td><td>Enable bit for MIPS DSP ASE or MDMX instructions for this TC. This is per-TC view of the Status$_{MX}$ per-VPE field.</td><td>R/W</td><td>0</td></tr>
<tr><td>RNST</td><td>24:23</td><td>Per-TC status bits:<br><table><tr><th>Encoding</th><th>Definition</th></tr><tr><td>0</td><td>Not blocked.</td></tr><tr><td>1</td><td>Asleep after a WAIT.</td></tr><tr><td>2</td><td>Blocked on YIELD—waiting for one or more of the yield qualifier signals to activate.</td></tr><tr><td>3</td><td>Waiting for gating storage load/store.</td></tr></table></td><td>R</td><td>0</td></tr>
<tr><td>TDS</td><td>21</td><td>Indicates the thread is stopped in a branch delay slot. The TCRestart register will point to the branch. See Section 7.2.14 “TCRestart Register (CP0 Register 2, Select 3)”.</td><td>R</td><td>Undefined</td></tr>
<tr><td>DT</td><td>20</td><td>Set when any instructions complete for this TC (exceptions and errors are not counted). It is also set if the TC is successfully started after a fork.</td><td>R/W</td><td>0</td></tr>
<tr><td>TCEE</td><td>17</td><td>Enable bit for CorExtend instruction block. This is per-TC view of the Status$_{CEE}$ per-VPE field.</td><td>R/W</td><td>0</td></tr>
<tr><td>DA</td><td>15</td><td>Dynamically Allocatable: If set, this TC can be allocated after a <code>FORK</code>.</td><td>R/W</td><td>0</td></tr>
<tr><td>A</td><td>13</td><td>Activated: If set, run instructions for this TC. This bit is set by <code>FORK</code> and cleared by <code>YIELD $0</code>.</td><td>R/W</td><td>1 for TC0,<br>0 for all others</td></tr>
<tr><td>TKSU</td><td>12:11</td><td>These bits determine the privilege state of the CPU. The encoding of this field is:<br><table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>00</td><td>Base mode is Kernel Mode</td></tr><tr><td>01</td><td>Base mode is Supervisor Mode</td></tr><tr><td>10</td><td>Base mode is User Mode</td></tr><tr><td>11</td><td>Reserved</td></tr></table>This is per-TC view of the Status$_{KSU}$ per-VPE field. Reset condition is copied from the forking thread.</td><td>R/W</td><td>0</td></tr>
</table>

**Table 7.15 TCStatus Register Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| IXMT | 10 | If set, prevent this TC from handling interrupts. | R/W | 0 |
| TASID | 7:0 | Address space identifier field. This is the per-TC view of the *EntryHi*$_{ASID}$ per-VPE field. Reset condition is copied from the forking thread. | R/W | Undefined |
| 0 | 26:25, 22, 19:18, 16, 14, 9:8 | Must be written as zeros; returns zeros on reads. | 0 | 0 |

## 7.2.13 TCBind Register (CP0 Register 2, Select 2)

The *TCBind* register is instantiated per-TC as part of the system co-processor. It defines the VPE affiliation and identification number of this TC.

**Figure 7.13 TCBind Register Format**

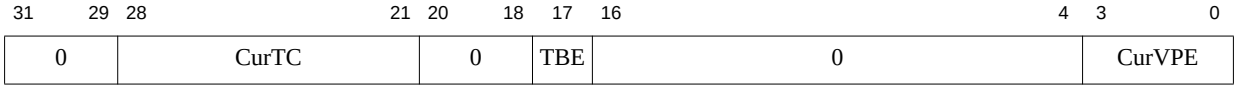



**Table 7.16 TCBind Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| 0 | 31:29, 20:18, 16:4 | Must be written as zeros; returns zeros on reads. | 0 | 0 |
| CurTC | 28:21 | Returns the ID number of this TC. | R | Preset |
| TBE | 17 | Thread Bus Error: A load instruction from this TC caused an error. | R/W | 0 |
| CurVPE | 3:0 | The ID number of the VPE affiliation of this TC. Externally set on reset based on *SI_Vpe0MaxTC*. In a two VPE system, all TCs between 0 and *SI_Vpe0MaxTC* inclusive are bound to VPE0 on reset and remaining ones are bound to VPE1. Writable when *MVPControl*$_{VPC}$ is set | R | External |

## 7.2.14 TCRestart Register (CP0 Register 2, Select 3)

The *TCRestart* register is instantiated per-TC, with the same width as the processor GPRs. When *TCStatus*$_{TDS}$ is set (the thread is stopped in a branch delay slot), this register contains the address of instruction the TC will run when it is restarted.

## 7.2.15 TCHalt Register (CP0 Register 2, Select 4)

The *TCHalt* register is instantiated per-TC as part of the system coprocessor. It is set to 1 to stop the TC for inspection or modification.

## 7.2.16 TCContext Register (CP0 Register 2, Select 5)

The *TCContext* register is instantiated per-TC. It is a 32-bit read-write scratch register for OS use.

## 7.2.17 TCSchedule Register (CP0 Register 2, Select 6)

The *Scheduler Hint* is a per-TC value whose interpretation is scheduler implementation-dependent. For example, it could encode a description of the requested issue bandwidth for the associated thread, as in the *VPESchedule* register, or it could encode a priority level.

A *TCSchedule* register value of zero is the default, and should result in a well-behaved default scheduling of the associated thread.

## 7.2.18 TCScheFBack Register (CP0 Register 2, Select 7)

The *Scheduler Feedback* is a per-TC feedback value from scheduler hardware to software. The interpretation is scheduler implementation-dependent. For example, it might encode the number of instructions retired in the instruction stream corresponding to the TC since the last time the value was cleared by software.

## 7.2.19 Context Register (CP0 Register 4, Select 0)

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

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

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

**Figure 7.14 Context Register Format**

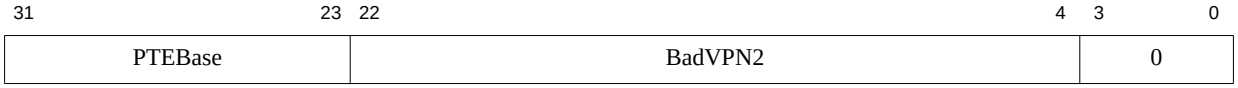



**Table 7.17 Context Register Field Descriptions**

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

## 7.2.20 PageMask Register (CP0 Register 5, Select 0)

The *PageMask* register is a read/write register used for reading from and writing to the TLB. It holds a comparison mask that sets the variable page size for each TLB entry, as shown in Table 7.19.

This register is only valid with the TLB. It is reserved if the FM is implemented.

**Figure 7.15 PageMask Register Format**

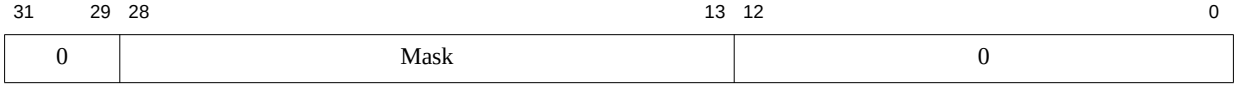



**Table 7.18 PageMask Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| 0 | 31:29, 12:0 | Ignored on write; returns zero on read. | R | 0 |
| Mask | 28:13 | 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 |

**Table 7.19 Values for the Mask Field of the PageMask Register**

| Page Size | Bit | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 |
| 4 KBytes | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 16 KBytes | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 64 KBytes | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 256 KBytes | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 MByte | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 4 MByte | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 16 MByte | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 64 MByte | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 256 MByte | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Software may determine which page sizes are supported by writing all ones to the *PageMask* register, then reading the value back. If a pair of bits reads back as ones, the processor implements that page size. The operation of the processor is **UNDEFINED** if software loads the *Mask* field with a value other than one of those listed in Table 7.19, even if the hardware returns a different value on read. Hardware may depend on this requirement in implementing hardware structures.

## 7.2.21 Wired Register (CP0 Register 6, Select 0)

The *Wired* register is a read/write register that specifies the boundary between the wired and random entries in the TLB as shown in Figure 7.16. The width of the *Wired* field is calculated in the same manner as that described for the

*Index* register above. Wired entries are fixed, non-replaceable entries that are not overwritten by a TLBWR instruction. Wired entries can be overwritten by a TLBWI instruction.

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

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

This register is only valid with a TLB. It is reserved if the FM is implemented.

**Figure 7.16 Wired and Random Entries in the TLB**

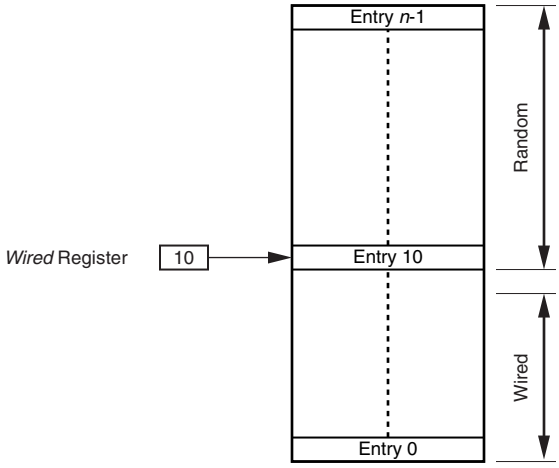



**Figure 7.17 Wired Register Format**


**Table 7.20 Wired Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| 0 | 31:6 | Must be written as zero; returns zero on reads. | 0 | 0 |
| Wired | 5:0 | TLB wired boundary.<br>For 16 and 32 entry TLBs, behavior is undefined if value is set to a value larger than last TLB entry. | R/W | 0 |

## 7.2.22 SRSConf0 (CP0 Register 6, Select 1)

The *SRSConf0* register is instantiated per-VPE. It indicates the binding of TCs or other GPR resources to Shadow Register Sets 1 through 3.

When *SRSConf0* is written, $SRSCtl_{HSS}$ is automatically updated by hardware to indicate the the highest numbered valid SRS. Software should ensure that the new *HSS* value is not less than the current value of the $SRSCtl_{CSS}$ or $SRSCtl_{PSS}$

**Figure 7.18 SRSConf0 Register Format**

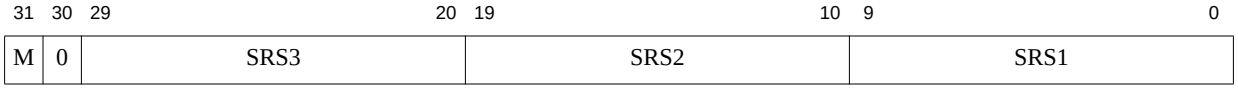



**Table 7.21 HWREna Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| M | 31 | Continuation indication. Since there is no *SRSConf1* in the 34K core, it will read zero. | R/W | 0 |
| 0 | 30 | Must be written with zero; returns zero on read | 0 | 0 |
| SRS3-1 | 29:20, 19:10, 9:0 | Indicates the GPR set to be used for corresponding shadow set number (1-3). Shadow set 0 refers to the register set normally associated with the current TC.<br>Note if a particular SRS is instantiated, all other lower order SRSs must also be instantiated.<br>If set to 0x3ff indicates this SRS is not supported.<br>If set to 0x3fe indicates this SRS is not assigned (invalid). | R/W | 0x3fe or 0x3ff |

## 7.2.23 SRSConf1-4 (CP0 Register 6, Select 2-5)

Not implemented on the 34K core.

## 7.2.24 HWREna Register (CP0 Register 7, Select 0)

The *HWREna* register contains a bit mask that determines which hardware registers are accessible via the RDHWR instruction.

**Figure 7.19 HWREna Register Format**

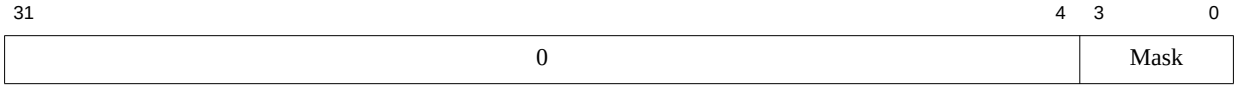



**Table 7.22 HWREna Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| 0 | 31..4 | Must be written with zero; returns zero on read | 0 | 0 |
| Mask | 3..0 | Each bit in this field enables access by the RDHWR instruction to a particular hardware register (which may not be an actual register). If bit ‘n’ in this field is a 1, access is enabled to hardware register ‘n’.<br>If bit ‘n’ of this field is a 0, access is disabled.<br>See the RDHWR instruction for a list of valid hardware registers. | R/W | 0 |

Privileged software may determine which of the hardware registers are accessible by the RDHWR instruction. In doing so, a register may be virtualized at the cost of handling a Reserved Instruction Exception, interpreting the instruction, and returning the virtualized value. For example, if it is not desirable to provide direct access to the *Count* register, access to that register may be individually disabled and the return value can be virtualized by the operating system.

## 7.2.25 BadVAddr Register (CP0 Register 8, Select 0)

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

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

The *BadVAddr* register does not capture address information for cache or bus errors, since they are not addressing errors.

**Figure 7.20 BadVAddr Register Format**

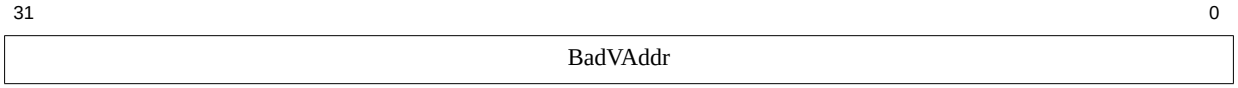



**Table 7.23 BadVAddr Register Field Description**

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

## 7.2.26 Count Register (CP0 Register 9, Select 0)

The *Count* register acts as a timer, incrementing at a constant rate, whether or not an instruction is executed, retired, or any forward progress is made through the pipeline. If enabled, the counter increments every other clock. Setting the DC bit in the *Cause* register to 0 enables counting.

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

By writing the CountDM bit in the *Debug* register, it is possible to control whether the *Count* register continues incrementing while the processor is in debug mode.

**Figure 7.21 Count Register Format**

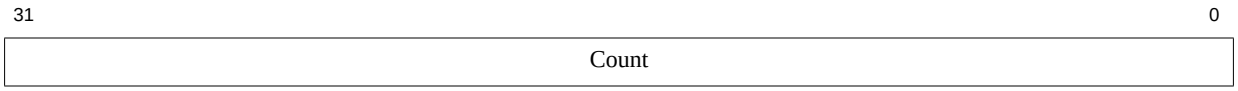



**Table 7.24 Count Register Field Description**

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

## 7.2.27 EntryHi Register (CP0 Register 10, Select 0)

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

A TLB exception (TLB Refill, TLB Invalid, or TLB Modified) causes bits $VA_{31..13}$ of the virtual address to be written into the *VPN2* field of the *EntryHi* register. A TLBR instruction writes the *EntryHi* register with the corresponding fields from the selected TLB entry. The *ASID* field is written by software with the current address space identifier value and is used during the TLB comparison process to determine TLB match.

Because the *ASID* field is overwritten by a TLBR instruction, software must save and restore the value of *ASID* around use of the TLBR. This is especially important in TLB Invalid and TLB Modified exceptions, and in other memory management software.

The *VPN2* field of the *EntryHi* register is not defined after an address error exception and this field may be modified by hardware during the address error exception sequence. Software writes of the *EntryHi* register (via MTC0) do not cause the implicit write of address-related fields in the *BadVAddr*, *Context* registers.

This register is only valid with the TLB. It is reserved if the FM is implemented.

**Figure 7.22 EntryHi Register Format**

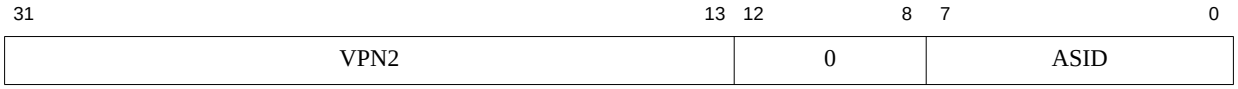



**Table 7.25 EntryHi Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| VPN2 | 31..13 | $VA_{31..13}$ of the virtual address (virtual page number / 2). This field is written by hardware on a TLB exception or on a TLB read, and is written by software before a TLB write. | R/W | Undefined |
| 0 | 12..8 | Must be written as zero; returns zero on read. | 0 | 0 |
| ASID | 7..0 | Address space identifier. This field is written by hardware on a TLB read and by software to establish the current ASID value for TLB write and against which TLB references match each entry's TLB ASID field.<br>This field is per-TC field visible in $TCStatus_{TASID}$. | R/W | Undefined |

## 7.2.28 Compare Register (CP0 Register 11, Select 0)

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

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

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

**Figure 7.23 Compare Register Format**

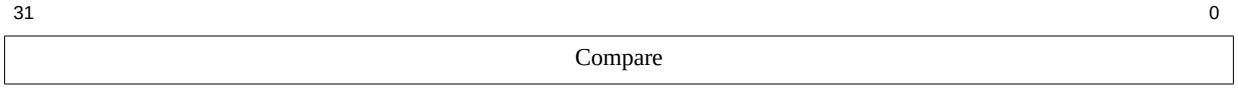



**Table 7.26 Compare Register Field Description**

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

## 7.2.29 Status Register (CP0 Register 12, Select 0)

The *Status* register 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. Refer to Section 5.2 "Modes of Operation" for a discussion of operating modes, and Section 6.3 "Interrupts" for a discussion of interrupt modes.

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

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

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

### 7.2.29.1 Operating Modes

#### *Debug Mode*

The processor is operating in Debug Mode if the *DM* bit in the CP0 *Debug* register is a one. If the processor is running in Debug Mode, it has full access to all resources that are available to Kernel Mode operation.

#### *Kernel Mode*

The processor is operating in Kernel Mode when the *DM* bit in the *Debug* register is a zero and any of the following three conditions is true:

- The *KSU* field in the CP0 *Status* register contains 2#00
- The *EXL* bit in the *Status* register is one
- The *ERL* bit in the *Status* register is one

The processor enters Kernel Mode at power-up, or as the result of an interrupt, exception, or error. The processor leaves Kernel Mode and enters User Mode or Supervisor Mode when all of the previous three conditions are false, usually as the result of an ERET instruction.

#### *Supervisor Mode*

The processor is operating in Supervisor Mode when all of the following conditions are true:

- The *DM* bit in the *Debug* register is a zero
- The *KSU* field in the *Status* register contains 2#01
- The *EXL* and *ERL* bits in the *Status* register are both zero

Supervisor mode is not supported with the Fixed Mapping MMU.

#### *User Mode*

The processor is operating in User Mode when all of the following conditions are true:

- The *DM* bit in the *Debug* register is a zero
- The *KSU* field in the *Status* register contains 2#10
- The *EXL* and *ERL* bits in the *Status* register are both zero

### 7.2.29.2 Coprocessor Accessibility

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

**Figure 7.24 Status Register Format**

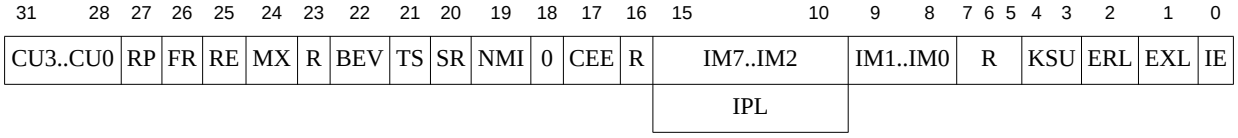



**Table 7.27 Status Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| CU3 | 31 | Reserved. This is a per-VPE view of the $TCStatus_{TCU3}$ per-TC field. | R | 0 |
| CU2 | 30 | Controls access to Coprocessor 2<br>Encoding / Meaning:<br>0 – Access not allowed<br>1 – Access allowed<br>This bit can only be written when a coprocessor 2 unit is present. This bit cannot be written and will read as 0 if coprocessor 2 unit is not present.<br>This is a per-VPE view of the $TCStatus_{TCU2}$ per-TC field. | R/W | Undefined |

**Table 7.27 Status Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| CU1 | 29 | Controls access to Coprocessor 1<br>Encoding: 0 = Access not allowed; 1 = Access allowed<br>This bit can only be written when the Floating Point Unit is present (34Kf core); in the 34Kc core, this bit cannot be written and will read as 0.<br>This is a per-VPE view of the *TCStatus*$_{TCU1}$ per-TC field. | R/W | Undefined |
| CU0 | 28 | Controls access to coprocessor 0<br>Encoding: 0 = Access not allowed; 1 = Access allowed<br>Coprocessor 0 is always usable when the processor is running in kernel mode, independent of the state of the CU0 bit.<br>This is a per-VPE view of the *TCStatus*$_{TCU0}$ per-TC field. | R/W | Undefined |
| RP | 27 | Enables reduced power mode. The state of the *RP* bit is available on the external core interface as the *SI_RP* signal. | R/W | 0 |
| FR | 26 | This bit is used to control the floating point register mode for 64-bit floating point units:<br>Encoding: 0 = Floating point registers can contain any 32-bit datatype. 64-bit datatypes are stored in even-odd pairs of registers; 1 = Floating point registers can contain any datatype<br>This bit must be ignored on write and read as zero under the following conditions<br>• No floating point unit is implemented<br>• 64-bit floating point unit is not implemented | R/W | 0 |
| RE | 25 | Used to enable reverse-endian memory references while the processor is running in user mode:<br>Encoding: 0 = User mode uses configured endianness; 1 = User mode uses reversed endianness<br>Neither Debug Mode nor Kernel Mode nor Supervisor Mode references are affected by the state of this bit. | R/W | Undefined |
| MX | 24 | Enables access to DSP ASE resources. An attempt to execute any DSP ASE instruction before this bit has been set to 1 will cause a DSP State Disabled exception.<br>This is a per-VPE view of the *TCStatus*$_{TMX}$ per-TC field. | R | |
| R | 23 | Reserved. This field is ignored on write and read as 0. | R | 0 |

**Table 7.27 Status Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| BEV | 22 | Controls the location of exception vectors:<br>Encoding 0: Normal<br>Encoding 1: Bootstrap | R/W | 1 |
| TS | 21 | TLB shutdown. Indicates that the TLB has detected a match on multiple entries. This bit is set if a TLBWI or TLBWR instruction is issued that would cause a TLB shutdown condition if allowed to complete. A machine check exception is also issued. This bit is reserved if the TLB is not implemented.<br>Software can only write a 0 to this bit to clear it and cannot force a 0-1 transition | R/W0 | 0 |
| SR | 20 | Indicates that the entry through the reset exception vector was due to a Soft Reset. Soft Reset is not supported on this processor and this bit is not writeable and will always read as 0 | R | 0 |
| NMI | 19 | Indicates that the entry through the reset exception vector was due to an NMI:<br>Encoding 0: Not NMI (Reset)<br>Encoding 1: NMI<br>Software can only write a 0 to this bit to clear it and cannot force a 0-1 transition. | R/W0 | 1 for NMI; 0 otherwise |
| 0 | 18 | Must be written as zero; returns zero on read. | 0 | 0 |
| CEE | 17 | CorExtend Enable: This bit is sent to the CorExtend block to be used to enable the CorExtend block. The usage of this signal by a CorExtend block is implementation dependent.<br>This bit is reserved if CorExtend is not present.<br>This is a per-VPE view of the *TCStatus*$_{TCEE}$ per-TC field. | R/W | Undefined |
| R | 16 | Reserved. Ignored on write and read as zero. | R | 0 |
| IM7..IM2 | 15..10 | Interrupt Mask: Controls the enabling of each of the hardware interrupts. Refer to Section 6.3 "Interrupts" for a complete discussion of enabled interrupts.<br>An interrupt is taken if interrupts are enabled and the corresponding bits are set in both the Interrupt Mask field of the *Status* register and the Interrupt Pending field of the *Cause* register and the *IE* bit is set in the *Status* register.<br>Encoding 0: Interrupt request disabled<br>Encoding 1: Interrupt request enabled<br>In implementations of Release 2 of the Architecture in which EIC interrupt mode is enabled (*Config3*$_{VEIC}$ = 1), these bits take on a different meaning and are interpreted as the *IPL* field, described below. | R/W | Undefined |

**Table 7.27 Status Register Field Descriptions**

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>IPL</td><td>15..10</td><td>Interrupt Priority Level: In implementations of Release 2 of the Architecture in which EIC interrupt mode is enabled (Config3$_{VEIC}$ = 1), this field is the encoded (0..63) value of the current IPL. An interrupt will be signaled only if the requested IPL is higher than this value.<br>If EIC interrupt mode is not enabled (Config3$_{VEIC}$ = 0), these bits take on a different meaning and are interpreted as the IM7..IM2 bits, described above.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>IM1..IM0</td><td>9..8</td><td>Interrupt Mask: Controls the enabling of each of the software interrupts. Refer to Section 6.3 “Interrupts” for a complete discussion of enabled interrupts.<br><table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Interrupt request disabled</td></tr><tr><td>1</td><td>Interrupt request enabled</td></tr></table>In implementations of Release 2 of the Architecture in which EIC interrupt mode is enabled (Config3$_{VEIC}$ = 1), these bits are writable, but have no effect on the interrupt system.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>R</td><td>7..5</td><td>Reserved. This field is ignored on write and read as 0.</td><td>R</td><td>0</td></tr>
<tr><td>KSU</td><td>4..3</td><td>This field denotes the base operating mode of the processor. See Section 5.2 “Modes of Operation” for a full discussion of operating modes. The encoding of this field is:<br><table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>00</td><td>Base mode is Kernel Mode</td></tr><tr><td>01</td><td>Base mode is Supervisor Mode</td></tr><tr><td>10</td><td>Base mode is User Mode</td></tr><tr><td>11</td><td>Reserved</td></tr></table>Note that the processor can also be in kernel mode if ERL or EXL is set, regardless of the state of the KSU field.<br>This is a per-VPE view of the TCStatus$_{TKSU}$ per-TC field.</td><td>R/W</td><td>Undefined</td></tr>
</table>

**Table 7.27 Status Register Field Descriptions**

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>ERL</td><td>2</td><td>Error Level; Set by the processor when a Reset, Soft Reset, NMI or Cache Error exception are taken.<br><table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Normal level</td></tr><tr><td>1</td><td>Error level</td></tr></table>When ERL is set:<br>• The processor is running in kernel mode<br>• Interrupts are disabled<br>• The ERET instruction will use the return address held in ErrorEPC instead of EPC<br>• The lower $2^{29}$ bytes of kuseg are treated as an unmapped and uncached region. See Chapter 5, “Memory Management of the 34K™ Core”. This allows main memory to be accessed in the presence of cache errors. The operation of the processor is UNDEFINED if the ERL bit is set while the processor is executing instructions from kuseg.</td><td>R/W</td><td>1</td></tr>
<tr><td>EXL</td><td>1</td><td>Exception Level; Set by the processor when any exception other than Reset, Soft Reset, or NMI exceptions is taken.<br><table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Normal level</td></tr><tr><td>1</td><td>Exception level</td></tr></table>When EXL is set:<br>• The processor is running in Kernel Mode<br>• Interrupts are disabled.<br>• TLB Refill exceptions use the general exception vector instead of the TLB Refill vector.<br>• EPC, $Cause_{BD}$ and SRSCtl (implementations of Release 2 of the Architecture only) will not be updated if another exception is taken</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>IE</td><td>0</td><td>Interrupt Enable: Acts as the master enable for software and hardware interrupts:<br><table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Interrupts are disabled</td></tr><tr><td>1</td><td>Interrupts are enabled</td></tr></table>In Release 2 of the Architecture, this bit may be modified separately via the DI and EI instructions.</td><td>R/W</td><td>Undefined</td></tr>
</table>

## 7.2.30 IntCtl Register (CP0 Register 12, Select 1)

The *IntCtl* register controls the expanded interrupt capability added in Release 2 of the Architecture, including vectored interrupts and support for an external interrupt controller. This register does not exist in implementations of Release 1 of the Architecture.

**Figure 7.25 IntCtl Register Format**


**Table 7.28 IntCtl Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| IPTI | 31:29 | For Interrupt Compatibility and Vectored Interrupt modes, this field specifies the IP number to which the Timer Interrupt request is merged, and allows software to determine whether to consider $Cause_{TI}$ for a potential interrupt.<br>Encoding / IP bit / Hardware Interrupt Source: 2/2/HW0; 3/3/HW1; 4/4/HW2; 5/5/HW3; 6/6/HW4; 7/7/HW5<br>The value of this bit is set by the static input, *SI_IPTI[2:0]*. This allows external logic to communicate the specific *SI_Int* hardware interrupt pin to which the *SI_TimerInt* signal is attached.<br>The value of this field is not meaningful if External Interrupt Controller Mode is enabled. The external interrupt controller is expected to provide this information for that interrupt mode. | R | Externally Set |
| IPPCI | 28:26 | For Interrupt Compatibility and Vectored Interrupt modes, this field specifies the IP number to which the Performance Counter Interrupt request is merged, and allows software to determine whether to consider $Cause_{PCI}$ for a potential interrupt.<br>The value of this field is not meaningful if External Interrupt Controller Mode is enabled. The external interrupt controller is expected to provide this information for that interrupt mode. | R | Externally Set |
| VS | 9:5 | Vector Spacing. If vectored interrupts are implemented (as denoted by $Config3_{VInt}$ or $Config3_{VEIC}$), this field specifies the spacing between vectored interrupts.<br>Encoding / Spacing Between Vectors (hex) / Spacing Between Vectors (decimal): 16#00/16#000/0; 16#01/16#020/32; 16#02/16#040/64; 16#04/16#080/128; 16#08/16#100/256; 16#10/16#200/512<br>All other values are reserved. The operation of the processor is **UNDEFINED** if a reserved value is written to this field. | R/W | 0 |
| 0 | 25:10, 4:0 | Must be written as zero; returns zero on read. | 0 | 0 |

## 7.2.31 SRSCtl Register (CP0 Register 12, Select 2)

The *SRSCtl* register controls the operation of GPR shadow sets in the processor.

**Figure 7.26 SRSCtl Register Format**

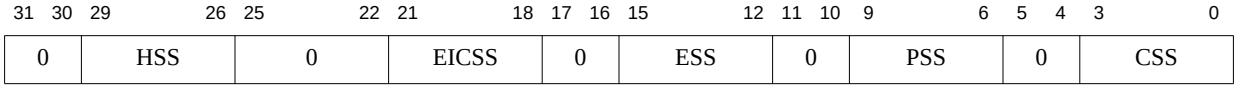



**Table 7.29 SRSCtl Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| HSS | 29:26 | Highest Shadow Set. This field contains the highest shadow set number that is implemented by this processor. A value of zero in this field indicates that only the normal GPRs are implemented. Possible values of this field for the 34K processor are:<br>Encoding / Meaning:<br>0 — One shadow set (normal GPR set) is present.<br>1 — Two shadow sets are present.<br>2 — Three shadow sets are present.<br>3 — Four shadow sets are present.<br>3-15 — Reserved<br>The value in this field also represents the highest value that can be written to the *ESS*, *EICSS*, *PSS*, and *CSS* fields of this register, or to any of the fields of the *SRSMap* register. The operation of the processor is **UNDEFINED** if a value larger than the one in this field is written to any of these other fields. This field is automatically updated when *SRSConf0* is written. | R | Preset |
| EICSS | 21:18 | EIC interrupt mode shadow set. If $Config3_{VEIC}$ is 1 (EIC interrupt mode is enabled), this field is loaded from the external interrupt controller for each interrupt request and is used in place of the *SRSMap* register to select the current shadow set for the interrupt. See Section 6.3.1.3 "External Interrupt Controller Mode" for a discussion of EIC interrupt mode. If $Config3_{VEIC}$ is 0, this field returns zero on read. | R | Undefined |
| ESS | 15:12 | Exception Shadow Set. This field specifies the shadow set to use on entry to Kernel Mode caused by any exception other than a vectored interrupt. The operation of the processor is **UNDEFINED** if software writes a value into this field that is greater than the value in the HSS field. | R/W | 0 |

**Table 7.29 SRSCtl Register Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| PSS | 9:6 | Previous Shadow Set. If GPR shadow registers are implemented, and with the exclusions noted in the next paragraph, this field is copied from the CSS field when an exception or interrupt occurs. An ERET instruction copies this value back into the CSS field if $Status_{BEV}$ = 0.<br>This field is not updated on any exception which sets $Status_{ERL}$ to 1 (i.e., Reset, Soft Reset, NMI, cache error), an entry into EJTAG Debug mode, or any exception or interrupt that occurs with $Status_{EXL}$ = 1, or $Status_{BEV}$ = 1. This field is not updated on an exception that occurs while $Status_{ERL}$ = 1.<br>The operation of the processor is **UNDEFINED** if software writes a value into this field that is greater than the value in the *HSS* field. | R/W | 0 |
| CSS | 3:0 | Current Shadow Set. If GPR shadow registers are implemented, this field is the number of the current GPR set. With the exclusions noted in the next paragraph, this field is updated with a new value on any interrupt or exception, and restored from the PSS field on an ERET. Table 7.30 describes the various sources from which the CSS field is updated on an exception or interrupt.<br>This field is not updated on any exception which sets $Status_{ERL}$ to 1 (i.e., Reset, Soft Reset, NMI, cache error), an entry into EJTAG Debug mode, or any exception or interrupt that occurs with $Status_{EXL}$ = 1, or $Status_{BEV}$ = 1. Neither is it updated on an ERET with $Status_{ERL}$ = 1 or $Status_{BEV}$ = 1. This field is not updated on an exception that occurs while $Status_{ERL}$ = 1.<br>The value of CSS can be changed directly by software only by writing the PSS field and executing an ERET instruction. | R | 0 |
| 0 | 31:30, 25:22, 17:16, 11:10, 5:4 | Must be written as zeros; returns zero on read. | 0 | 0 |

**Table 7.30 Sources for new $SRSCtl_{CSS}$ on an Exception or Interrupt**

| Exception Type | Condition | $SRSCtl_{CSS}$ Source | Comment |
|---|---|---|---|
| Exception | All | $SRSCtl_{ESS}$ | |
| Non-Vectored Interrupt | $Cause_{IV}$ = 0 | $SRSCtl_{ESS}$ | Treat as exception |
| Vectored Interrupt | $Cause_{IV}$ = 1 and $Config3_{VEIC}$ = 0 and $Config3_{VInt}$ = 1 | $SRSMap_{VECTNUM}$ | Source is internal map register. (for VECTNUM see Table 6.4) |
| Vectored EIC Interrupt | $Cause_{IV}$ = 1 and $Config3_{VEIC}$ = 1 | $SRSCtl_{EICSS}$ | Source is external interrupt controller. |

## 7.2.32 SRSMap Register (CP0 Register 12, Select 3)

The *SRSMap* register contains 8 4-bit fields that provide the mapping from an vector number to the shadow set number to use when servicing such an interrupt. The values from this register are not used for a non-interrupt exception, or a non-vectored interrupt ($Cause_{IV}$ = 0 or $IntCtl_{VS}$ = 0). In such cases, the shadow set number comes from $SRSCtl_{ESS}$.

If $SRSCtl_{HSS}$ is zero, the results of a software read or write of this register are **UNPREDICTABLE**.

The operation of the processor is **UNDEFINED** if a value is written to any field in this register that is greater than the value of $SRSCtl_{HSS}$.

The *SRSMap* register contains the shadow register set numbers for vector numbers 7..0. The same shadow set number can be established for multiple interrupt vectors, creating a many-to-one mapping from a vector to a single shadow register set number.

**Figure 7.27 SRSMap Register Format**

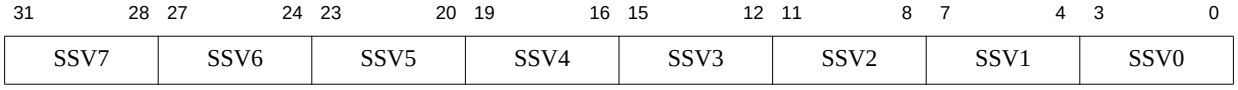



**Table 7.31 SRSMap Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| SSV7 | 31..28 | Shadow register set number for Vector Number 7 | R/W | 0 |
| SSV6 | 27..24 | Shadow register set number for Vector Number 6 | R/W | 0 |
| SSV5 | 23..20 | Shadow register set number for Vector Number 5 | R/W | 0 |
| SSV4 | 19..16 | Shadow register set number for Vector Number 4 | R/W | 0 |
| SSV3 | 15..12 | Shadow register set number for Vector Number 3 | R/W | 0 |
| SSV2 | 11..8 | Shadow register set number for Vector Number 2 | R/W | 0 |
| SSV1 | 7..4 | Shadow register set number for Vector Number 1 | R/W | 0 |
| SSV0 | 3..0 | Shadow register set number for Vector Number 0 | R/W | 0 |

## 7.2.33 Cause Register (CP0 Register 13, Select 0)

The *Cause* register primarily describes the cause of the most recent exception. In addition, fields also control software interrupt requests and the vector through which interrupts are dispatched. With the exception of the $IP_{1..0}$, *DC*, *IV*, and *WP* fields, all fields in the *Cause* register are read-only. Release 2 of the Architecture added optional support for an External Interrupt Controller (EIC) interrupt mode, in which $IP_{7..2}$ are interpreted as the Requested Interrupt Priority Level (RIPL).

**Figure 7.28 Cause Register Format**

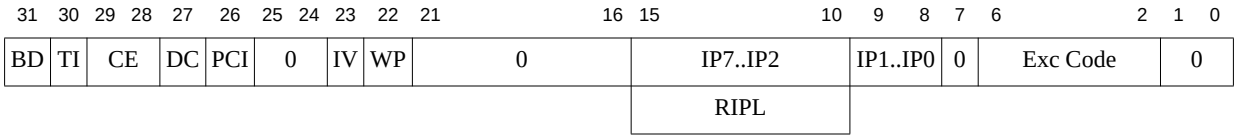


**Table 7.32 Cause Register Field Descriptions**

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>BD</td><td>31</td><td>Indicates whether the last exception taken occurred in a branch delay slot:<br><table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Not in delay slot</td></tr><tr><td>1</td><td>In delay slot</td></tr></table>The processor updates BD only if Status$_{EXL}$ was zero when the exception occurred.</td><td>R</td><td>Undefined</td></tr>
<tr><td>TI</td><td>30</td><td>Timer Interrupt. This bit denotes whether a timer interrupt is pending (analogous to the IP bits for other interrupt types):<br><table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No timer interrupt is pending</td></tr><tr><td>1</td><td>Timer interrupt is pending</td></tr></table>The state of the TI bit is available on the external core interface as the SI_TimerInt signal.</td><td>R</td><td>Undefined</td></tr>
<tr><td>CE</td><td>29:28</td><td>Coprocessor unit number referenced when a Coprocessor Unusable exception is taken. This field is loaded by hardware on every exception, but is UNPREDICTABLE for all exceptions except for Coprocessor Unusable.</td><td>R</td><td>Undefined</td></tr>
<tr><td>DC</td><td>27</td><td>Disable Count register. In some power-sensitive applications, the Count register is not used and is the source of meaningful power dissipation. This bit allows the Count register to be stopped in such situations.<br><table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Enable counting of Count register</td></tr><tr><td>1</td><td>Disable counting of Count register</td></tr></table></td><td>R/W</td><td>0</td></tr>
<tr><td>PCI</td><td>26</td><td>Performance Counter Interrupt: This bit denotes whether a performance counter interrupt is pending (analogous to the IP bits for other interrupt types):<br><table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No performance counter interrupt is pending</td></tr><tr><td>1</td><td>Performance counter interrupt is pending</td></tr></table>The state of the PCI bit is available on the external core interface as the SI_PCInt signal.</td><td>R</td><td>Undefined</td></tr>
<tr><td>IV</td><td>23</td><td>Indicates whether an interrupt exception uses the general exception vector or a special interrupt vector:<br><table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Use the general exception vector (16#180)</td></tr><tr><td>1</td><td>Use the special interrupt vector (16#200)</td></tr></table>If the Cause$_{IV}$ is 1 and Status$_{BEV}$ is 0, the special interrupt vector represents the base of the vectored interrupt table.</td><td>R/W</td><td>Undefined</td></tr>
</table>

**Table 7.32 Cause Register Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| WP | 22 | Indicates that a watch exception was deferred because $Status_{EXL}$ or $Status_{ERL}$ were a one at the time the watch exception was detected. This bit both indicates that the watch exception was deferred, and causes the exception to be initiated once $Status_{EXL}$ and $Status_{ERL}$ are both zero. As such, software must clear this bit as part of the watch exception handler to prevent a watch exception loop. Software should not write a 1 to this bit when its value is a 0, thereby causing a 0-to-1 transition. If such a transition is caused by software, it is **UNPREDICTABLE** whether hardware ignores the write, accepts the write with no side effects, or accepts the write and initiates a watch exception once $Status_{EXL}$ and $Status_{ERL}$ are both zero. | R/W | Undefined |
| IP7..IP2 | 15:10 | Indicates an interrupt is pending:<br>Bit 15: Name IP7: Hardware interrupt 5<br>Bit 14: Name IP6: Hardware interrupt 4<br>Bit 13: Name IP5: Hardware interrupt 3<br>Bit 12: Name IP4: Hardware interrupt 2<br>Bit 11: Name IP3: Hardware interrupt 1<br>Bit 10: Name IP2: Hardware interrupt 0<br>If EIC interrupt mode is not enabled ($Config3_{VEIC}$ = 0), timer interrupts are combined in a system-dependent way with any hardware interrupt. If EIC interrupt mode is enabled ($Config3_{VEIC}$ = 1), these bits take on a different meaning and are interpreted as the *RIPL* field, described below.<br>See Section 6.3 "Interrupts" for a general description of interrupt processing. | R | Undefined |
| RIPL | 15:10 | Requested Interrupt Priority Level: If EIC interrupt mode is enabled ($Config3_{VEIC}$ = 1), this field is the encoded (0..63) value of the requested interrupt. A value of zero indicates that no interrupt is requested.<br>If EIC interrupt mode is not enabled ($Config3_{VEIC}$ = 0), these bits take on a different meaning and are interpreted as the *IP7..IP2* bits, described above. | R | Undefined |
| IP1..IP0 | 9:8 | Controls the request for software interrupts:<br>Bit 9: Name IP1: Request software interrupt 1<br>Bit 8: Name IP0: Request software interrupt 0<br>These bits are exported to an external interrupt controller for prioritization in EIC interrupt mode with other interrupt sources. The state of these bits is available on the external core interface as the *SI_SWInt[1:0]* bus. | R/W | Undefined |
| ExcCode | 6:2 | Exception code - see Table 7.33 | R | Undefined |

**Table 7.32 Cause Register Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| 0 | 25:24, 21:16, 7, 1:0 | Must be written as zero; returns zero on read. | 0 | 0 |

**Table 7.33 Cause Register ExcCode Field**

| Exception Code Value | | Mnemonic | Description |
|---|---|---|---|
| Decimal | Hexadecimal | | |
| 0 | 16#00 | Int | Interrupt |
| 1 | 16#01 | Mod | TLB modification exception |
| 2 | 16#02 | TLBL | TLB exception (load or instruction fetch) |
| 3 | 16#03 | TLBS | TLB exception (store) |
| 4 | 16#04 | AdEL | Address error exception (load or instruction fetch) |
| 5 | 16#05 | AdES | Address error exception (store) |
| 6 | 16#06 | IBE | Bus error exception (instruction fetch) |
| 7 | 16#07 | DBE | Bus error exception (data reference: load or store) |
| 8 | 16#08 | Sys | Syscall exception |
| 9 | 16#09 | Bp | Breakpoint exception. If an SDBBP instruction is executed while the processor is running in EJTAG Debug Mode, this value is written to the $Debug_{DExcCode}$ field to denote an SDBBP in Debug Mode. |
| 10 | 16#0a | RI | Reserved instruction exception |
| 11 | 16#0b | CpU | Coprocessor Unusable exception |
| 12 | 16#0c | Ov | Arithmetic Overflow exception |
| 13 | 16#0d | Tr | Trap exception |
| 14 | 16#0e | - | Reserved |
| 15 | 16#0f | FPE | Floating point exception |
| 16 | 16#10 | IS1 | Coprocessor 2 implementation specific exception |
| 17 | 16#11 | CEU | CorExtend Unusable |
| 18 | 16#12 | C2E | Precise Coprocessor 2 exception |
| 19-22 | 16#13-16#16 | - | Reserved |
| 23 | 16#17 | WATCH | Reference to *WatchHi*/*WatchLo* address |
| 24 | 16#18 | MCheck | Machine check |
| 25 | 16#19 | Thread | Thread exception. $VPEControl_{EXCPT}$ specifies the type of the thread exception. |
| 26 | 16#1a | DSPDis | DSP ASE State Disabled exception |
| 27-29 | 16#1b-16#1d | - | Reserved |

**Table 7.33 Cause Register ExcCode Field (Continued)**

| Exception Code Value | | Mnemonic | Description |
|---|---|---|---|
| Decimal | Hexadecimal | | |
| 30 | 16#1e | CacheErr | Cache error. In normal mode, a cache error exception has a dedicated vector and the Cause register is not updated. If a cache error occurs while in Debug Mode, this code is written to the $Debug_{DExcCode}$ field to indicate that re-entry to Debug Mode was caused by a cache error. |
| 31 | 16#1f | - | Reserved |

## 7.2.34 Exception Program Counter (CP0 Register 14, Select 0)

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

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

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

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

In processors that implement the MIPS16 ASE, a read of the *EPC* register (via MFC0) returns the following value in the destination GPR:

$$GPR[rt] \leftarrow ExceptionPC_{31..1} \,||\, ISAMode_0$$

That is, the upper 31 bits of the exception PC are combined with the lower bit of the ISAMode field and written to the GPR.

Similarly, a write to the *EPC* register (via MTC0) takes the value from the GPR and distributes that value to the exception PC and the ISAMode field, as follows

$$ExceptionPC \leftarrow GPR[rt]_{31..1} \,||\, 0$$
$$ISAMode \leftarrow 2\#0 \,||\, GPR[rt]_0$$

That is, the upper 31 bits of the GPR are written to the upper 31 bits of the exception PC, and the lower bit of the exception PC is cleared. The upper bit of the ISAMode field is cleared and the lower bit is loaded from the lower bit of the GPR.

**Figure 7.29 EPC Register Format**

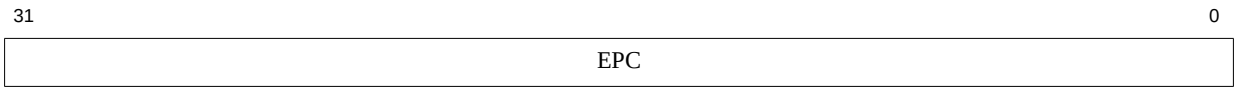


**Table 7.34 EPC Register Field Description**

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

## 7.2.35 Processor Identification (CP0 Register 15, Select 0)

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

**Figure 7.30 PRId Register Format**

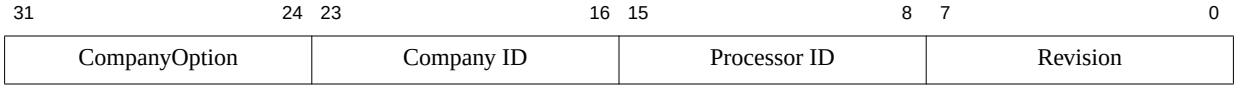



**Table 7.35 PRId Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| Company Option | 31:24 | Implementation specific values | R | Preset |
| Company ID | 23:16 | Identifies the company that designed or manufactured the processor. In the 34K this field contains a value of 1 to indicate 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 Technologies processors. | R | 0x95 |
| Revision | 7:0 | Specifies the revision number of the processor. This field allows software to distinguish between one revision and another of the same processor type.<br>This field is broken up into the following three subfields: | R | Preset |

| Bit(s) | Name | Meaning |
|---|---|---|
| 7:5 | Major Revision | This number is increased on major revisions of the processor core |
| 4:2 | Minor Revision | This number is increased on each incremental revision of the processor and reset on each new major revision |
| 1:0 | Patch Level | If a patch is made to modify an older revision of the processor, this field will be incremented |

## 7.2.36 EBase Register (CP0 Register 15, Select 1)

The *EBase* register is a read/write register containing the base address of the exception vectors used when $Status_{BEV}$ equals 0, and a read-only CPU number value that may be used by software to distinguish different processors in a multi-processor system.

The *EBase* register provides the ability for software to identify the specific processor within a multi-processor system, and allows the exception vectors for each processor to be different, especially in systems composed of heteroge-

neous processors. Bits 31:12 of the *EBase* register are concatenated with zeros to form the base of the exception vectors when $Status_{BEV}$ is 0. The exception vector base address comes from the fixed defaults (see Section 6.5 "Exception Vector Locations") when $Status_{BEV}$ is 1, or for any EJTAG Debug exception. The reset state of bits 31:12 of the *EBase* register initialize the exception base register to `16#8000.0000`, providing backward compatibility with Release 1 implementations.

Bits 31:30 of the *EBase* Register are fixed with the value `2#10` to force the exception base address to be in the kseg0 or kseg1 unmapped virtual address segments. Bit 29 of exception base address will be forced to 1 on Cache Error exceptions so the exception handler will be executed from the uncached kseg1 segment.

If the value of the exception base register is to be changed, this must be done with $Status_{BEV}$ equal 1. The operation of the processor is **UNDEFINED** if the Exception Base field is written with a different value when $Status_{BEV}$ is 0.

Combining bits 31:12 with the Exception Base field allows the base address of the exception vectors to be placed at any 4KBbyte page boundary.

**Figure 7.31 EBase Register Format**

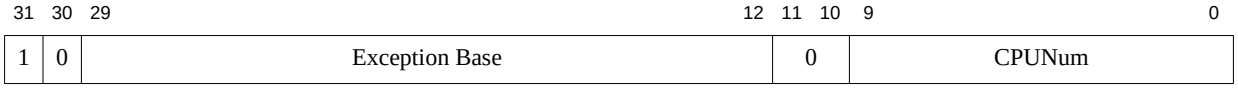



**Table 7.36 EBase Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| 1 | 31 | This bit is ignored on write and returns one on read. | R | 1 |
| Exception Base | 29:12 | In conjunction with bits 31..30, this field specifies the base address of the exception vectors when $Status_{BEV}$ is zero. | R/W | 0 |
| CPUNum | 9:0 | This field specifies the number of the CPU in a multi-processor system and can be used by software to distinguish a particular processor from the others. The value in this field is set by the *SI_CPUNum[9:0]* static input pins to the core. In a single processor system, this value should be set to zero.<br>In a two VPE system, the lowest bit of *SI_CPUNum* is ignored and *CPUNum[0]* is reset to 0 for VPE0 and to 1 for VPE1. | R | Externally Set |
| 0 | 30, 11:10 | Must be written as zero; returns zero on read. | 0 | 0 |

## 7.2.37 Config Register (CP0 Register 16, Select 0)

The *Config* register specifies various configuration and capabilities information. Most of the fields in the *Config* register are initialized by hardware during the Reset exception process, or are constant. The *K0*, *KU*, and *K23* fields must be initialized by software in the Reset exception handler, if the reset value is not desired.

**Figure 7.32 Config Register Format — Select 0**

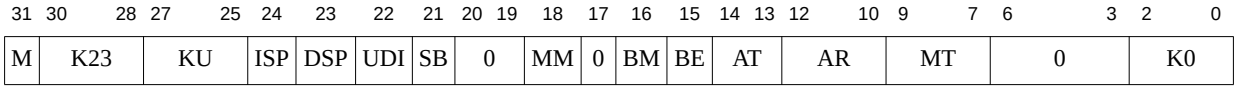


**Table 7.37 Config Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| M | 31 | This bit is hardwired to '1' to indicate the presence of the Config1 register. | R | 1 |
| K23 | 30:28 | This field controls the cacheability of the kseg2 and kseg3 address segments in FM implementations.<br>Refer to Table 7.38 for the field encoding. | FM: R/W<br>TLB: R | FM: 010<br>TLB: 000 |
| KU | 27:25 | This field controls the cacheability of the kuseg and useg address segments in FM implementations.<br>Refer to Table 7.38 for the field encoding. | FM: R/W<br>TLB: R | FM: 010<br>TLB: 000 |
| ISP | 24 | I-side ScratchPad RAM present | R | Preset |
| DSP | 23 | D-side ScratchPad RAM present | R | Preset |
| UDI | 22 | This bit indicates that CorExtend User Defined Instructions have been implemented.<br>**Encoding** / **Description**<br>0 — No User Defined Instructions are implemented<br>1 — User Defined Instructions are implemented<br>This bit is automatically updated by hardware when *VPEconf0*$_{NCX}$ is written. | R | Preset |
| SB | 21 | Indicates whether SimpleBE bus mode is enabled. Set via *SI_SimpleBE* input pin.<br>**Encoding** / **Description**<br>0 — No reserved byte enables on OCP interface<br>1 — Only simple byte enables allowed on OCP interface | R | Externally Set |
| MM | 18 | This bit indicates whether write-through merging is enabled in the 32 byte collapsing write buffer.<br>**Encoding** / **Description**<br>0 — No Merging<br>1 — Merging allowed<br>This bit is implemented per-processor and not per-VPE like all other writable fields of this register. | R/W | 1 |
| BM | 16 | Burst order. Set via *SI_SBlock* input pin.<br>**Encoding** / **Description**<br>0 — Sequential<br>1 — SubBlock | R | Externally Set |
| BE | 15 | Indicates the endian mode in which the processor is running. Set via *SI_Endian* input pin.<br>**Encoding** / **Description**<br>0 — Little endian<br>1 — Big endian | R | Externally Set |

**Table 7.37 Config Register Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| AT | 14:13 | Architecture type implemented by the processor. This field is always 00 to indicate the MIPS32 architecture. | R | 00 |
| AR | 12:10 | Architecture revision level. This field is always 001 to indicate MIPS32 Release 2.<br>Encoding: 0 — Release 1; 1 — Release 2; 2:7 — Reserved | R | 001 |
| MT | 9:7 | MMU Type:<br>Encoding: 1 — Standard TLB; 3 — Fixed Mapping; 0, 2, 4:7 — Reserved | R | Preset |
| K0 | 2:0 | Kseg0 coherency algorithm. Refer to Table 7.38 for the field encoding. | R/W | 010 |
| 0 | 20:19, 17, 6:3 | Must be written as zeros; returns zeros on reads. | 0 | 0 |

**Table 7.38 Cache Coherency Attributes**

| K0(2:0) Value | Cache Coherency Attribute |
|---|---|
| 0 | Cacheable, noncoherent, write-through, no write allocate |
| 1 | Reserved |
| 2 | Uncached |
| 3 | Cacheable, noncoherent, write-back, write allocate |
| 4,5,6 | Reserved |
| 7 | Uncached Accelerated |

## 7.2.38 Config1 Register (CP0 Register 16, Select 1)

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

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

```
Associativity * Line Size * Sets Per Way
```

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

**Figure 7.33 Config1 Register Format**

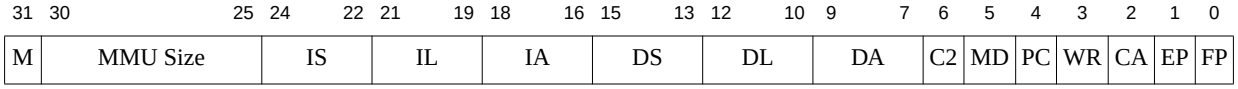



**Table 7.39 Config1 Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| M | 31 | This bit is hardwired to ‘1’ to indicate the presence of the Config2 register. | R | 1 |
| MMU Size | 30:25 | This field contains the number of entries in the TLB minus one. The field is read as 0 decimal if the TLB is not implemented | R | Preset |
| IS | 24:22 | This field contains the number of instruction cache sets per way. The corresponding total instruction cache size is shown in parantheses<br>**Encoding** / **Description**<br>0x0 — 64 (8KB )<br>0x1 — 128 (16KB)<br>0x2 — 256 (32KB)<br>0x3 — 512 (64KB)<br>0x4:0x7 — Reserved | R | Preset |
| IL | 21:19 | This field contains the instruction cache line size The cache line size is fixed at 32 bytes when the ICache is present. A value of 0 indicates no ICache.<br>**Encoding** / **Description**<br>0x0 — No ICache present<br>0x1:0x3 — Reserved<br>0x4 — 32 bytes<br>0x5:0x7 — Reserved | R | Preset |
| IA | 18:16 | This field contains the level of instruction cache associativity This field is fixed at 4-way set associative<br>**Encoding** / **Description**<br>0x0:0x2 — Reserved<br>0x3 — 4-way<br>0x4:0x7 — Reserved | R | 0x3 |
| DS | 15:13 | This field contains the number of data cache sets per way. The corresponding total data cache size is shown in parantheses<br>**Encoding** / **Description**<br>0x0 — 64 (8KB )<br>0x1 — 128 (16KB)<br>0x2 — 256 (32KB)<br>0x3 — 512 (64KB)<br>0x4:0x7 — Reserved | R | Preset |

**Table 7.39 Config1 Register Field Descriptions (Continued)**

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>DL</td><td>12:10</td><td>This field contains the data cache line size. The cache line size is fixed at 32 bytes when a Dcache is present. This field reads 0 when a Dcache is not present.<br><table><tr><th>Encoding</th><th>Description</th></tr><tr><td>0x0</td><td>No DCache present</td></tr><tr><td>0x1:0x3</td><td>Reserved</td></tr><tr><td>0x4</td><td>32 bytes</td></tr><tr><td>0x5:0x7</td><td>Reserved</td></tr></table></td><td>R</td><td>Preset</td></tr>
<tr><td>DA</td><td>9:7</td><td>This field contains the type of set associativity for the data cache The associativity is fixed at 4-way.<br><table><tr><th>Encoding</th><th>Description</th></tr><tr><td>0x0:0x2</td><td>Reserved</td></tr><tr><td>0x3</td><td>4-way</td></tr><tr><td>0x4:0x7</td><td>Reserved</td></tr></table></td><td>R</td><td>0x3</td></tr>
<tr><td>C2</td><td>6</td><td>Coprocessor 2 present.<br><table><tr><th>Encoding</th><th>Description</th></tr><tr><td>0</td><td>Coprocessor2 not present</td></tr><tr><td>1</td><td>Coprocessor2 present</td></tr></table>This bit is automatically updated by hardware when VPEConf1$_{NCP2}$ is written.</td><td>R</td><td>Preset</td></tr>
<tr><td>MD</td><td>5</td><td>MDMX implemented.</td><td>R</td><td>0</td></tr>
<tr><td>PC</td><td>4</td><td>Performance Counter registers implemented.</td><td>R</td><td>1</td></tr>
<tr><td>WR</td><td>3</td><td>Watch registers implemented.<br><table><tr><th>Encoding</th><th>Description</th></tr><tr><td>0</td><td>No Watch registers are present</td></tr><tr><td>1</td><td>One or more Watch registers are present</td></tr></table></td><td>R</td><td>1</td></tr>
<tr><td>CA</td><td>2</td><td>Code compression (MIPS16) implemented.<br><table><tr><th>Encoding</th><th>Description</th></tr><tr><td>0</td><td>No MIPS16 present</td></tr><tr><td>1</td><td>MIPS16 is implemented</td></tr></table></td><td>R</td><td>1</td></tr>
<tr><td>EP</td><td>1</td><td>EJTAG present: This bit is always set to indicate that the core implements EJTAG.</td><td>R</td><td>1</td></tr>
<tr><td>FP</td><td>0</td><td>FPU implemented.<br>This bit is automatically updated by hardware when VPEConf1$_{NCP1}$ is written.</td><td>R</td><td>Preset</td></tr>
</table>

## 7.2.39 Config2 Register (CP0 Register 16, Select 2)

The *Config2* register is an adjunct to the *Config* register and is reserved to encode additional capabilities information. *Config2* is allocated for showing the configuration of level 2/3 caches. L2 values reflect the configuration information input from the L2 module. L3 fields are reset to 0 because L3 caches are not supported by the 34K core. All fields in the *Config2* register are read-only.

**Figure 7.34 Config2 Register Format**

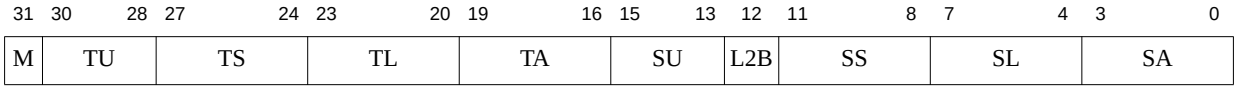



**Table 7.40 Config2 Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| M | 31 | This bit is hardwired to ‘1’ to indicate the presence of the Config3 register. | R | 1 |
| TU | 30:28 | Implementation specific tertiary cache control. Tertiary cache not supported | R | 0 |
| TS | 27:24 | Tertiary cache sets per way. Tertiary cache not supported | R | 0 |
| TL | 23:20 | Tertiary cache line size. Tertiary cache not supported | R | 0 |
| TA | 19:16 | Tertiary cache associativity. Tertiary cache not supported | R | 0 |
| SU | 15:13 | Reserved | R | 0 |
| L2B | 12 | L2 Bypass/L2_Bypassed. In systems which include an L2 cache, writing a 1 to this bit, will set the L2_Bypass output from the core. Setting the L2_Bypass output, directs the L2 cache to go into bypass mode. L2 responds by asserting its L2_Bypassed output pin. The value of L2_Bypassed is returned when L2B is read. | R/W | 0 |
| SS | 11:8 | Secondary cache sets per way<br><br>Encoding / Sets Per Way:<br>0 – 64<br>1 – 128<br>2 – 256<br>3 – 512<br>4 – 1024<br>5 – 2048<br>6 – 4096<br>7 – 8192<br>8-15 – Reserved | R | Preset |

**Table 7.40 Config2 Register Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| SL | 7:4 | Secondary cache line size<br>Encoding: Sets Per Way<br>0: No cache present<br>1: 4<br>2: 8<br>3: 16<br>4: 32<br>5: 64<br>6: 128<br>7: 256<br>8-15: Reserved | R | Preset |
| SA | 3:0 | Secondary cache associativity<br>Encoding: Sets Per Way<br>0: Direct mapped<br>1: 2<br>2: 3<br>3: 4<br>4: 5<br>5: 6<br>6: 7<br>7: 8<br>8-15: Reserved | R | Preset |

## 7.2.40 Config3 Register (CP0 Register 16, Select 3)

The *Config3* register encodes additional capabilities. All fields in the *Config3* register are read-only.

**Figure 7.35 Config3 Register Format**

| 31 | 30–11 | 10 | 9–7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| M | 0 | DSPP | 0 | VEIC | VInt | SP | 0 | MT | SM | TL |

**Table 7.41 Config3 Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| M | 31 | This bit is reserved to indicate if a Config4 register is present. | R | 0 |
| DSPP | 10 | DSP Present. Indicates whether support for the DSP ASE is implemented.<br>On the 34K core, this bit is always 1 since the DSP ASE is implemented. | R | Preset<br>1 |

**Table 7.41 Config3 Register Field Descriptions (Continued)**

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>VEIC</td><td>6</td><td>Support for an external interrupt controller is implemented.<br><table><tr><th>Encoding</th><th>Description</th></tr><tr><td>0</td><td>Support for EIC interrupt mode is not implemented</td></tr><tr><td>1</td><td>Support for EIC interrupt mode is implemented</td></tr></table>The value of this bit is set by the static input, SI_EICPresent. This allows external logic to communicate whether an external interrupt controller is attached to the processor or not.</td><td>R</td><td>Externally Set</td></tr>
<tr><td>VInt</td><td>5</td><td>Vectored interrupts implemented. This bit indicates whether vectored interrupts are implemented.<br><table><tr><th>Encoding</th><th>Description</th></tr><tr><td>0</td><td>Vector interrupts are not implemented</td></tr><tr><td>1</td><td>Vectored interrupts are implemented</td></tr></table>On the 34K core, this bit is always a 1 since vectored interrupts are implemented.</td><td>R</td><td>1</td></tr>
<tr><td>SP</td><td>4</td><td>Small (1KByte) page support is implemented, and the PageGrain register exists. This bit will always be 0 since small pages are not supported.<br><table><tr><th>Encoding</th><th>Description</th></tr><tr><td>0</td><td>Small page support is not implemented</td></tr><tr><td>1</td><td>Small page support is implemented</td></tr></table></td><td>R</td><td>0</td></tr>
<tr><td>MT</td><td>2</td><td>This bit indicates if the MIPS MT (multi-threading) ASE implemented.<br><table><tr><th>Encoding</th><th>Description</th></tr><tr><td>0</td><td>MIPS MT ASE is not implemented</td></tr><tr><td>1</td><td>MIPS MT ASE is implemented</td></tr></table></td><td>R</td><td>1</td></tr>
<tr><td>SM</td><td>1</td><td>This bit indicates whether the SmartMIPS™ ASE is implemented. Since SmartMIPS is not present on the 34K core, this bit will always be 0.<br><table><tr><th>Encoding</th><th>Description</th></tr><tr><td>0</td><td>SmartMIPS ASE is not implemented</td></tr><tr><td>1</td><td>SmartMIPS ASE is implemented</td></tr></table></td><td>R</td><td>0</td></tr>
<tr><td>TL</td><td>0</td><td>Trace Logic implemented. This bit indicates whether MIPS trace support is implemented.<br><table><tr><th>Encoding</th><th>Description</th></tr><tr><td>0</td><td>Trace logic is not implemented</td></tr><tr><td>1</td><td>Trace logic is implemented</td></tr></table></td><td>R</td><td>Preset</td></tr>
<tr><td>0</td><td>30:11, 9:7, 3</td><td>Must be written as zeros; returns zeros on read</td><td>0</td><td>0</td></tr>
</table>

## 7.2.41 Config7 Register (CP0 Register 16, Select 7)

The *Config7* register contains implementation specific configuration information. A number of these bits are writeable to disable certain performance enhancing features within the core.

**Figure 7.36 Config7 Register Format**

| 31 | 18 | 17 | 16 | 15 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | HCI | FPR | AR | 0 | | ES | 0 | CPOOO | NBLSU | ULB | BP | RPS | BHT | SL |

**Table 7.42 Config7 Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| 0 | 31:19, 15:9, 7 | These bits are unused and should be written as 0. | R | 0 |
| HCI | 18 | Hardware Cache Initialization: Indicates that a cache does not require initialization by software. This bit is implemented per-processor.<br>This bit will most likely only be set on simulation only cache models and not on real hardware. | R | Based on HW present |
| FPR | 17 | Floating Point Ratio: Indicates clock ratio between integer core and floating point unit on 34Kf cores. Reads as 0 on 34Kc cores.<br>**Encoding** / **Description**<br>0 – FP clock frequency is the same as the integer clock<br>1 – FP clock frequency is one-half the integer clock<br>This bit is implemented per-processor. | R | Based on HW present |
| AR | 16 | Alias removed: This bit indicates that the data cache is organized to avoid virtual aliasing problems. This bit is only set if the data cache config and MMU type would normally cause aliasing - i.e., only for the 32KB data cache and TLB-based MMU.<br>This bit is implemented per-processor. | R | Based on HW present |
| ES | 8 | Externalize Sync: If this bit is set, and if the downstream device is capable of accepting SYNC's (indicated via the pin *SI_SyncTxEn*), the SYNC instruction will cause a SYNC specific transaction to go out on the external bus. If this bit is cleared or if *SI_SyncTxEn* is deasserted, no transaction will go out, but all SYNC handling internal to the core will still be performed. When this bit is read, the value returned depends on the state of the *SI_SyncTxEn pin.* If SI_SyncTxEn is 0, a value of 0 is returned. If *SI_SyncTxEn* is 1, the value returned is the last value that was written to this bit. Refer to SYNC instruction description for more information.<br>This bit is implemented per-processor. | R/W | 0 |
| CPOOO | 6 | Out-of-order data return on the Coprocessor interfaces:Writing 1 to this bit disables the out-of-order data return for the FPU and COP2 | R/W | 0 |
| NBLSU | 5 | Non-Blocking LSU: Writing 1 to this field will lock the LSU and ALU pipelines together. This forces LSU pipeline stalls to also stall the ALU pipeline.<br>This bit is implemented per-processor. | R/W | 0 |

**Table 7.42 Config7 Register Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| ULB | 4 | Uncached Loads Blocking: Writing 1 to this field will make all uncached loads blocking.<br>This bit is implemented per-processor. | R/W | 0 |
| BP | 3 | Branch Prediction: Writing 1 to this field will disable all speculative branch prediction. The fetch unit will wait for a branch to be resolved before fetching the target or fall-through path.<br>This bit is implemented per-VPE. | R/W | 0 |
| RPS | 2 | Return Prediction Stack: Writing 1 to this field will disable the use of the Return Prediction Stack. Returns (JR ra) will stall instruction fetch until the destination is calculated.<br>This bit is implemented per-VPE. | R/W | 0 |
| BHT | 1 | Branch History Table: Writing 1 to this field will disable the dynamic branch prediction. Branches will be statically predicted taken.<br>This bit is implemented per-VPE. | R/W | 0 |
| SL | 0 | Scheduled Loads: Writing 1 to this field will make load misses blocking.<br>This bit is implemented per-processor. | R/W | 0 |

## 7.2.42 LLAddr Register (CP0 Register 17, Select 0)

The *LLAddr* register is instantiated per-TC. It stores the physical address (at a minimum to the enclosing 32-byte block) of the target location of any LL/SC sequence. The 34K core detects possible non-atomicity by checking every write made by any thread against the *LLAddr* of all other TCs.

**Figure 7.37 LLAddr Register Format**

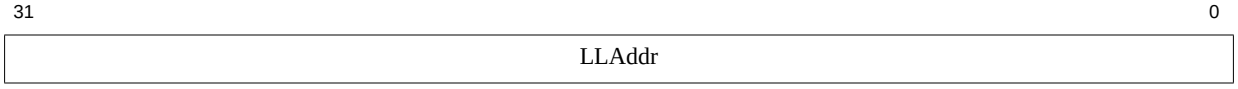



**Table 7.43 LLAddr Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| LLAddr | 31:0 | Address used by last LL instruction | R | Undefined |

## 7.2.43 WatchLo Register (CP0 Register 18, Select 0-3)

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

There are 4 sets of Watch register pairs (*WatchLo*, *WatchHi*). Two of them (select 0, 1) are associated with instruction addresses only. Thus, only the I bit is writeable, the R and W bits are tied to 0. The other two (select 2, 3) are associated with data addresses and can only be used for R or W watchpoints.

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

**Figure 7.38 WatchLo Register Format**

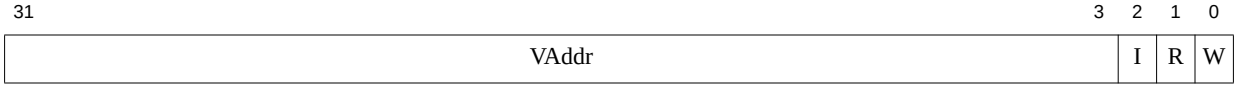



**Table 7.44 WatchLo Register Field Descriptions**

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

## 7.2.44 WatchHi Register (CP0 Register 19, Select 0-3)

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

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

There are 4 sets of Watch register pairs (*WatchLo*, *WatchHi*). Two of them (select 0, 1) are associated with instruction addresses only. Thus, only the I bit is meaningful, the *R* and *W* bits are tied to 0. The other two (select 2, 3) are associated with data addresses and can only be used for *R* or *W* watchpoints.

**Figure 7.39 WatchHi Register Format**

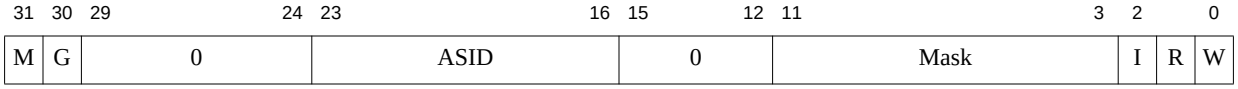



**Table 7.45 WatchHi Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| M | 31 | Indicates the presence of additional Watch registers. | R | Preset |

**Table 7.45 WatchHi Register Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| G | 30 | If this bit is one, any address that matches that specified in the *WatchLo* register causes a watch exception. If this bit is zero, the *ASID* field of the *WatchHi* register must match the *ASID* field of the *EntryHi* register to cause a watch exception. | R/W | Undefined |
| ASID | 23:16 | *ASID* value which is required to match that in the *EntryHi* register if the *G* bit is zero in the *WatchHi* register. | R/W | Undefined |
| Mask | 11:3 | Bit mask that qualifies the address in the *WatchLo* register. Any bit in this field that is a set inhibits the corresponding address bit from participating in the address match. | R/W | Undefined |
| I | 2 | This bit is set by hardware when an instruction fetch condition matches the values in this watch register pair. When set, the bit remains set until cleared by software, which is accomplished by writing a 1 to the bit. | W1C | Undefined |
| R | 1 | This bit is set by hardware when a load condition matches the values in this watch register pair. When set, the bit remains set until cleared by software, which is accomplished by writing a 1 to the bit. | W1C | Undefined |
| W | 0 | This bit is set by hardware when a store condition matches the values in this watch register pair. When set, the bit remains set until cleared by software, which is accomplished by writing a 1 to the bit. | W1C | Undefined |
| 0 | 29:24, 15:12 | Must be written as zero; returns zero on read. | 0 | 0 |

## 7.2.45 Debug Register (CP0 Register 23, Select 0)

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

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

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

- *DSS*, *DBp*, *DDBL*, *DDBS*, *DIB*, *DINT* are updated on both debug exceptions and on exceptions in debug modes
- *DExcCode* is updated on exceptions in debug mode, and is undefined after a debug exception
- *Halt* and *Doze* are updated on a debug exception, and are undefined after an exception in debug mode
- *DBD* is updated on both debug and on exceptions in debug modes

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

**Figure 7.40 Debug Register Format**

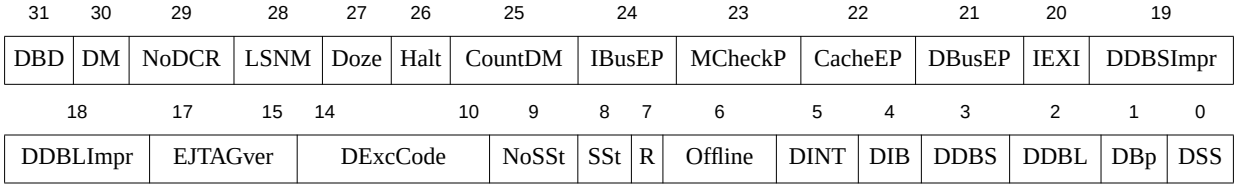



**Table 7.46 Debug Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| **Name** | **Bit(s)** | | | |
| DBD | 31 | Indicates whether the last debug exception or exception in debug mode, occurred in a branch delay slot:<br>**Encoding** / **Description**<br>0 – Not in delay slot<br>1 – In delay slot | R | Undefined |
| DM | 30 | Indicates that the processor is operating in debug mode:<br>**Encoding** / **Description**<br>0 – Processor is operating in non-debug mode<br>1 – Processor is operating in debug mode | R | 0 |
| NoDCR | 29 | Indicates whether the dseg memory segment is present:<br>**Encoding** / **Description**<br>0 – dseg is present<br>1 – No dseg present | R | 0 |
| LSNM | 28 | Controls access of load/store between dseg and main memory:<br>**Encoding** / **Description**<br>0 – Load/stores in dseg address range goes to dseg<br>1 – Load/stores in dseg address range goes to main memory | R/W | 0 |
| Doze | 27 | Indicates that the processor was in any kind of low power mode when a debug exception occurred:<br>**Encoding** / **Description**<br>0 – Processor not in low power mode when debug exception occurred<br>1 – Processor in low power mode when debug exception occurred | R | Undefined |
| Halt | 26 | Indicates that the internal system bus clock was stopped when the debug exception occurred:<br>**Encoding** / **Description**<br>0 – Internal system bus clock running<br>1 – Internal system bus clock stopped | R | Undefined |

**Table 7.46 Debug Register Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| CountDM | 25 | Indicates the Count register behavior in debug mode.<br>Encoding 0: Count register stopped in debug mode<br>Encoding 1: Count register is running in debug mode | R/W | 1 |
| IBusEP | 24 | Imprecise instruction fetch Bus Error exception Pending: All instruction bus errors are precise on the 34K core so this bit will always read as 0.<br>Set when an instruction fetch bus error event occurs or if a 1 is written to the bit by software. Cleared when a Bus Error exception on instruction fetch is taken by the processor, and by reset. If *IBusEP* is set when *IEXI* is cleared, a Bus Error exception on instruction fetch is taken by the processor, and *IBusEP* is cleared. | R | 0 |
| MCheckP | 23 | Indicates that an imprecise Machine Check exception is pending. Set when a Machine Check exception occurs or if a 1 is written to the bit by software. Cleared when a machine check exception is taken by the processor, and by reset. If *MCheckP* is set when *IEXI* is cleared, a Machine Check exception is taken by the processor, and *MCheckP* is cleared. | R | 0 |
| CacheEP | 22 | Indicates that an imprecise Cache Error is pending. | R/W1 | 0 |
| DBusEP | 21 | Data access Bus Error exception Pending: Set when an data bus error event occurs or if a 1 is written to the bit by software. Cleared when a Data Bus Error exception is taken by the processor, and by reset. If *DBusEP* is set when *IEXI* is cleared, a Data Bus Error exception is taken by the processor, and *DBusEP* is cleared. | R/W1 | 0 |
| IEXI | 20 | Imprecise Error eXception Inhibit: Controls exceptions taken due to imprecise error indications. Set when the processor takes a debug exception or exception in debug mode. Cleared by execution of the DERET instruction; otherwise modifiable by debug mode software. When *IEXI* is set, the imprecise error exception from a bus error on an instruction fetch or data access, cache error, or machine check is inhibited and deferred until the bit is cleared. | R/W | 0 |
| DDBSImpr | 19 | Indicates that an imprecise Debug Data Break Store exception was taken. | R | 0 |
| DDBLImpr | 18 | Indicates that an imprecise Debug Data Break Load exception was taken. | R | 0 |
| EJTAGver | 17:15 | EJTAG version.<br>Encoding 3: Version 3.x | R | 011 |
| DExcCode | 14:10 | Indicates the cause of the latest exception in debug mode. See Table 7.33 for a list of values.<br>Value is undefined after a debug exception. | R | Undefined |

**Table 7.46 Debug Register Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| NoSST | 9 | Indicates whether the single-step feature controllable by the *SSt* bit is available in this implementation:<br>Encoding: 0 — Single-step feature available<br>Encoding: 1 — No single-step feature available | R | 0 |
| SSt | 8 | Controls if debug single step exception is enabled:<br>Encoding: 0 — No debug single-step exception enabled<br>Encoding: 1 — Debug single step exception enabled | R/W | 0 |
| R | 7 | Reserved. Must be written as zeros; returns zeros on reads. | R | 0 |
| Offline | 6 | Implemented per-TC. When this bit is 1, TC is allowed to execute only in Debug mode. | R/W | 0 |
| DINT | 5 | Indicates that a debug interrupt exception occurred. Cleared on exception in debug mode.<br>Encoding: 0 — No debug interrupt exception<br>Encoding: 1 — Debug interrupt exception | R | Undefined |
| DIB | 4 | Indicates that a debug instruction break exception occurred. Cleared on exception in debug mode.<br>Encoding: 0 — No debug interrupt exception<br>Encoding: 1 — Debug interrupt exception | R | Undefined |
| DDBS | 3 | Indicates that a debug data break exception occurred on a store. Cleared on exception in debug mode.<br>Encoding: 0 — No debug data exception on a store<br>Encoding: 1 — Debug instruction exception on a store | R | Undefined |
| DDBL | 2 | Indicates that a debug data break exception occurred on a load. Cleared on exception in debug mode.<br>Encoding: 0 — No debug data exception on a load<br>Encoding: 1 — Debug instruction exception on a load | R | Undefined |

**Table 7.46 Debug Register Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| DBp | 1 | Indicates that a debug software breakpoint exception occurred. Cleared on exception in debug mode.<br>Encoding: 0 — No debug software breakpoint exception; 1 — Debug software breakpoint exception | R | Undefined |
| DSS | 0 | Indicates that a debug single-step exception occurred. Cleared on exception in debug mode.<br>Encoding: 0 — No debug single-step exception; 1 — Debug single-step exception | R | Undefined |

## 7.2.46 Trace Control Register (CP0 Register 23, Select 1)

The *TraceControl* register configuration is shown below.

**Figure 7.41 TraceControl Register Format**

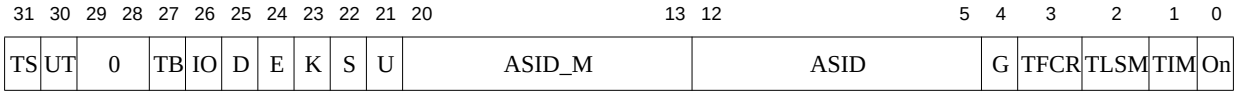



**Table 7.47 TraceControl Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| TS | 31 | The trace select bit is used to select between the hardware and the software trace control bits. A value of zero selects the external hardware trace block signals, and a value of one selects the trace control bits in the *TraceControl* register. | R/W | 0 |
| UT | 30 | This bit is used to indicate the type of user-triggered trace record. A value of zero implies a user type 1 and a value of one implies a user type 2.<br>The actual triggering of a user trace record happens on a write to the *UserTraceData* register. This is a 32-bit register for 32-bit processors and a 64-bit register for 64-bit processors. | R/W | Undefined |
| 0 | 29:28 | Reserved for future use; Must be written as zero; returns zero on read. | 0 | 0 |
| TB | 27 | Trace All Branch. When set to 1, this tells the processor to trace the PC value for all taken branches, not just the ones whose branch target address is statically unpredictable. | R/W | Undefined |

**Table 7.47 TraceControl Register Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| IO | 26 | Inhibit Overflow. This signal is used to indicate to the core trace logic that slow but complete tracing is desired. Hence, the core tracing logic must not allow a FIFO overflow and discard trace data. This is achieved by stalling the pipeline when the FIFO is nearly full, so that no trace records are ever lost. | R/W | Undefined |
| D | 25 | When set to one, this enables tracing in Debug Mode. For trace to be enabled in Debug mode, the On bit must be one, and either the G bit must be one, or the current process ASID must match the ASID field in this register.<br>When set to zero, trace is disabled in Debug Mode, irrespective of other bits. | R/W | Undefined |
| E | 24 | When set to one, this enables tracing in Exception Mode. For trace to be enabled in Exception mode, the On bit must be one, and either the G bit must be one, or the current process ASID must match the ASID field in this register.<br>When set to zero, trace is disabled in Exception Mode, irrespective of other bits. | R/W | Undefined |
| K | 23 | When set to one, this enables tracing in Kernel Mode . For trace to be enabled in Kernel mode, the On bit must be one, and either the G bit must be one, or the current process ASID must match the ASID field in this register.<br>When set to zero, trace is disabled in Kernel Mode, irrespective of other bits. | R/W | Undefined |
| S | 22 | When set to one, this enables tracing in Supervisor Mode.For trace to be enabled in Supervisor mode, the On bit must be one, and either the G bit must be one, or the current process ASID must match the ASID field in this register.<br>When set to zero, trace is disabled in Supervisor Mode, irrespective of other bits.<br>If the processor does not implement Supervisor Mode, this bit is ignored on write and returns zero on read. | R/W | Undefined |
| U | 21 | When set to one, this enables tracing in User Mode. For trace to be enabled in User mode, the On bit must be one, and either the G bit must be one, or the current process ASID must match the ASID field in this register.<br>When set to zero, trace is disabled in User Mode, irrespective of other bits. | R/W | Undefined |
| ASID_M | 20:13 | This is a mask value applied to the ASID comparison (done when the G bit is zero). A “1” in any bit in this field inhibits the corresponding ASID bit from participating in the match. As such, a value of zero in this field compares all bits of ASID. Note that the ability to mask the ASID value is not available in the hardware signal bit; it is only available via the software control register.<br>If the processor does not implement the standard TLB-based MMU, this field is ignored on write and returns zero on read. | R/W | Undefined |
| ASID | 12:5 | The ASID field to match when the G bit is zero. When the G bit is one, this field is ignored.<br>If the processor does not implement the standard TLB-based MMU, this field is ignored on write and returns zero on read. | R/W | Undefined |

**Table 7.47 TraceControl Register Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| G | 4 | When set, this implies that tracing is to be enabled for all processes, provided that other enabling functions (like U, S, etc.,) are also true. If the processor does not implement the standard TLB-based MMU, this field is ignored on write and returns 1 on read. This causes all match equations to work correctly in the absence of an ASID. | R/W | Undefined |
| TFCR | 3 | When asserted, used to trace function call and return instructions with full PC values. | R/W | Undefined |
| TLSM | 2 | When asserted, used to trace data cache load and store misses with full PC values, and potentially the data address and value as well. | R/W | Undefined |
| TIM | 1 | When asserted, used to trace instruction miss with full PC values. | R/W | Undefined |
| On | 0 | This is the master trace enable switch in software control. When zero, tracing is always disabled. When set to one, tracing is enabled whenever the other enabling functions are also true. | R/W | 0 |

## 7.2.47 Trace Control2 Register (CP0 Register 23, Select 2)

The *TraceControl2* register provides additional control and status information. Note that some fields in the *TraceControl2* register are read-only, but have a reset state of “Undefined”. This is because these values are loaded from the Trace Control Block (TCB) (see Section 11.9 “Trace Control Block (TCB) Registers (Hardware Control)”). As such, these fields in the *TraceControl2* register will not have valid values until the TCB asserts these values.

This register is only implemented if the MIPS Trace capability is present.

**Figure 7.42 TraceControl2 Register Format**

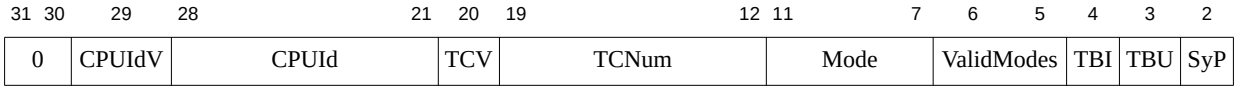



**Table 7.48 TraceControl2 Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| 0 | 31:30 | Reserved for future use; Must be written as zero; returns zero on read. | 0 | 0 |
| CPUIdV | 29 | When set, this bit specifies the VPE defined in *CPUId* must be traced. Otherwise, instructions from all VPEs are traced when other conditions for tracing are valid. This bit is ignored if *TCV* is asserted. | R/W | Undefined |
| CPUId | 28:21 | This field specifies the number of the VPE to trace when *CPUIdV* is set. | R/W | Undefined |
| TCV | 20 | When set, the *TCNum* field specifies the number of the TC that must be traced. Otherwise, instructions from all TCs are traced when other conditions for tracing are valid. | R/W | Undefined |
| TCNum | 19:12 | Specifies the TC to trace when *TCV* is set. The right-most bits only are used. | R/W | Undefined |

**Table 7.48 TraceControl2 Register Field Descriptions**

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>Mode</td><td>11:7</td><td>These 5 bits provide the same trace mode functions as the PDI_TraceMode[4:0] signal, and is described here again.<br>When tracing is turned on, this signal specifies what information is to be traced by the core. It uses 5 bits, where each bit turns on a tracing of a specific tracing mwhen that bit value is a 1. If the corresponding bit is 0, then the Trace Value shown in column two is not traced by the processor.<br>On the 34K core PC tracing is always enabled, regardless of the value on bit 23.ode. The table shows what trace value is turned on:I’
<table>
<tr><th>Bit</th><th>Trace the Following</th></tr>
<tr><td>0</td><td>PC</td></tr>
<tr><td>1</td><td>Load address</td></tr>
<tr><td>2</td><td>Store address</td></tr>
<tr><td>3</td><td>Load data</td></tr>
<tr><td>4</td><td>Store data</td></tr>
</table></td><td>R/W</td><td>Undefined</td></tr>
<tr><td>Valid-Modes</td><td>6:5</td><td>This field specifies the subset of tracing that is supported by the processor.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>00</td><td>PC tracing only</td></tr>
<tr><td>01</td><td>PC and load and store address tracing only</td></tr>
<tr><td>10</td><td>PC, load and store address, and load and store data</td></tr>
<tr><td>11</td><td>Reserved</td></tr>
</table></td><td>R</td><td>Preset</td></tr>
<tr><td>TBI</td><td>4</td><td>This bit indicates how many trace buffers are implemented by the TCB, as follows:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Only one trace buffer is implemented, and the TBU bit of this register indicates which trace buffer is implemented</td></tr>
<tr><td>1</td><td>Both on-chip and off-chip trace buffers are implemented by the TCB and the TBU bit of this register indicates to which trace buffer the traces is currently written.</td></tr>
</table>
This bit is loaded from the PDI_TBImpl signal when the PDI_SyncOffEn signal is asserted.</td><td>R</td><td>Undefined</td></tr>
<tr><td>TBU</td><td>3</td><td>This bit denotes to which trace buffer the trace is currently being written and is used to select the appropriate interpretation of the TraceControl2$_{SyP}$ field.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Trace data is being sent to an on-chip trace buffer</td></tr>
<tr><td>1</td><td>Trace Data is being sent to an off-chip trace buffer</td></tr>
</table>
This bit is loaded from the PDI_OffChipTB signal when the PDI_SyncOffEn signal is asserted.</td><td>R</td><td>Undefined</td></tr>
</table>

**Table 7.48 TraceControl2 Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| SyP | 2:0 | The period (in cycles) to which the internal synchronization counter is reset when tracing is started, or when the synchronization counter has overflowed.<br>SyP / Sync Period: 000 → $2^5$; 001 → $2^6$; 010 → $2^7$; 011 → $2^8$; 100 → $2^9$; 101 → $2^{10}$; 110 → $2^{11}$; 111 → $2^{12}$<br>This field is loaded from the *PDI_SyncPeriod* signal when the *PDI_SyncOffEn* signal is asserted. | R | Undefined |

## 7.2.48 User Trace Data Register (CP0 Register 23, Select 3)

A software write to any bits in the *UserTraceData* register will trigger a trace record to be written indicating a type 1 or type 2 user format. The type is based on the *UT* bit in the *TraceControl* register. This register cannot be written in consecutive cycles. The trace output data is **UNPREDICTABLE** if this register is written in consecutive cycles.

This register is only implemented if the MIPS Trace capability is present.

**Figure 7.43 User Trace Data Register Format**

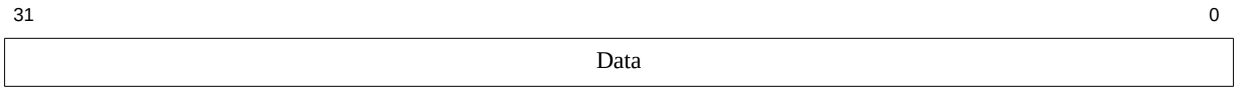



**Table 7.49 UserTraceData Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| Data | 31:0 | Software readable/writable data. When written, this triggers a user format trace record out of the PDtrace interface that transmits the Data field to trace memory. | R/W | 0 |

## 7.2.49 TraceIBPC Register (CP0 Register 23, Select 4)

The *TraceIBPC* register is used to control start and stop of tracing using an EJTAG Instruction Hardware breakpoint. TheInstruction Hardware breakpoint would then be set as a trigger source and optionally also as a Debug exception breakpoint.

This register is only implemented if both Hardware breakpoints and the MIPS Trace capability are present.

**Figure 7.44 TraceIBPC Register Format**

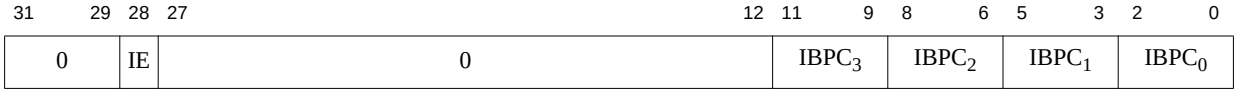



**Table 7.50 TraceIBPC Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| 0 | 31:29, 27:12 | Reserved for future implementation | R | 0/1 |
| IE | 28 | Used to specify whether the trigger signal from EJTAG instruction breakpoint should trigger tracing functions or not:<br>Encoding: 0 — Meaning: Disables trigger signals from instruction breakpoints<br>Encoding: 1 — Meaning: Enables trigger signals from instruction breakpoints | R/W | 0 |
| $IBPC_n$ | 3n-1:3n-3 | The three bits are decoded to enable different tracing modes. Table 7.52 shows the possible interpretations. Each set of 3 bits represents the encoding for the instruction breakpoint n in the EJTAG implementation, if it exists. If the breakpoint does not exist then the bits are reserved, read as zero and writes are ignored.<br>If bit 27 is zero, bits 3n-1:3n-2 are ignored, and only the bottom bit 3n-3 is used to start and stop tracing as specified in versions less than 4.00 of this specification. | R/W | 0 |

## 7.2.50 TraceDBPC Register (CP0 Register 23, Select 5)

The *TraceDBPC* register is used to control start and stop of tracing using an EJTAG Data Hardware breakpoint. The Data Hardware breakpoint would then be set as a trigger source and optionally also as a Debug exception breakpoint.

This register is only implemented if both Hardware breakpoints and the MIPS Trace capability are present

**Figure 7.45 TraceDBPC Register Format**

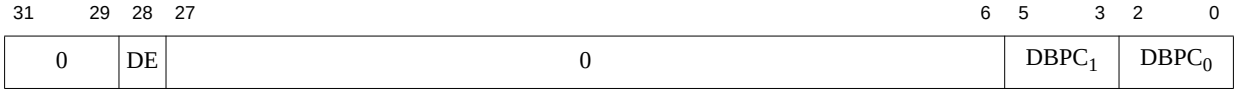



**Table 7.51 TraceDBPC Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| 0 | 31:29, 27:6 | Reserved for future implementation | R | 0/1 |

**Table 7.51 TraceDBPC Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| DE | 28 | Used to specify whether the trigger signal from EJTAG data breakpoint should trigger tracing functions or not:<br>Encoding 0: Disables trigger signals from data breakpoints<br>Encoding 1: Enables trigger signals from data breakpoints | R/W | 0 |
| $DBPC_n$ | 3n-1:3n-3 | The three bits are decoded to enable different tracing modes. Table 7.52 shows the possible interpretations. Each set of 3 bits represents the encoding for the data breakpoint n in the EJTAG implementation, if it exists. If the breakpoint does not exist then the bits are reserved, read as zero and writes are ignored.<br>If ATE is zero, bits 3n-1:3n-2 are ignored, and only the bottom bit 3n-3 is used to start and stop tracing as specified in versions less than 4.00 of this specification. | R/W | 0 |

**Table 7.52 BreakPoint Control Modes: IBPC and DBP**

| Value | Trigger Action | Description |
|---|---|---|
| 000 | Unconditional Trace Stop | Unconditionally stop tracing if tracing was turned on. If tracing is already off, then there is no effect. |
| 001 | Unconditional Trace Start | Unconditionally start tracing if tracing was turned off. If tracing is already turned off then there is no effect. |
| 010 to 111 | Not used | Reserved for future implementation |

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

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

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

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

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

In processors that implement the MIPS16 ASE, a read of the *DEPC* register (via MFC0) returns the following value in the destination GPR:

```
GPR[rt] ← DebugExceptionPC31..1 || ISAMode0
```

That is, the upper 31 bits of the debug exception PC are combined with the lower bit of the ISAMode field and written to the GPR.

Similarly, a write to the *DEPC* register (via MTC0) takes the value from the GPR and distributes that value to the debug exception PC and the ISAMode field, as follows

```
DebugExceptionPC ← GPR[rt]31..1 || 0
ISAMode ← 2#0 || GPR[rt]0
```

That is, the upper 31 bits of the GPR are written to the upper 31 bits of the debug exception PC, and the lower bit of the debug exception PC is cleared. The upper bit of the ISAMode field is cleared and the lower bit is loaded from the lower bit of the GPR.

**Figure 7.46 DEPC Register Format**

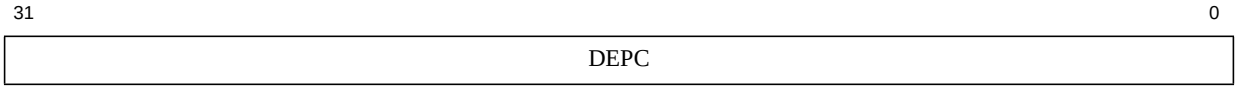



**Table 7.53 DEPC Register Formats**

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

## 7.2.52 Performance Counter Register (CP0 Register 25, select 0-7)

The 34K processor defines four performance counters and four 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 7.54.

**Table 7.54 Performance Counter Register Selects**

| Select[2:0] | Register |
|---|---|
| 0 | Register 0 Control |
| 1 | Register 0 Count |
| 2 | Register 1 Control |
| 3 | Register 1 Count |
| 4 | Register 2 Control |
| 5 | Register 2 Count |
| 6 | Register 3 Control |
| 7 | Register 3 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 counters are AND'ed with an interrupt enable bit, *IE*, of their respective control register, and then OR'ed together to create the *SI_PCI* output. This signal is combined with one of the *SI_Int* pins to signal an interrupt to the core. 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.

For a more detailed desciption of the performance counter events, please refer to the Programmer's Guide.

**Figure 7.47 Performance Counter Control Register**

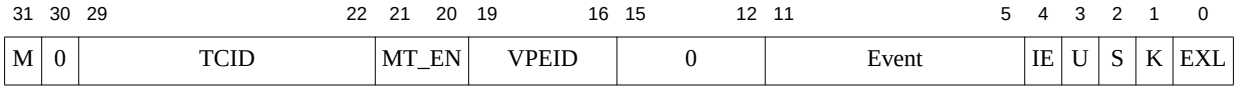



**Table 7.55 Performance Counter Control Register Field Descriptions**

| 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 | Preset |
| TCID | 29:22 | Specifies which TC events should be counted for if per-TC counting is enabled. | R/W | Undefined |
| MT_EN | 21:20 | Specifies which events should be counted:<br>Encoding / Meaning<br>00 – Count events from all TCs & VPEs<br>01 – Count events from all TCs of the VPE specified in VPEID<br>10 – Count events from the TC specified in TCID<br>11 – Reserved | R/W | Undefined |
| VPEID | 19:16 | Specifies which VPE events should be counter for if per-VPE counting is enabled. | R/W | Undefined |
| Event | 11:5 | Counter event enabled for this counter. Possible events are listed in Table 7.56. | 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, 15:12 | Must be written as zeroes; returns zeroes when read. | 0 | 0 |

Table 7.56 describes the events countable with the two performance counters. The type field indicates whether the event can be per-TC (T), per-VPE (V), or per-Processor (P). TC countable events can also be counted in VPE or Processor modes, and VPE countable events can also be counted in Processor mode. The operation of a counter is

**Table 7.56 Performance Counter Count Register Field Descriptions**

| Event Num | Counter 0/2 | Type | Counter 1/3 | Type |
|---|---|---|---|---|
| 0 | Cycles | P | Cycles | P |
| 1 | Instructions completed | T | Instructions completed | T |
| 2 | branch instructions | T | Branch mispredictions | T |
| 3 | JR r31 (return) instructions | T | JR r31 mispredictions | T |
| 4 | JR (not r31) instructions | T | JR r31 not predicted | T |
| 5 | ITLB accesses | T | ITLB misses | T |
| 6 | DTLB accesses | T | DTLB misses | T |
| 7 | JTLB instruction accesses | T | JTLB instruction misses | T |
| 8 | JTLB data accesses | T | JTLB data misses | T |
| 9 | Instruction Cache accesses | T | Instruction cache misses | T |
| 10 | Data cache accesses | T | Data cache writebacks | T |
| 11 | Data cache misses | T | Data cache misses | T |
| 12 | Reserved | | Reserved | |
| 13 | Reserved | | Reserved | |
| 14 | integer instructions completed | T | FPU instructions completed | T |
| 15 | loads completed | T | stores completed | T |
| 16 | J/JAL completed | T | MIPS16 instructions completed | T |
| 17 | no-ops completed | T | integer multiply/divide completed | T |
| 18 | Stalls | P | replay traps (other than uTLB) | T |
| 19 | SC instructions completed | T | SC instructions failed | T |
| 20 | Prefetch instructions completed | T | Prefetch instructions completed with cache hit | T |
| 21 | L2 cache writebacks | P | L2 cache accesses | P |
| 22 | L2 cache misses | P | L2 cache single bit errors corrected | P |
| 23 | Exceptions taken | T | Single Threaded Mode | P |
| 24 | cache fixup | T | Refetches | T |
| 25 | IFU stalls | P | ALU stalls | P |
| 26 | DSP Instructions Completed | T | ALU-DSP Saturations Done | T |
| 27 | Reserved | | MDU-DSP Saturations Done | T |
| 28 | Impl. specific PM event | T | Impl. specific Cp2 event | T |
| 29 | Impl. specific ISPRAM event | T | Impl. specific DSPRAM event | T |
| 30 | Impl. specific CorExtend event | T | Reserved | |

**Table 7.56 Performance Counter Count Register Field Descriptions**

| Event Num | Counter 0/2 | Type | Counter 1/3 | Type |
|---|---|---|---|---|
| 31 | Impl. specific XYM event | T | Impl. specific ITC event | T |
| 32 | ITC Loads | T | ITC Stores | T |
| 33 | Uncached Loads | T | Uncached Stores | T |
| 34 | FORK Instructions completed | T | YIELD instruction completed | T |
| 35 | CP2 Arithmetic Instns Completed | T | CP2 To/From Instns completed | T |
| 36 | Reserved | | | |
| 37 | I$ Miss Stall cycles | T | D$ miss stall cycles | T |
| 38 | L2 Instn miss stall cycles | P | L2 data miss stall cycles | P |
| 39 | D$ miss cycles | P | L2 miss cycles | P |
| 40 | Uncached stall cycles | T | ITC stall cycles | T |
| 41 | MDU stall cycles | T | FPU stall cycles | T |
| 42 | CP2 stall cycles | T | CorExtend stall cycles | T |
| 43 | ISPRAM Stall Cycles | T | DSPRAM stall cycles | T |
| 44 | CACHE Instn stall cycles | P | Reserved | |
| 45 | Load to Use stalls | T | ALU to AGEN stalls | T |
| 46 | Other interlock stalls | T | Branch mispredict stalls | Pi |
| 47 | Relax bubbles | V | Reserved | |
| 48 | IFU FB full refetches | T | FB entry allocated | P |
| 49 | EJTAG Instruction Triggerpoints | T | EJTAG Data Triggerpoints | T |
| 50 | FSB < 1/4 full | P | FSB 1/4-1/2 full | P |
| 51 | FSB > 1/2 full | P | FSB full pipeline stalls | P |
| 52 | LDQ < 1/4 full | P | LDQ 1/4-1/2 full | P |
| 53 | LDQ > 1/2 full | P | LDQ full pipeline stalls | P |
| 54 | WBB < 1/4 full | P | WBB 1/4-1/2 full | P |
| 55 | WBB > 1/2 full | P | WBB full pipeline stalls | P |
| 56-63 | Reserved | | | |

**Table 7.57 Event Descriptions**

| Event Name | Counter | Event Number | Description |
|---|---|---|---|
| Cycles | 0/1/2/3 | 0 | Total number of cycles |
| **Instruction Completion**: The following events indicate completion of various types of instructions | | | |
| Instructions | 0/1/2/3 | 1 | Total number of instructions completed. |

**Table 7.57 Event Descriptions**

| Event Name | Counter | Event Number | Description |
|---|---|---|---|
| Branch instns | 0/2 | 2 | Counts all branch instructions that completed |
| JR R31 (return) instns | 0/2 | 3 | Counts all JR R31 instructions that completed |
| JR (not R31) | 0/2 | 4 | Counts all JR $xx (not $31) and JALR instructions (indirect jumps). |
| Integer instns | 0/2 | 14 | Non floating point, non Coprocessor 2 instructions. |
| FPU instns | 1/3, | 14 | Floating point instructions |
| Loads | 0/2 | 15 | Includes both integer and coprocessor loads |
| Stores | 1/3 | 15 | Includes both integer and coprocessor stores |
| J/JAL | 0/2 | 16 | Direct Jump (And Link) instruction |
| MIPS16e | 1/3 | 16 | All MIPS16e instruction |
| no-ops | 0/2 | 17 | This includes all instructions that normally write to a GPR, but where the destination register was set to r0 |
| Integer Multiply/Divide | 1/3 | 17 | Counts all Integer Multiply/Divide instructions (MULxx, DIVx, MADDx, MSUBx) |
| SC | 0/2 | 19 | Counts conditional stores regardless of whether they succeeded. |
| PREF | 0/2 | 20 | Note that this only counts PREFs that are actually attempted. PREFs to uncached addresses or ones with translation errors are not counted |
| DSP instns | 0/2 | 26 | Counts DSP ASE instructions. |
| ITC Loads | 0/2 | 32 | Counts loads issued to ITC. This includes loads that are rolled back due to the parent TC getting halted or taking an exception. |
| ITC Stores | 1/3, | 32 | Counts stores issued to ITC. This includes stores that are rolled back due to the parent TC getting halted or taking an exception |
| Uncached Loads | 0/2 | 33 | Include both Uncached and Uncached Accelerated CCAs |
| Uncached Stores | 1/3, | 33 | |
| FORK instns | 0/2 | 34 | MT ASE Fork instruction |
| YIELD instns | 1/3 | 34 | MT ASE YIELD instruction |
| Cp2 Arithmetic instns | 0/2 | 35 | Counts Coprocessor 2 register-to-register instructions. |
| Cp2 To/From instns | 1/3 | 35 | Includes move to/from, control to/from, and cop2 loads and stores |
| **Instruction execution events** | | | |
| Branch mispredicts | 1/3 | 2 | Counts all branch instructions which completed, but were mispredicted. |
| JR r31 mispredicts | 1/3 | 3 | Counts all JR $31 instructions which completed, used the RPS for a prediction, but were mispredicted |
| JR r31 not-predicted | 1/3 | 4 | RPS will be dynamically associated with only one TC, returns on other TCs will not be predicted |
| ITLB accesses | 0/2 | 5 | Counts ITLB accesses that are due to fetches showing up in IF stage of the pipe and do not use fixed mapping or are not in unmapped space. If an address is fetched twice down the pipe (as in the case of a cache miss), that instruction will count 2 ITLB accesses. Also, since each fetch gets us 2 instructions, there is one access marked per double word |

**Table 7.57 Event Descriptions**

| Event Name | Counter | Event Number | Description |
|---|---|---|---|
| ITLB misses | 1/3 | 5 | Counts all misses in ITLB except ones that are on the back of another miss. We cannot process back to back misses and thus those are ignored for this purpose. Also ignored if there is some form of address error. |
| DTLB accesses | 0/2 | 6 | Counts DTLB access including those in unmapped address spaces. |
| DTLB misses | 1/3 | 6 | Counts DTLB misses. Back to back misses that result in only one DTLB entry getting refilled are counted as a single miss. |
| JTLB instruction accesses | 0/2 | 7 | Instruction JTLB accesses are counted exactly the same as ITLB misses |
| JTLB instruction misses | 1/3 | 7 | Counts instruction JTLB accesses that result in no match or a match on an invalid translation |
| JTLB data accesses | 0/2 | 8 | Data JTLB accesses. |
| JTLB data misses | 1/3 | 8 | Counts data JTLB accesses that result in no match or a match on an invalid translation |
| I$ accesses | 0/2 | 9 | Counts every time the instruction cache is accessed. All replays, wasted fetches etc. are counted. For example, following a branch, even the prediction is taken, the fall through access is counted. |
| I$ misses | 1/3 | 9 | Counts all instruction cache misses that result in a bus request. |
| D$ accesses | 0/2 | 10 | Counts cached loads and stores. |
| D$ writebacks | 1/3 | 10 | Counts cache lines written back to memory due to replacement or cacheops. |
| D$ misses | 0/1/2/3 | 11 | Counts loads and stores that miss in the cache |
| SC instructons failed | 1/3 | 19 | SC instruction that did not update memory |
| PREF completed with cache hit | 1/3 | 20 | Counts PREF instructions that hit in the cache |
| L2 Cache Writebacks | 0/2 | 21 | Counts cache lines written back to memory due to replacement or cacheops |
| L2 Cache Accesses | 1/3 | 21 | Number of accesses to L2 Cache |
| L2 Cache Misses | 0/2 | 22 | Number of accesses that missed in the L2 cache |
| L2 Cache Single Bit Errors Corrected | 1/3 | 22 | Single bit errors in L2 Cache that were detected and corrected |
| Exceptions Taken | 0/2 | 23 | Any type of exception taken |
| ALU-DSP Saturations Done | 1/3 | 26 | Number of times a DSP instruction caused an ALU accumulator to saturate |
| MDU-DSP Saturations Done | 1/3 | 27 | Number of times a DSP instruction caused an MDU accumulator to saturate |
| EJTAG instruction triggers | 0/2 | 49 | Number of times an EJTAG Instruction Trigger Point condition matched |
| EJTAG data triggers | 1/3 | 49 | Number of times an EJTAG Data Trigger Point condition matched |
| **Pipeline Fun** | | | |

**Table 7.57 Event Descriptions**

| Event Name | Counter | Event Number | Description |
|---|---|---|---|
| Replays | 1/3 | 18 | Counts all replayed instructions. When a long stall condition is detected, instructions are flushed back to the instruction buffer to allow other TCs to advance. The flushed instructions must then be replayed. Count includes instructions that have been replayed multiple times. |
| Single Threaded mode | 1/3 | 23 | Counts all cycles where one and only one TC is eligible for scheduling instructions. Does not include cycles when zero or two or more TCs are eligible for scheduling. |
| Refetches | 1/3 | 24 | Counts the number of replayed instructions that are sent back to IFU to be refetched. If a replay condition is detected, but the instruction is no longer in the instruction buffer, the IFU will need to refetch it. |
| Cache fixup | 0/2 | 24 | Counts cycles where the LSU is in fixup and cannot accept a new instruction from the ALU. Fixups are replays within the LSU that occur when an instruction needs to re-access the cache or the DTLB |
| **General Stalls** | | | |
| IFU Stalls | 0/2 | 25 | Counts the number of cycles where the fetch unit is not providing a valid instruction to the ALU. |
| ALU Stalls | 1/3 | 25 | Counts the number of cycles where the ALU pipeline cannot advance. |
| Stalls | 0/2 | 18 | Counts the total number of cycles where no instructions are issued by IFU to ALU (the RF stage does not advance). This includes both of the previous two events. This is different than the sum of them though because cycles when both stalls are active will only be counted once. |
| **Specific stalls** - these events will count the number of cycles lost due to this. This will include bubbles introduced by replays within the pipe. If multiple stall sources are active simultaneously, the counters for each of the active events will be incremented. | | | |
| I$ miss stall cycles | 0/2 | 37 | When IFU stalls because an I$ miss caused a TC not to have any runnable instructions. Ignores the stalls due to ITLB misses as well as the 4 cycles following a redirect. |
| D$ miss stall cycles | 1/3 | 37 | Counts all cycles where integer pipeline waits on Load return data due to a D-cache miss. The LSU can signal a “long stall” on a D-cache misses, in which case the waiting TC might be rescheduled so other TCs can execute instructions till the data returns. |
| L2 Instn Miss Stall cycles | 0/2 | 38 | L2 Instruction miss is stalling the pipeline. |
| L2 Data Miss stall cycles | 1/3 | 38 | L2 Data miss is stalling the pipeline |
| D$ miss cycles | 0/2 | 39 | D$ miss is outstanding, but not necessarily stalling the pipeline. The difference between this and D$ miss stall cycles can show the gain from non-blocking cache misses. |
| L2 miss cycles | 1/3 | 39 | L2 miss is outstanding, but not necessarily stalling the pipeline |
| Uncached Stall Cycles | 0/2 | 40 | Cycles where the processor is stalled on an uncached fetch, load, or store |
| ITC Load/Store stall | 1/3 | 40 | Counts all cycles where a TC is waiting on a ITC load or store to complete and there are no other TCs that can execute. |
| MDU stall | 0/2 | 41 | Counts all cycles where integer pipeline waits on MDU return data. MDU block can signal a “long stall”, in which case the waiting TC might be rescheduled so other TCs can execute instructions till the data returns. |

**Table 7.57 Event Descriptions**

| Event Name | Counter | Event Number | Description |
|---|---|---|---|
| FPU stall | 1/3 | 41 | Counts all cycles where integer pipeline waits on FPU return data. FPU block can signal a “long stall”, in which case the waiting TC might be rescheduled so other TCs can execute instructions till the data returns. |
| Cp2 stall | 0/2 | 42 | Counts all cycles where integer pipeline waits on CP2 return data. CP2 block can signal a “long stall”, in which case the waiting TC might be rescheduled so other TCs can execute instructions till the data returns. |
| CorExtend stall | 1/3 | 42 | Counts all cycles where integer pipeline waits on CorExtend return data. CorExtend block can signal a “long stall”, in which case TC might be rescheduled so other TCs can execute instructions till the data returns. |
| ISPRAM stall | 0/2 | 43 | Count all pipeline bubbles that are a result of multicycle ISPRAM access. Pipeline bubbles are defined as all cycles that IFU doesn’t present an instruction to ALU. The four cycles after a redirect are not counted. |
| DSPRAM stall | 1/3 | 43 | Counts stall cycles created by an instruction waiting for access to DSPRAM. |
| CACHE instn stall | 0/2 | 44 | Counts all cycles where pipeline is stalled due to CACHE instructions. Includes cycles where CACHE instructions themselves are stalled in the ALU, and cycles where CACHE instructions cause subsequent instructions to be stalled |
| Load to Use stall | 0/2 | 45 | Counts all cycles where integer pipeline waits on Load return data. LSU block can signal a “long stall”, in which case the waiting TC might be rescheduled so other TCs can execute instructions till the data returns. |
| ALU to AGEN stall | 1/3 | 45 | Counts stall cycles due to skewed ALU where the bypass to the address generation takes an extra cycle. |
| Other interlocks | 0/2 | 46 | Counts all cycles where integer pipeline waits on return data from MFC0, RDHWR, MFTTR instructions. |
| Branch mispredict stalls | 1/3 | 46 | This counts the number of cycles from a mispredicted branch until the next non-delay slot instruction executes. Count is not very meaningful when executing from multiple TCs |
| Relax stalls | 0/2 | 47 | Number of cycles that a low power op is ‘executed’ as requested by Policy Manager |
| FSB full pipeline stalls | 1/3 | 51 | Cycles where the pipeline is stalled because the Fill-Store Buffer in LSU is full. |
| LDQ full pipeline stalls | 1/3 | 53 | Cycles where the pipeline is stalled because the Load Data Queue in the LSU is full |
| Write Back Buffer full stalls | 1/3 | 55 | Cycles where the pipeline is stalled because the WriteBack Buffer in the BIU is full. |
| **Implementation specific events** - Modules that can be replaced by the customer will have an event signal associated with them. The core will output a mask indicating which TCs events should be counted for, based on the state of the control register. | | | |
| Policy Manager | 0 | 28 | |
| Cp2 | 1 | 28 | |

**Table 7.57 Event Descriptions**

| Event Name | Counter | Event Number | Description |
|---|---|---|---|
| ISPRAM | 0 | 29 | |
| DSPRAM | 1 | 29 | |
| XYM | 0 | 31 | |
| ITC | 1 | 31 | |
| CorExtend | 0 | 30 | |
| **Buffer usage events** - These count the number of cycles that buffers within the core spend at various levels of fullness. These events cannot be qualified by TC or VPE number | | | |
| Fill Store Buffer < 1/4 full | 0/2 | 50 | |
| Fill Store Buffer 1/4 to 1/2 full | 1/3 | 50 | |
| Fill Store Buffer > 1/2 full | 0/2 | 51 | |
| Load Data Queue < 1/4 full | 0/2 | 52 | |
| Load Data Queue 1/4 to 1/2 full | 1/3 | 52 | |
| Load Data Queue > 1/2 full | 0/2 | 53 | |
| Write Back Buffer < 1/4 full | 0/2 | 54 | |
| Write Back Buffer 1/4 to 1/2 full | 1/3 | 54 | |
| Write Back Buffer > 1/2 full | 0/2 | 55 | |
| IFU Fill buffer allocated | 1/3 | 48 | Number of cycles where at least one of the IFU fill buffers is allocated (miss pending) |
| Refetches due to all IFU Fill Buffers allocated | 0/2 | 48 | Counts the number of times an instruction cache miss was detected, but both fill buffers were already allocated. |

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.

**Figure 7.48 Performance Counter Count Register**

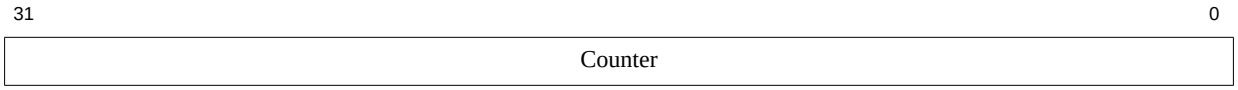



**Table 7.58 Performance Counter Count Register Field Descriptions**

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

## 7.2.53 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 and scratchpad RAMs.

Parity protection can be enabled or disabled using the *PE* bit. When parity is enabled and the *PO* bit is deasserted, the CACHE Index Store Tag and Index Store Data operations will internally generate parity to be written into the RAM arrays. However, when the *PO* bit is asserted, tag array parity is written using the *P* bit of the *TagLo* register and data array parity is written using the *PI*/*PD* bits of *ErrCtl*.

ECC protection for the secondary cache is controlled by a combination of PE and the L2P bits.

A CACHE Index Load Tag operation to the instruction cache will update the *PCI* field with the instruction precode bits from the data array and the *PI* field with the parity bits from the data array if parity is supported. A CACHE Index Load Tag operation to the data cache will cause the *PD* bits to be updated with the byte parity for the selected word of the data array if parity is implemented. If parity is disabled or not implemented, the contents of the *PI* and *PD* fields after a CACHE Index Load Tag operation will be 0.

The *PCO* field can be used for testing the precode bits of the instruction cache data array. When the *PCO* bit is cleared, the CACHE Index Store Data instruction will internally generate the precode bits to be written into the instruction cache data array. However, when the *PCO* bit is set, the CACHE Index Store Data instruction will write the value in the *PCI* field to the precode bits in the data array. Setting an illegal value in the precode bits will cause unpredictable behavior. This mechanism should only be used for software testing of the cache arrays. Furthermore, the cache should be flushed after testing.

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 instead of the TAG RAMs. In addition, when the *WST* bit is set, the CACHE Index Store Data can be used for testing the data RAM.

Setting the *SPR* bit enables scratchpad test mode. This mode allows reading and writing of the scratchpad pseudo-tags as well the scratchpad data array.

Setting the *ITC* bit enables access to the ITC pseudo-tags that control the addressing information

At most one of the *WST*, *SPR*, and *ITC* bits should be set. Setting multiple bits will lead to unpredictable behavior.

**Figure 7.49 ErrCtl Register**

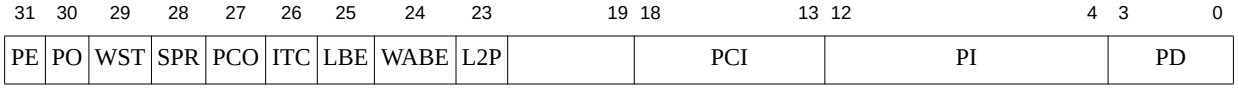



**Table 7.59 ErrCtl Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| PE | 31 | Parity Enable. This bit enables or disables the cache parity protection for both the instruction cache and the data cache.<br><br>Encoding 0: Parity disabled<br>Encoding 1: Parity enabled<br><br>This field is only write-able if the cache parity option was implemented when the core was built. If cache parity is not supported, this field is always read as 0. Software can test for cache parity support by attempting to write a 1 to this field, then read back the value. | R or R/W | 0 |

**Table 7.59 ErrCtl Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| PO | 30 | Parity Overwrite. If set, the *PI*/*PD* fields of this register overwrites calculated parity for the data array. In addition, the *P* field of the TagLo register overwrites calculated parity for the tag array. This bit only has significance during CACHE Index Store Tag and CACHE Index Store Data operations.<br>Encoding: 0 = Use calculated parity; 1 = Override calculated parity | R/W | 0 |
| WST | 29 | Way Selection Test. If set, way-selection RAM test mode is enabled. This affects only the CACHE instruction operation.<br>Encoding: 0 = Test mode disabled; 1 = Test mode enabled | R/W | 0 |
| SPR | 28 | ScratchPadRAM test. If set, indexed CACHE instructions operate on the ScratchPad RAM. Undefined behavior if ScratchPad RAM is not present | R/W | 0 |
| PCO | 27 | Precode override. If set, the contents of the *PCI* field overwrite the calculated precode bits when data is written to the instruction cache for indexed CACHE instruction operations.<br>Encoding: 0 = Use calculated precode; 1 = Override calculated precode | R/W | 0 |
| ITC | 26 | InterThread Communication. If set, Index Load Tag and Index Store Tag CACHE instructions operate on the ITC tag. | R/W | 0 |
| LBE | 25 | Bit indicating that the most recent Data Bus Error was involved a load instruction. A Per-TC BE bit will indicate which TCs were impacted. | R | Undefined |
| WABE | 24 | Bit indicating that the most recent Data Bus Error was due to a write allocate and that store data was lost. There is no indication of which TC(s) the store request came from.<br>It is possible for both *LBE* and *WABE* to be set if the bus error was on a line being used for both loads and stores. | R | Undefined |
| L2P | 23 | L2 ECC Enable. This bit can be set only if the L2 is ECC capable. This bit in conjunction with the PE bit enables or disables the ECC protection for the L2 cache:<br>PE=1, L2P=0, L2 check=1<br>PE=1, L2P=1, L2 check=0<br>PE=0, L2P=0, L2 check=0<br>PE=0, L2P=1, L2 check=1 | R/W | 0 |
| 0 | 22:19 | Must be written as zeroes; returns zeroes when read. | 0 | 0 |

**Table 7.59 ErrCtl Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| PCI | 18:13 | Instruction precode bits read from or written to the instruction cache data RAM. | R/W | Undefined |
| PI | 12:4 | Parity bit read from or written to instruction cache data RAM.<br>Bits: 12 — Even parity bit for the pre-code bits; 11:4 — Per-byte even parity bits for the 64b of data | R/W | Undefined |
| PD | 3:0 | Parity bits read from or written to data cache data RAM. *PD[0]* is even parity for the least-significant byte of the requested data. | R/W | Undefined |

## 7.2.54 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. The format of the cache error register is different for Primary caches and the Secondary Cache. The EC bit ([30]) indicates the format to be used for decoding the contents of the CacheErr register.

**Figure 7.50 CacheErr Register (Primary Caches)**

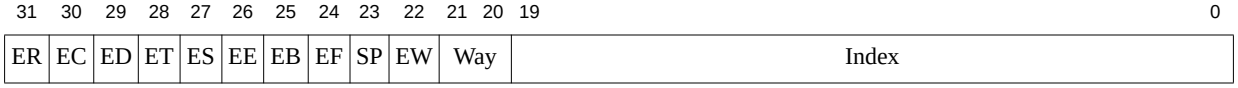



**Table 7.60 CacheErr Register Field Descriptions (Primary Caches)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| ER | 31 | Error Reference. Indicates the type of reference that encountered an error.<br>Encoding: 0 — Instruction; 1 — Data | R | Undefined |
| EC | 30 | Indicates the cache level at which the error was detected:<br>Encoding: 0 — Primary; 1 — Non-primary | R | Undefined |
| ED | 29 | Error Data. Indicates a data RAM error.<br>Encoding: 0 — No data RAM error detected; 1 — Data RAM error detected | R | Undefined |

**Table 7.60 CacheErr Register Field Descriptions (Primary Caches) (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| ET | 28 | Error Tag. Indicates a tag RAM error.<br>Encoding: 0 — No tag RAM error detected; 1 — Tag RAM error detected | R | Undefined |
| ES | 27 | Error source. Indicates whether error was caused by internal processor or external snoop request.<br>Encoding: 0 — Error on internal request; 1 — Error on external request | R | Undefined |
| EE | 26 | Error external: Indicates whether a bus parity error was detected. Not supported | R | 0 |
| EB | 25 | Error Both. Indicates that a data cache error occurred in addition to an instruction cache error.<br>Encoding: 0 — No additional data cache error; 1 — Additional data cache error<br>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 |
| EF | 24 | Error Fatal. Indicates that a fatal cache error has occurred.<br>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.<br>The following 6 cases are indicated as fatal cache errors by the *EF* bit:<br>1 Dirty parity error in dirty victim (dirty bit cleared in tag)<br>2 Tag parity error in dirty victim<br>3 Data parity error in dirty victim<br>4 WB store miss and EW error at the requested index<br>5 Dual/Triple errors from different transactions, e.g. scheduled and non-scheduled load.<br>6 Multiple data cache errors detected before the first instruction of the cache error handler is issued.<br>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 |

**Table 7.60 CacheErr Register Field Descriptions (Primary Caches) (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| SP | 23 | Scratchpad. Indicates Scratchpad RAM parity error.<br>Encoding 0: No Scratchpad RAM error detected<br>Encoding 1: Scratchpad RAM error detected | R | 0 |
| EW | 22 | Error Way. Indicates a way selection RAM error.<br>Encoding 0: No way selection RAM error detected<br>Encoding 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) or Scratchpad RAM error is detected (SP=1). | R | Undefined |
| Index | 19:0 | Index. Specifies the cache or Scratchpad RAM 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.<br>Bits [19:16] are only used used for errors in the Scratchpad RAM. | R | Undefined |

**Figure 7.51 CacheErr Register (Secondary Cache)**

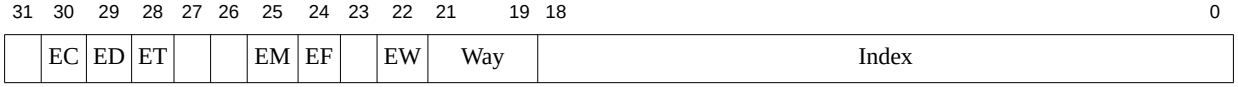



**Table 7.61 CacheErr Register Field Descriptions (Secondary Cache)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| Reserved | 31 | Reserved | R | Undefined |
| EC | 30 | Indicates the cache level at which the error was detected:<br>Encoding 0: Primary<br>Encoding 1: Non-primary | R | Undefined |

**Table 7.61 CacheErr Register Field Descriptions (Secondary Cache) (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| ED | 29 | Error Data. Indicates a data RAM error.<br>Encoding 0: No data RAM error detected<br>Encoding 1: Data RAM error detected | R | Undefined |
| ET | 28 | Error Tag. Indicates a tag RAM error.<br>Encoding 0: No tag RAM error detected<br>Encoding 1: Tag RAM error detected | R | Undefined |
| Reserved | 27 | Reserved | R | Undefined |
| Reserved | 26 | Reserved | R | Undefined |
| EM | 25 | Error Multi. Indicates that a cache error occurred in multiple L2 arrays.<br>Encoding 0: No multi error<br>Encoding 1: Multi error<br>In the case of multiple errors, the Tag ram error has the highest priority, followed by the Data ram error, followed by the Way Select ram. Only the highest priority error information is recorded in the CacheErr register. | R | Undefined |
| EF | 24 | Error Fatal. Indicates that a fatal cache error has occurred.<br>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.<br><br>This bit is set when a second L2 error occurs before taking the exception for the first L2 error. | R | Undefined |
| Reserved | 23 | Reserved | R | Undefined |
| EW | 22 | Error Way. Indicates a way selection RAM error.<br>Encoding 0: No way selection RAM error detected<br>Encoding 1: Way selection RAM error detected | R | Undefined |
| Way | 21:19 | Way. Specifies the cache way in which the error was detected. It is not valid if a Tag RAM error is detected (ET=1) or Scratchpad RAM error is detected (SP=1). | R | Undefined |

**Table 7.61 CacheErr Register Field Descriptions (Secondary Cache) (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| Index | 18:0 | 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 *Config2* 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 |

## 7.2.55 TagLo Register (CP0 Register 28, Select 0,2,4)

The *TagLo* register acts as the interface to the cache tag array. The Index Store Tag and Index Load Tag operations of the CACHE instruction use the *TagLo* register as the source of tag information. Note that the 34K core does not implement the *TagHi* register.

When the *WST* bit of the *ErrCtl* register is asserted, this register becomes the interface to the way-selection RAM. In this mode, the fields are redefined to give appropriate access the contents of the WS array instead of the Tag array. Refer to Figure 9.2 for the layout of the way-selection RAM.

Note that there are separate registers for each of the caches (L1 I-cache: select 0, L1 D-cache: select 2, L2 cache: select 4).

**Figure 7.52 TagLo Register Format (ErrCtl$_{WST}$=0, ErrCtl$_{SPR}$=0)**

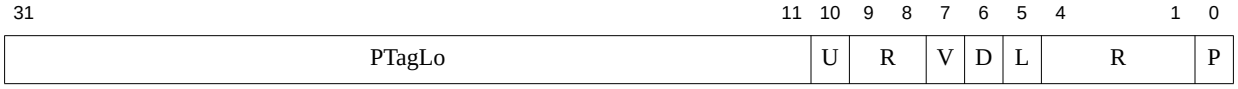



**Figure 7.53 TagLo Register Format (ErrCtl$_{WST}$=1, ErrCtl$_{SPR}$=0)**

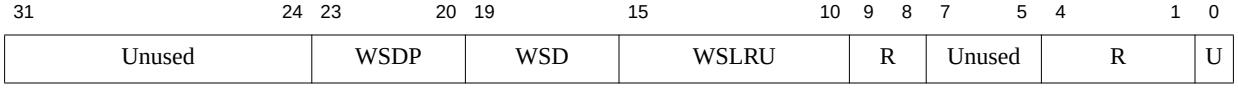



**Figure 7.54 TagLo Register Format (ErrCtl$_{WST}$=0, ErrCtl$_{SPR}$=1)**

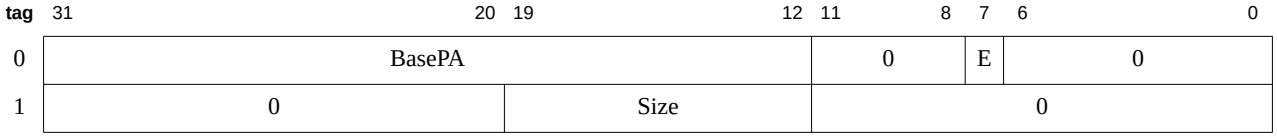



**Table 7.62 TagLo Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| Unused/U | various | Not used in certain modes of operation. | R/W | Undefined |

**Table 7.62 TagLo Register Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| PTagLo | 31:11 | This field contains the physical address of the cache line. Bit 31 corresponds to bit 31 of the PA and bit 11 corresponds to bit 11 of the PA.<br>Bit 11 is only used when 8KB caches are implemented. For other cache sizes, this bit will not exist in the tag and will be written as a 0 on IndexLoadTag operations. | R/W | Undefined |
| R | 9:8, 4:1 | Must be written as zero; returns zero on read. | 0 | 0 |
| V | 7 | This field indicates whether the cache line is valid. | R/W | Undefined |
| D | 6 | This field indicates whether the cache line is dirty. It will only be set if bit 7 (valid) is also set. For L1 I-cache, this field must be written as zero and returns zero on read. | R/W | Undefined |
| L | 5 | Specifies the lock bit for the cache tag. When this bit is set, and the valid bit is set, the corresponding cache line will not be replaced by the cache replacement algorithm. | R/W | Undefined |
| P | 0 | Parity. Specifies the parity bit for the cache tag. This bit is updated with tag array parity on CACHE Index Load Tag operations and used as tag array parity on Index Store Tag operations when the *PO* bit of the *ErrCtl* register is set.<br>NOTE: For the Data cache, this parity does not cover the dirty bit; the dirty bit has a separate parity bit placed in the way selection RAM. | R/W | Undefined |
| WSDP | 23:20 | Dirty Parity (Optional, D-side only). This field contains the value read from the WS array during a CACHE Index Load WS operation. If the *PO* field of the *ErrCtl* register is asserted, then this field is used to store the dirty parity bits during a CACHE Index Store WS operation. | R/W | Undefined |
| WSD | 19:16 | Dirty bits (D-side only). This field contains the value read from the WS array after a CACHE Index Load WS operation. It is used to store into the WS array during CACHE Index Store WS operations. | R/W | Undefined |
| WSLRU | 15:10 | LRU bits. This field contains the value read from the WS array after a CACHE Index Load WS operation. It is used to store into the WS array during CACHE Index Store WS operations. | R/W | Undefined |
| BasePA | 31:12 | When reading pseudo-tag 0 of a scratchpad RAM, this field will contain bits [31:12] of the base address of the scratchpad region | R/W | Undefined |
| E | 7 | When reading pseudo-tag 0 of a scratchpad RAM, this bit will indicate whether the scratchpad is enabled | R/W | Undefined |
| Size | 19:12 | When reading pseudo-tag 1 of a scratchpad RAM, this field indicates the size of the scratchpad array. This field is the number of 4KB sections it contains. (Combined with the 0's in 11:0, the register will contain the number of bytes in the scratchpad region) | R/W | Undefined |

In addition to the three uses of the *TagLo* register specified above, there is a fourth application where *TagLo* is used to access the pseudo-tags (control registers) of the ITC block. This is done by executing the Index Store Tag or Index Load Tag operation of the CACHE instruction with the $ErrCtl_{ITC}$ set to 1 (and $ErrCtl_{SPR}$/$ErrCtl_{WST}$ set to 0).

## 7.2.56 DataLo Register (CP0 Register 28, Select 1,3,5)

The *DataLo* register is a register that acts as the interface to the cache data array and is intended for diagnostic operations only. The Index Load Tag operation of the CACHE instruction reads the corresponding data values into the *DataLo* register. If the WST bit in the *ErrCtl* register is set, then the contents of *DataLo* can be written to the cache data array by doing an Index Store Data CACHE instruction. If the SPR bit in the *ErrCtl* register is set, then the contents of *DataLo* can be written to the scratchpad RAM data array by doing an Index Store Data CACHE instruction.

Note that there are separate *DataLo* registers for each of the caches (L1 I-cache: select 1, L1 D-cache: select 3, L2: select 5).

**Figure 7.55 DataLo Register Format**

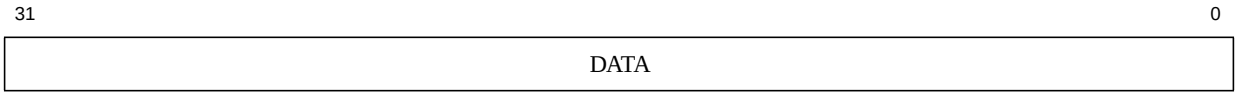



**Table 7.63 DataLo Register Field Description**

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

## 7.2.57 DataHi Register (CP0 Register 29, Select 1,5)

The *DataHi* register is a register that acts as the interface to the cache data array and is intended for diagnostic operations only. The Index Load Tag operation of the CACHE instruction reads the corresponding data values into the *DataHi* register. If the *WST* bit in the *ErrCtl* register is set, then the contents of *DataHi* can be written to the cache data array by doing an Index Store Data CACHE instruction. If the *SPR* bit in the *ErrCtl* register is set, then the contents of *DataHi* can be written to the scratchpad RAM data array by doing an Index Store Data CACHE instruction.

The *DataHi* register only exists for the primary Instruction Cache (select 1) and the secondary cache (select 5). The interface to the I-cache only operates on pairs of instructions - the high instruction will be written into the *DataHi* register.

Note that *DataHi* and *DataLo* reflect the memory ordering of the instructions. Depending on the endianness of the system, Instruction0 belongs in either *DataHi* (BigEndian) or *DataLo* (LittleEndian) and vice versa for Instruction1.

**Figure 7.56 DataHi Register Format**

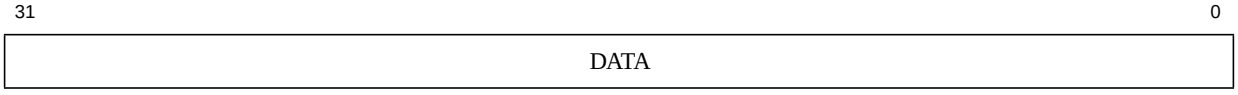



**Table 7.64 DataHi Register Field Description**

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

## 7.2.58 ErrorEPC (CP0 Register 30, Select 0)

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

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

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

On a reset exception, VPE0's ErrorEPC contains the virtual address at which TC0 would have resumed processing after servicing the error. This in conjunction with TCRestart registers of other TC's can provide valuable debug information about the state of the various TC's when the error occurred.

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

In processors that implement the MIPS16 ASE, a read of the *ErrorEPC* register (via MFC0) returns the following value in the destination GPR:

$$GPR[rt] \leftarrow ErrorExceptionPC_{31..1} \,||\, ISAMode_0$$

That is, the upper 31 bits of the error exception PC are combined with the lower bit of the ISAMode field and written to the GPR.

Similarly, a write to the *ErrorEPC* register (via MTC0) takes the value from the GPR and distributes that value to the error exception PC and the ISAMode field, as follows

$$ErrprExceptionPC \leftarrow GPR[rt]_{31..1} \,||\, 0$$
$$ISAMode \leftarrow 2\#0 \,||\, GPR[rt]_0$$

That is, the upper 31 bits of the GPR are written to the upper 31 bits of the error exception PC, and the lower bit of the error exception PC is cleared. The upper bit of the ISAMode field is cleared and the lower bit is loaded from the lower bit of the GPR.

**Figure 7.57 ErrorEPC Register Format**

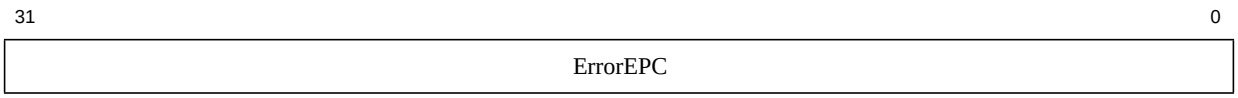



**Table 7.65 ErrorEPC Register Field Description**

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

## 7.2.59 DeSave Register (CP0 Register 31, Select 0)

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

**Figure 7.58 DeSave Register Format**


**Table 7.66 DeSave Register Field Description**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| DESAVE | 31:0 | Debug exception save contents. | R/W | Undefined |

*Chapter 8*

# Hardware and Software Initialization of the 34K™ Core

A 34K processor core contains only a minimal amount of hardware initialization and relies on software to fully initialize the device.

This chapter contains the following sections:

- Section 8.1 “Hardware-Initialized Processor State”
- Section 8.2 “Software Initialized Processor State”

## 8.1 Hardware-Initialized Processor State

A 34K processor core, like most other MIPS processors, is not fully initialized by hardware reset. Only a minimal subset of the processor state is cleared. This is enough to bring the core up while running in unmapped and uncached code space. All other processor state can then be initialized by software. Unlike previous MIPS processors, there is no distinction between cold and warm resets (or hard and soft resets). *SI_Reset* is used for both power-up reset and soft reset.

### 8.1.1 Coprocessor 0 State

Much of the hardware initialization occurs in Coprocessor 0.

- $MVPControl_{CPA}$ - cleared to 0 on Reset
- $MVPControl_{STLB}$ - cleared to 0 on Reset
- $MVPControl_{VPC}$ - cleared to 0 on Reset
- $MVPControl_{EVP}$ - cleared to 0 on Reset
- *Random* - cleared to maximum value on Reset (TLB MMU only)
- $VPEControl_{YSI}$ - cleared to 0 on Reset
- $VPEControl_{GSI}$ - cleared to 0 on Reset
- $VPEControl_{TE}$ - cleared to 0 on Reset
- $VPEConf0_{XTC}$ - cleared to 0 on Reset
- $VPEConf0_{MVP}$ - set to 1 for VPE0, cleared to 0 for other VPEs on Reset

- *VPEConf0*$_{VPA}$ - set to 1 for VPE0, cleared to 0 for other VPEs on Reset
- *YQMask*$_{Mask}$ - cleared to 0 on Reset
- *TCStatus*$_{TMX}$ - cleared to 0 on Reset
- *TCStatus*$_{DT}$ - cleared to 0 on Reset
- *TCStatus*$_{DA}$ - cleared to 0 on Reset
- *TCStatus*$_{A}$ - set to 1 for TC0, cleared to 0 for all other TCs on Reset
- *TCStatus*$_{IXMT}$ - cleared to 0 on Reset
- *TCBind*$_{TBE}$ - cleared to 0 on Reset
- *Wired* - cleared to 0 on Reset (TLB MMU only)
- *Status*$_{BEV}$ - set to 1 on Reset
- *Status*$_{TS}$ - cleared to 0 on Reset
- *Status*$_{NMI}$ - cleared to 0 on Reset
- *Status*$_{ERL}$ - set to 1 on Reset
- *Status*$_{RP}$ - cleared to 0 on Reset
- *WatchLo*$_{I,R,W}$ - cleared to 0 on Reset
- *Config* fields related to static inputs - set to input value by Reset
- *Config*$_{K0}$ - set to 010 (uncached) on Reset
- *Config*$_{KU}$ - set to 010 (uncached) on Reset (FM MMU only)
- *Config*$_{K23}$ - set to 010 (uncached) on Reset (FM MMU only)
- *Debug*$_{DM}$ - cleared to 0 on Reset (unless EJTAGBOOT option is used to boot into DebugMode, see Chapter 11, “EJTAG Debug Support in the 34K™ Core” for details)
- *Debug*$_{LSNM}$ - cleared to 0 on Reset
- *Debug*$_{IBusEP}$ - cleared to 0 on Reset
- *Debug*$_{DBusEP}$ - cleared to 0 on Reset
- *Debug*$_{IEXI}$ - cleared to 0 on Reset

- $Debug_{SSt}$ - cleared to 0 on Reset

### 8.1.2 TLB Initialization

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

### 8.1.3 Bus State Machines

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

### 8.1.4 Static Configuration Inputs

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

### 8.1.5 Fetch Address

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

## 8.2 Software Initialized Processor State

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

### 8.2.1 Register File

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

### 8.2.2 TLB

Because of the hidden bit indicating initialization, the core does not initialize the TLB upon Reset. This is an implementation specific feature of the 34K core and cannot be relied upon if writing generic code for MIPS32/64 processors.

### 8.2.3 Caches

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

### 8.2.4 Coprocessor 0 State

Miscellaneous COP0 states need to be initialized prior to leaving the boot code. There are various exceptions which are blocked by ERL=1 or EXL=1 and which are not cleared by Reset. These can be cleared to avoid taking spurious exceptions when leaving the boot code.

- *Cause*: WP (Watch Pending), SW0/1 (Software Interrupts) should be cleared.
- *Config*: K0 should be set to the desired Cache Coherency Algorithm (CCA) prior to accessing Kseg0.
- *Config*: (FM MMU only) KU and K23 should be set to the desired CCA for USeg/KUSeg and KSeg2/3 respectively prior to accessing those regions.
- *Count*: Should be set to a known value if Timer Interrupts are used.
- *Compare*: Should be set to a known value if Timer Interrupts are used. The write to compare will also clear any pending Timer Interrupts (Thus, *Count* should be set before *Compare* to avoid any unexpected interrupts).
- *Status*: Desired state of the device should be set.
- Other COP0 state: Other registers should be written before they are read. Some registers are not explicitly writeable, and are only updated as a by-product of instruction execution or a taken exception. Uninitialized bits should be masked off after reading these registers.

### 8.2.5 Multi-Threading Initialization

In order to start multi-threading on a 34K processor core, a few extra initialization steps are needed - mostly on CP0 registers. Please refer to Chapter 4, “Initializing the 34K™ core - Multi-Threaded bootstrap issues” in *Programming the MIPS32® 34K™ Core Family.*

*Chapter 9*

# Caches of the 34K™ Core

This chapter describes the caches present in a 34K processor core. It contains the following sections:



## 9.1 Cache Configurations

A 34K processor core has separate instruction and data caches which allows instruction and data references to proceed simultaneously. Each of the caches is 4-way set associative and they can be independently configured at build time to be 8, 16, 32, or 64KB. Both caches use a 32B line size and support locking on a per line basis. Parity protection of the cache arrays is an optional feature.

## 9.2 Instruction Cache

Table 9.1 shows the key characteristics of the instruction cache. Figure 9.1 shows the format of an entry in the three arrays comprising the instruction cache: tag, data, and way-select. Note that for 8KB caches, there is one extra tag bit needed.

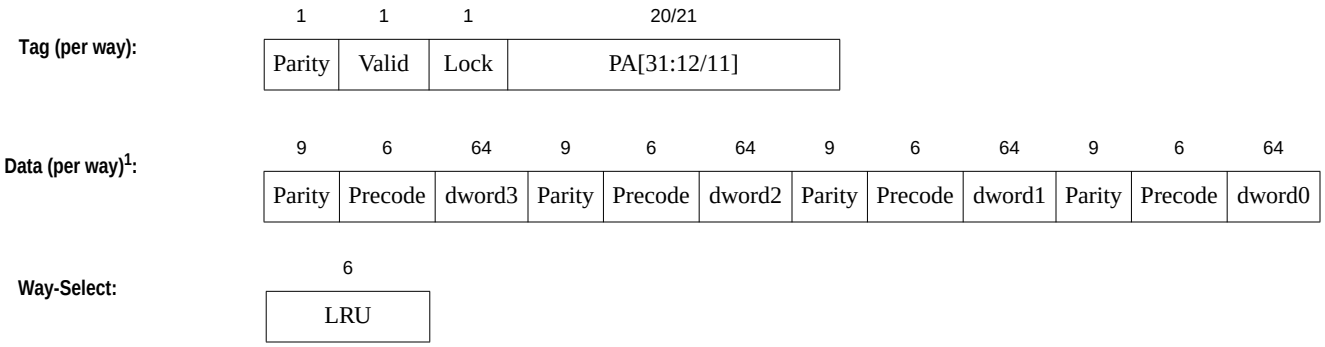
**Table 9.1 Instruction Cache Attributes**

| Attribute | With Parity | Without Parity |
|---|---|---|
| Size | 0, 8, 16, 32, 64KB | |
| Line Size | 32B | |
| Number of Cache Sets | 64,128,256,512 | |
| Associativity | 4 way | |
| Replacement | LRU | |
| Cache Locking | per line | |

**Table 9.1 Instruction Cache Attributes**

| Attribute | With Parity | Without Parity |
|---|---|---|
| Data Array | | |
| Read Unit | 79b x 4 | 70b x 4 |
| Write Unit | 79b | 70b |
| Tag Array | | |
| Read Unit (8KB) | 24b x 4 | 23b x 4 |
| Read Unit (non-8KB) | 23b x 4 | 22b x 4 |
| Write Unit (8KB) | 24b | 23b |
| Write Unit (non-8KB) | 23b | 22b |
| Way-Select Array | | |
| Read Unit | 6b | |
| Write Unit | 1-6b | |

**Figure 9.1 Instruction Cache Organization**




1. Parity Bits in data array will be interleaved with precode and data bytes

## 9.2.1 Virtual Aliasing

The instruction cache on the 34K processor core is virtually indexed and physically tagged. The lower bits of the virtual address are used to access the cache arrays and the physical address is used in the tags. Because the way size can be larger than the minimum TLB page size, there is a potential for virtual aliasing. This means that one physical address can exist in multiple indices within the cache if it is accessed with different virtual addresses.

This reduces the cache efficiency somewhat, but is generally not a problem unless the instruction stream is being written to. When instructions are written, software must ensure that the store data is written out to memory and the old data is invalidated in the instruction cache (via the CACHE or SYNCI instruction). Because one physical address can exist in multiple locations, the cache should be invalidated using all of the virtual addresses used to access that physical address. Alternatively, all of the relevant cache indices or the entire cache can be invalidated.

To avoid virtual aliasing in the Data Cache, the core can be configured to use physical addresses to access the cache.

### 9.2.2 Precode bits

In order for the fetch unit to quickly detect branches and jumps when executing code, the instruction cache array contains some additional precode bits. These bits indicate the type and location of branch or jump instructions within a 64b fetch bundle. These precode bits are not used when executing MIPS16e code.

### 9.2.3 Parity

Parity protection of the instruction cache arrays can optionally be included. The data array has a 9 parity bits - one for the 6 precode bits and one for each byte of the 64b data. The tag array has a single parity bit for each tag. The LRU array does not have any parity.

## 9.3 Data Cache

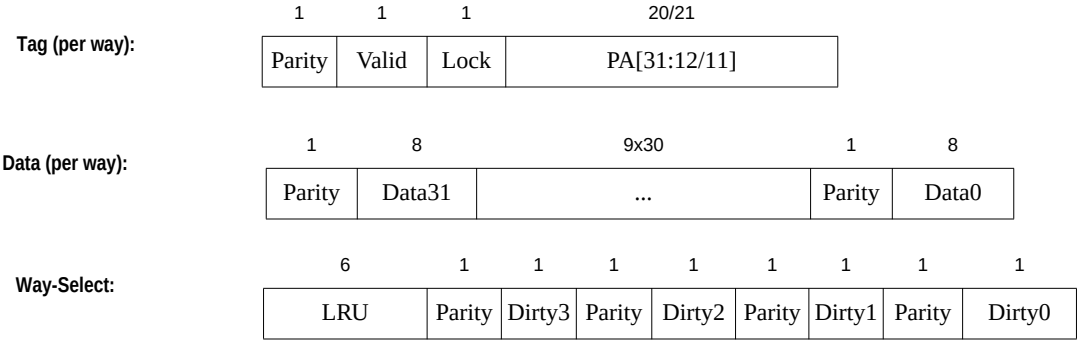
The data cache is similar to the instruction cache, with a few key differences. The data cache does not contain any precode information. To handle store bytes, the data array is byte accessible and the optional data parity is 1 bit per byte. Additionally, the way-select array for the data cache also holds the dirty bits (and optional dirty parity bits) for each cache line, in addition to the LRU information.

Table 9.2 shows the key characteristics of the data cache. Figure 9.2 shows the format of an entry in the three arrays comprising the data cache: tag, data, and way-select.

**Table 9.2 Data Cache Attributes**

| Attribute | With Parity | Without Parity |
|---|---|---|
| Size | 0, 8, 16, 32, 64KB | |
| Line Size | 32B | |
| Number of Cache Sets | 64,128,256,512 | |
| Associativity | 4 way | |
| Replacement | LRU | |
| Cache Locking | per line | |
| Data Array | | |
| Read Unit | 72b x 4 | 64b x 4 |
| Write Unit | 9b | 8b |
| Tag Array | | |
| Read Unit (8KB) | 24b x 4 | 23b x 4 |
| Read Unit (non-8KB) | 23b x 4 | 22b x 4 |
| Write Unit (8KB) | 24b | 23b |
| Write Unit (non-8KB) | 23b | 22b |
| Way-Select Array | | |
| Read Unit | 14b | 10b |
| Write Unit | 1-14b | |

**Figure 9.2 Data Cache Organization**




### 9.3.1 Parity

Parity protection of the data cache arrays can optionally be included. The data array requires a parity bit for each byte, to correspond to the minimum write quantum for a store. The tag array has a single parity bit for each tag. The way-select array has separate parity bits to cover each dirty bit, but the LRU bits are not covered by parity.

## 9.4 Uncached Accelerated Stores

Uncached Accelerated gathering is supported for word and double word stores only.

Gathering of uncached accelerated stores will start on cache-line aligned addresses, i.e. 32 byte aligned addresses. Uncached accelerated word or double word stores that do not to meet the conditions required to start gathering will be treated like regular uncached stores.

Once an uncached accelerated store meets the requirements needed to start gathering a gather buffer is reserved for this store. All subsequent uncached accelerated word or double word stores to the same cache line will write sequentially into this buffer, independent of the word address associated with these stores. The uncached accelerated buffer is tagged with the address of the first store.

An uncached accelerated buffer is written to memory (flushed) if:

1. The last word in the entry being gathered is written. (Implicit flush)
2. A PREF Nudge which match the address associated with the gather buffer (Explicit flush).
3. A SYNC instruction is executed. (Explicit flush)
4. Bits <31:5> of the address of a Load instruction match the address associated with the gather buffer. (Implicit flush)
5. Uncached Accelerated store to a different 32B line (Implicit flush)
6. An exception occurs. (Implicit flush)

When an uncached accelerated buffer is flushed, the address sent out on the system interface is the address associated with the gather buffer.

Caveats:

- Any uncached stores and any uncached loads to unrelated addresses that occur between uncached accelerated stores that are part of a gather sequence will go out of order. They will not enforce ordering.
- The only constraint imposed on the gathering is that doubleword stores are only allowed to write to double word aligned locations in the buffer. For example if uncached accelerated gathering starts with a Store Word (SW), it may not be followed by a Store Double (SDC1).
- Uncached accelerated stores of the following types are not intended to be used by software and may generate unpredictable results:
  1. Half word Stores
  2. Unaligned Stores
  3. Store conditionals
- In order for software to be able to run functionally correct on implementations without uncached accelerated stores, software should always generate accesses starting on a cache-line aligned address, proceed to generate correctly incremented sequential addresses and observe the restrictions for uncached accelerated stores.

## 9.5 Cache Protocols

This section describes cache organization, attributes, and cache-line replacement for the instruction and data caches. This section also discusses issues relating to virtual aliasing.

### 9.5.1 Cache Organization

The instruction and data caches each consist of three arrays: tag, data and way-select. The caches are virtually indexed, since a virtual address is used to select the appropriate line within each of the three arrays. The caches are physically tagged, as the tag array contains a physical, not virtual, address.

The tag and data arrays hold 4 ways of information per set, corresponding to the 4-way set associativity of the cache. The way-select array holds information to choose the way to be filled, as well as dirty bits in the case of the data cache.

Figure 9.1 (instruction cache) and Figure 9.2 (data cache) show the format of each line in the tag, data and way-select arrays.

A tag entry consists of the upper 20 or 21 bits of the physical address (bits [31:12/11]) (bit 11 is only used for 8KB caches), one valid bit for the line, and a lock bit. A data entry contains the four 64-bit doublewords in the line, for a total of 32 bytes. All four words in the line are present or not in the data array together, hence the single valid bit stored with the tag. Once a valid line is resident in the cache, byte, halfword, triple-byte or full word stores can update all or a portion of the words in that line. The tag and data entries are repeated for each of the 4 lines in the set.

A way-select entry holds bits choosing the way to be replaced according to a Least Recently Used (LRU) algorithm. The LRU information applies to all the ways and there is one way-select entry for all the ways in the set. The array with way-select entries for the data cache also holds dirty bits for the lines. One dirty bit is required per line, as shown in Figure 9.2. The instruction cache only supports reads, hence only LRU entries are stored in the instruction way-select array.

### 9.5.2 Cacheability Attributes

A 34K core supports the following cacheability attributes:

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

- *Write-back with write allocation*: Loads and instruction fetches first search the cache, reading main memory only if the desired data does not reside in the cache. On data store operations, the cache is first searched to see if the target address is cache resident. If it is resident, the cache contents are updated, but main memory is not written. If the cache lookup misses on a store, main memory is read to bring the line into the cache and merge it with the new store data. Hence, the allocation policy on a cache miss is read- or write-allocate. Data stores will update the appropriate dirty bit in the way-select array to indicate that the line contains modified data. When a line with dirty data is displaced from the cache, it is written back to memory.

- *Write-through with no write allocation*: Loads and instruction fetches first search the cache, reading main memory only if the desired data does not reside in the cache. On data store operations, the cache is first searched to see if the target address is cache resident. If it is resident, the cache contents are updated, and main memory is also written. If the cache lookup misses on a store, only main memory is written. Hence, the allocation policy on a cache miss is read-allocate only.

- *Uncached Accelerated: Uncached stores are gathered together for more efficient bus utilization*. See Section 9.4 “Uncached Accelerated Stores” for more details

Some segments of memory employ a fixed caching policy; for example kseg1 is always uncacheable. Other segments of memory allow the caching policy to be selected by software. Generally, the cache policy for these programmable regions is defined by a cacheability attribute field associated with that region of memory. See Chapter 5, “Memory Management of the 34K™ Core” for further details.

### 9.5.3 Replacement Policy

The replacement policy refers to how a way is chosen to hold an incoming cache line on a miss which will result in a cache fill. The replacement policy is least recently used (LRU), but excluding any locked ways. The LRU bit(s) in the way-select array encode the order in which ways on that line have been accessed.

On a cache miss, the lock and LRU bits for the tag and way-select entries of the selected line may be used to determine the way which will be chosen.

The LRU field in the way select array is updated as follows:

- On a cache hit, the associated way is updated to be the most recently used. The order of the other ways relative to each another is unchanged.

- On a cache refill, the filled way is updated to be the most recently used.

- On CACHE instructions, the update of the LRU bits depends on the type of operation to be performed:

    - **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 Table 9.4 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:** For instruction cache, no update. For data cache, most-recently used.

If all ways are valid, then any locked ways will be excluded from consideration for replacement. For the unlocked ways, the LRU bits are used to identify the way which has been used least recently, and that way is selected for replacement.

The 34K core does not support the locking of all 4 ways of either cache at a particular index. If all 4 ways of the cache at a given index are locked by either Fetch and Lock or Index Store Tag CACHE instructions, subsequent cache misses at that cache index will displace one of the locked lines.

If the way selected for replacement has its dirty bit asserted in the way-select array, then that 32-byte line will be written back to memory before the new fill can occur.

### 9.5.4 Virtual Aliasing

Since the caches are virtually indexed and physically tagged, a potential issue referred to as *virtual aliasing* might exist. Virtual aliasing occurs if the virtual bits used to index a cache array are not consistent with the overlapping physical bits, after the virtual address has been translated to a physical address. The possibility of virtual aliasing only occurs in address regions which are mapped through a TLB-based memory management unit.

In TLB-mapped address regions, virtual aliasing may occur if the cache size per way is greater than the page size. For example, consider a 32KB cache organized as 4-way set associative. The size per way is then 8 KB, so virtual address bits [12:0] are used to index the array. If the address is in a translated region with a page size of 4 KB, then address bits [11:0] are untranslated but address bits [31:12] will be mapped and for these bits the virtual and physical addresses may be different. In this example, bit [12] could pose a potential problem due to virtual aliasing. Imagine two virtual addresses, VA0 and VA1, whose only difference is the value of bit [12], which map to the same physical address. These two virtual addresses would be indexed to two different lines by the cache, even though they were intended to represent the same physical address. Then if a program does a load using VA0 and a store using VA1, or vice-versa, the cache may not return the expected data.

Table 9.3 shows the overlapped virtual/physical address bits which could potentially be involved in virtual aliasing, given the possible minimum page sizes and cache way sizes supported by a 34K core. Virtual aliasing is generally only a problem for the D-cache, since stores don’t happen to the I-cache. A special hardware mechanism is available to prevent the possibility of virtual aliasing in 32KB and 64KB data caches. In cores not configured with this mechanism, virtual aliasing must be handled by software. The software solution must ensure that the mapping of virtual

address bits which overlap with physical address bits be handled consistently. The simplest approach is to ensure that the overlapping bits are unity-mapped (VA equals PA).

**Table 9.3 Potential Virtual Aliasing Bits**

| Minimum Page Size (KB) | Cache Way Size (KB) | Overlapped address bits with possible aliasing |
|---|---|---|
| 4 | 8 | [12] |
| | 16 | [13:12] |
| 8 | 16 | [13] |

A related issue can occur in virtually indexed, physically tagged caches if the number of physical bits stored in the tag array does not fully overlap the physically translated bits for the smallest page size. For a 34K core, there are always at least 20 address bits stored in the cache tag, representing bits [31:12] of the physical address. Since the minimum page size is 4KB with bits [31:12] physically translated by the TLB, the cache tag size does overlap the translated bits and this issue will not occur.

## 9.6 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. These instructions are described in detail in Chapter 15, "34K™ Processor Core Instructions".

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 7.2.53 "ErrCtl Register (CP0 Register 26, Select 0)".) Similarly, the *SPR* bit in the *ErrCtl* register will cause Index Load Tag and Index Store Tag instructions to access the pseudo-tags associated with the scratchpad RAM array. Finally, the *ITC* bit in the *ErrCtl* register will cause Index Load Tag and Index Store Tag instructions to access the pseudo-tags associated with the ITC block. Note that when the *WST, ITC,* and *SPR* bits are 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 9.4.

**Table 9.4 Way Selection Encoding, 4 Ways**

| Selection Order[1] | 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 |

**Table 9.4 Way Selection Encoding, 4 Ways (Continued)**

| Selection Order[1] | WS[5:0] | Selection Order | WS[5:0] |
|---|---|---|---|
| 1302 | 001101 | 3201 | 111011 |
| 1320 | 101101 | 3210 | 111111 |

1. The order is indicated by listing the least-recently used way to the left and the most-recently used way to the right, etc.

# 9.7 Software Cache Testing

Typically, the cache RAM arrays will be tested using BIST. It is, however, possible for software running on the processor to test all of the arrays. Of course, testing of the I-cache arrays should be done from an uncacheable space with interrupts disabled in order to maintain the cache contents. There are multiple methods for testing these arrays in software, only one is presented here.

## 9.7.1 I-Cache/D-cache Tag Arrays

These arrays can be tested via the Index Load Tag and Index Store Tag varieties of the CACHE instruction. Index Store Tag will write the contents of the *TagLo* register into the selected tag entry. Index Load Tag will read the selected tag entry into the *TagLo*.

If parity is implemented, the parity bits can be tested as a normal bit by setting the PO bit in the ErrCtl register. This will override the parity calculation and write P bit in TagLo as the parity value.

## 9.7.2 I-Cache Data Array

This array can be tested using the Index Store Data and Index Load Tag varieties of the CACHE instruction. The Index Store Data variety is enabled by setting the WST bit in the ErrCtl register.

The precode bits in the array can be tested by setting the PCO bit in the ErrCtl register. This will write the PCI field in the ErrCtl register instead of calculating the precode bits on a write.

The parity bits in the array can be tested by setting the PO bit in the ErrCtl register. This will use the PI field in ErrCtl instead of calculating the parity on a write.

The rest of the data bits are read/written to/from the DataLo and DataHi registers.

## 9.7.3 I-Cache WS Array

The testing of this array is done with via Index Load Tag and Index Store Tag CACHE instructions. By setting the WST bit in the ErrCtl register, these operations will read and write the WS array instead of the tag array.

## 9.7.4 D-Cache Data Array

This array can be tested using the Index Store Tag CACHE, SW, and LW instructions. First, use Index Store Tag to set the initial state of the tags to valid with a known physical address (PA). Write the array using SW instructions to the PAs that are resident in the cache. The value can then be read using LW instructions and compared to the expected data.

The parity bits can be implicitly tested using this mechanism. The parity bits can be explicitly tested by setting the PO bit in ErrCtl and using Index Store Data and Index Load Tag CACHE operations. The parity bits (one bit per byte) are read/written to/from the PD field in ErrCtl. Unlike the I-cache, the DataHi register is not used and only 32b of data is read/written per operation.

### 9.7.5 D-cache WS Array

The dirty bits in this array will be tested when the data tag is tested. The LRU bits can be tested using the same mechanism as the I-cache WS array.

## 9.8 Memory Coherence Issues

A cache presents coherency issues within the memory hierarchy which must be considered in the system design. Since a cache holds a copy of memory data, it is possible for another memory master to modify a memory location, thus making other copies of that location stale if those copies are still in use. A detailed discussion of memory coherence is beyond the scope of this document, but following are a few related comments.

A34K processor contains no direct hardware support for managing coherency with respect to its caches, so it must be handled via system design or software. The data cache supports either write-back or write-through protocols.

In write-through mode, all data writes will eventually be sent to memory. Due to write buffers, however, there could be a delay in how long it takes for the write to memory to actually occur. If another memory master updates cacheable memory which could also be in the cores caches, then those locations may need to be flushed from the cache. The only way to accomplish this invalidation is by use of the CACHE instruction.

In write-back mode, data writes only go to the cache and not to memory. So the processor cache may contain the *only* copy of data in the system until that data is written to main memory. Dirty lines are only written to memory when displaced from the cache as a new line is filled or if explicitly forced by certain flavors of the CACHE or PREF instructions.

The SYNC instruction may also be useful to software enforcing memory coherence, as it flushes the core's write buffers.

*Chapter 10*

# Power Management in the 34K™ Core

A 34K processor core offers a number of power management features, including low-power design, active power management and power-down modes of operation. The core is a static design that supports a WAIT instruction designed to signal the rest of the device that execution and clocking should be halted, reducing system power consumption during idle periods.

The core provides threemechanisms for system level low-power support discussed in the following sections.

- Section 10.1 "Register-Controlled Power Management"
- Section 10.2 "Instruction-Controlled Power Management"
- Section 10.3 "Thread-Controlled Power Management"

## 10.1 Register-Controlled Power Management

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

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

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

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

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

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

- The *SI_RP* signal represents the state of the RP bit (27) in the CP0 *Status* register.
- The *SI_EXL* signal represents the state of the EXL bit (1) in the CP0 *Status* register.

- The *SI_ERL* signal represents the state of the ERL bit (2) in the CP0 *Status* register.
- The *EJ_DebugM* signal indicates that the processor has entered debug mode.

## 10.2 Instruction-Controlled Power Management

A second mechanism for invoking power down mode within a single threaded execution path is through execution of the WAIT instruction. The WAIT instruction brings the processor into a low power state where the internal clocks are suspended and the pipeline is frozen. However, the internal timer and some of the input pins (*SI_Int*[5:0], *SI_NMI*, *SI_Reset*, and *EJ_DINT*) continue to run. The clocks are not shut down until all bus and coprocessor transactions have completed. Once the CPU is in instruction controlled power management mode, any enabled interrupt, NMI, debug interrupt, or reset condition causes the CPU to exit this mode and resume normal operation. While the core is in this low-power mode, the *SI_SLEEP* signal is asserted to indicate to external agents what the state of the chip is.

Within multi-threading, WAIT execution regulates the low power behavior per hardware thread, deactivating execution logic for this waiting thread. The simultaneous execution of WAIT for all hardware threads leads to a complete low power state as described above for the single-threaded mode.

## 10.3 Thread-Controlled Power Management

Third, the execution of a yield instruction and its successive thread suspend will lead to low power consumption of the yielding thread. Simultaneous yielding within all hardware threads causes low power conditions equivalent to WAIT execution - as described in Section 10.2 “Instruction-Controlled Power Management”

*Chapter 11*

# EJTAG Debug Support in the 34K™ Core

The EJTAG debug logic in the 34K processor core is compliant with EJTAG Specification 3.20 and includes:

1. Standard core debug features
2. Optional hardware breakpoints
3. Standard Test Access Port (TAP) for a dedicated connection to a debug host
4. Optional MIPS Trace capability for program counter/data address/data value trace to On-chip memory or to Trace probe

This chapter contains the following sections:

## 11.1 Debug Control Register

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

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

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

The SRE bit allows implementation dependent masking of some sources for reset. The 34K core does not distinguish between soft and hard reset, but typically only soft reset sources in the system would be maskable and hard sources such as the reset switch would not be. The soft reset masking should only be applied to a soft reset source if that source can be efficiently masked in the system, thus resulting in no reset at all. If that is not possible, then that soft reset source should not be masked, since a partial soft reset may cause the system to fail or hang. There is no automatic indication of whether the SRE is effective, so the user must consult system documentation.

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

**Figure 11.1 Debug Control Register**

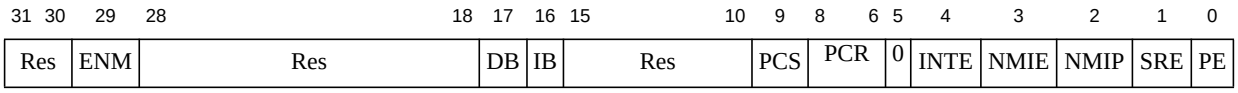



**Table 11.1 Debug Control Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| Res | 31:30 | Reserved | R | 0 |
| ENM | 29 | Endianess in Kernel and Debug mode<br>0: Little Endian<br>1: Big Endian | R | Preset |
| Res | 28:18 | Reserved | R | 0 |
| DB | 17 | Data Break Implemented<br>0: No Data Break feature implemented<br>1: Data Break feature is implemented | R | Preset |
| IB | 16 | Instruction Break Implemented<br>0: No Instruction Break feature implemented<br>1: Instruction Break feature is implemented | R | Preset |
| Res | 15:10 | Reserved | R | 0 |
| PCS | 9 | Program Counter Sampling Implemented<br>0: No PC Sampling implemented<br>1: PC Sampling implemented | R | 1 |

**Table 11.1 Debug Control Register Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| PCR | 8:6 | PC Sampling Rate: Values from 0 to 7 map to $2^5$ to $2^{12}$ respectively. That is, a PC sample is written out every 32, 64, 128, 256, 512, 1024, 2048, or 4096 cycles. The external probe or software is allowed to set this value to the desired sample rate | R/W | 7 |
| INTE | 4 | Interrupt Enable in Normal Mode. This bit provides the hardware and software interrupt enable for non-debug mode, in addition to other masking mechanisms:<br>0: Interrupts disabled.<br>1: Interrupts enabled (depending on other enabling mechanisms). | R/W | 1 |
| NMIE | 3 | Non-Maskable Interrupt Enable for non-debug mode<br>0: NMI disabled.<br>1: NMI enabled. | R/W | 1 |
| NMIP | 2 | NMI Pending Indication.<br>0: No NMI pending.<br>1: NMI pending. | R | 0 |
| SRE | 1 | Soft Reset Enable<br>This bit allows the system to mask soft resets. The core does not internally mask resets. Rather the state of this bit appears on the *EJ_SRstE* external output signal, allowing the system to mask soft resets if desired. | R/W | 1 |
| PE | 0 | Probe Enable<br>This bit reflects the ProbEn bit in the EJTAG Control register.<br>0: No accesses to dmseg allowed<br>1: EJTAG probe services accesses to dmseg | R | Same value as ProbEn in ECR (see Table 11.25) |

## 11.2 Hardware Breakpoints

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

There are two types of simple hardware breakpoints implemented in the 34K core; Instruction breakpoints and Data breakpoints.

A core may be configured with the following breakpoint options:

- Zero or four instruction breakpoints
- Zero or two data breakpoints

### 11.2.1 Features of Instruction Breakpoint

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

Instruction breakpoints compare the virtual address of the executed instructions (PC) and the ASID with the registers for each instruction breakpoint including masking of address and ASID. When an instruction breakpoint matches, a debug exception and/or a trigger is generated. An internal bit in the instruction breakpoint registers is set to indicate that the match occurred.

### 11.2.2 Features of Data Breakpoint

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

Data breakpoints compare the transaction type (TYPE), which may be load or store, the virtual address of the transaction (ADDR), the ASID, accessed bytes (BYTELANE) and data value (DATA), with the registers for each data breakpoint including masking or qualification on the transaction properties. When a data breakpoint matches, a debug exception and/or a trigger is generated, and an internal bit in the data breakpoint registers is set to indicate that the match occurred. The match is precise in that the debug exception or trigger occurs on the instruction that caused the breakpoint to match.

### 11.2.3 Instruction Breakpoint Registers Overview

The register with implementation indication and status for instruction breakpoints in general is shown in Table 11.2.

**Table 11.2 Overview of Status Register for Instruction Breakpoints**

| Register Mnemonic | Register Name and Description |
|---|---|
| *IBS* | Instruction Breakpoint Status |

The four instruction breakpoints are numbered 0 to 3 for registers and breakpoints, and the number is indicated by n. The registers for each breakpoint are shown in Table 11.3.

**Table 11.3 Overview of Registers for Each Instruction Breakpoint**

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

### 11.2.4 Data Breakpoint Registers Overview

The register with implementation indication and status for data breakpoints in general is shown in Table 11.4.

**Table 11.4 Overview of Status Register for Data Breakpoints**

| Register Mnemonic | Register Name and Description |
|---|---|
| *DBS* | Data Breakpoint Status |

The two data breakpoints are numbered 0 and 1 for registers and breakpoints, and the number is indicated by n. The registers for each breakpoint are shown in Table 11.5.

**Table 11.5 Overview of Registers for Each Data Breakpoint**

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

### 11.2.5 Conditions for Matching Breakpoints

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

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

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

#### 11.2.5.1 Conditions for Matching Instruction Breakpoints

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

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

```
IB_match =
          ( ! IBCn_TCuse || ( TC == IBC_TC ) ) &&
          ( ! IBCn_ASIDuse || ( ASID == IBASIDn_ASID ) ) &&
          ( <all 1's> == ( IBMn_IBM | ~ ( PC ^ IBAn_IBA ) &&
          ( (IBMn_ISAM | ~(ISAMode ^ IBAn_ISA))) )
```

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

#### 11.2.5.2 Conditions for Matching Data Breakpoints

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

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

The overall match equation is the DB_match.

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

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

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

The size of $DBCn_{BAI}$ and BYTELANE is 8 bits. They are 8 bits to allow for data value matching on doubleword floating point loads and stores. For non-doubleword loads and stores, only the lower 4 bits will be used.

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

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

The size of $DBCn_{BLM}$, $DBCn_{BAI}$ and BYTELANE is 8 bits.

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

```
DB_value_match =
    ( ( 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[23:16] == DBVn_DBV[23:16] ) || !BYTELANE[2] || DBCn_BLM[2] || DBCn_BAI[2] )&&
```

```
( ( DATA[31:24] == DBVn[DBV[31:24]] ) || !BYTELANE[3] || DBCn[BLM[3]] || DBCn[BAI[3]] )&&
( ( DATA[39:32] == DBVn[DBV[39:32]] ) || !BYTELANE[4] || DBCn[BLM[4]] || DBCn[BAI[4]] )&&
( ( DATA[47:40] == DBVn[DBV[47:40]] ) || !BYTELANE[5] || DBCn[BLM[5]] || DBCn[BAI[5]] )&&
( ( DATA[55:48] == DBVn[DBV[55:48]] ) || !BYTELANE[6] || DBCn[BLM[6]] || DBCn[BAI[6]] )&&
( ( DATA[63:56] == DBVn[DBV[63:56]] ) || !BYTELANE[7] || DBCn[BLM[7]] || DBCn[BAI[7]] ))
```

The match for a data breakpoint without value compare is always precise, since the match expression is fully evaluated at the time the load/store instruction is executed. A true DB_match can thereby be indicated on the very same instruction causing the DB_match to be true. The match for data breakpoints with value compare is always imprecise.

## 11.2.6 Debug Exceptions from Breakpoints

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

### 11.2.6.1 Debug Exception by Instruction Breakpoint

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

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

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

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

### 11.2.6.2 Debug Exception by Data Breakpoint

If the breakpoint is enabled by BE bit in the *DBCn* register, then a debug exception occurs when the DB_match condition is true. The corresponding BS[n] bit in the *DBS* register is set when the breakpoint generates the debug exception. A matching data breakpoint generates either a precise or imprecise debug exception

#### *Debug Data Break Load/Store Exception as a Precise Debug Exception*

A precise debug data break exception occurs when a data breakpoint without value compare indicates a match. In this case the *DEPC* register and DBD bit in the *Debug* register points to the instruction that caused the DB_match equation to be true.

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

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

The result of this is that the load or store instruction causing the debug data break exception appears as not executed.

If both data breakpoints without and with data value compare would match the same transaction and generate a debug exception, then the rules shown in Table 11.6 apply with respect to updating the BS[n] bits.

**Table 11.6 Rules for Update of BS Bits on Data Breakpoint Exceptions**

| 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 break-points) |
| 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 break-points) | BS bits set for all |
| Store | One or more | One or more | BS bits set for all | BS bits set for all |
| Store | None | One or more | (No matching break-points) | BS bits set for all |

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

The debug handler usually returns to the instruction causing the debug data break exception, whereby the instruction is re-executed. Debug software is responsible for disabling breakpoints when returning to the instruction, otherwise the debug data break exception will reoccur.

### *Debug Data Break Load/Store Exception as a Imprecise Debug Exception*

An Debug Data Break Load/Store Imprecise exception occurs when a data breakpoint indicates an imprecise match. Imprecise matches are generated when data value compare is used. 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/store instruction that caused the DB_match equation to be true.

The load/store instruction causing the Debug Data Break Load/Store Imprecise exception always updates the destination register and completes the access to the external memory system. Therefore this load/store instruction is not re-executed on return from the debug handler, because the DEPC register and DBD bit do not point to that instruction.

Several imprecise data breakpoints can be pending at a given time, if the bus system supports multiple outstanding data accesses. The breakpoints are evaluated as the accesses finalize, and a Debug Data Break Load/Store Imprecise exception is generated only for the first one matching. Both the first and succeeding matches cause corresponding BS bits and DDBLImpr/DDBSImpr to be set, but no debug exception is generated for succeeding matches because the processor is already in Debug Mode. Similarly, if a debug exception had already occurred at the time of the first match (for example, due to a precise debug exception), then all matches cause the corresponding BS bits and DDBLImpr/DDBSImpr to be set, but no debug exception is generated because the processor is already in Debug Mode.

The SYNC instruction, followed by appropriate spacing must be executed before the BS bits and DDBLImpr/DDBSImpr bits are accessed for read or write. This delay ensures that these bits are fully updated.

Any BS bit set prior to the match and debug exception are kept set, because only debug software can clear the BS bits.

## 11.2.7 Breakpoint used as TriggerPoint

Both instruction and data hardware breakpoints can be setup by software so a matching breakpoint does not generate a debug exception, but only an indication through the BS[n] bit. The TE bit in the *IBCn* or *DBCn* register controls if an instruction or data breakpoint is used as a so-called triggerpoint. The triggerpoints are, like breakpoints, only compared for instructions executed in non-debug mode.

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

The triggerpoint feature can be used to start and stop tracing. See Section 11.10 "Enabling MIPS Trace" for details.

## 11.2.8 Instruction Breakpoint Registers

The registers for instruction breakpoints are described below. These registers have implementation information and are used to set up the instruction breakpoints. All registers are in drseg, and the addresses are shown in Table 11.7.

**Table 11.7 Addresses for Instruction Breakpoint Registers**

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

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

### 11.2.8.1 Instruction Breakpoint Status (IBS) Register

**Compliance Level:** Implemented only if instruction breakpoints are implemented.

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

The ASID applies to all the instruction breakpoints.

**Figure 11.2 IBS Register Format**

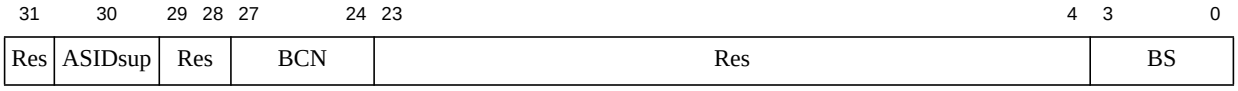


**Table 11.8 IBS Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| Res | 31 | Must be written as zero; returns zero on read. | R | 0 |
| ASIDsup | 30 | Indicates that ASID compare is supported in instruction breakpoints.<br>0: No ASID compare.<br>1: ASID compare (IBASIDn register implemented).<br><br>1: Supported<br>0: Not supported | R | Fixed MMU - 0<br>TLB - 1 |
| Res | 29:28 | Must be written as zero; returns zero on read. | R | 0 |
| BCN | 27:24 | Number of instruction breakpoints implemented. | R | 4 |
| Res | 23:4 | Must be written as zero; returns zero on read. | R | 0 |
| BS | 3:0 | Break status for breakpoint n is at BS[n], with n from 0 to 3. The bit is set to 1 when the condition for the corresponding breakpoint has matched. | R/W | Undefined |

### 11.2.8.2 Instruction Breakpoint Address n (IBAn) Register

**Compliance Level:** Implemented only for implemented instruction breakpoints.

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

**Figure 11.3 IBAn Register Format**

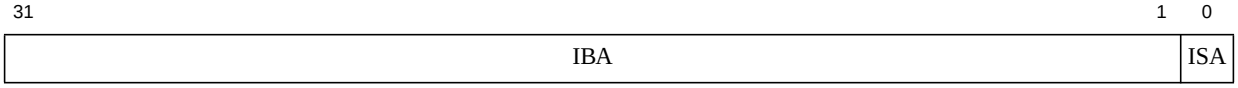



**Table 11.9 IBAn Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| IBA | 31:1 | Instruction breakpoint address for condition. | R/W | Undefined |
| ISA | 0 | Instruction breakpoint ISA mode for condition | R/W | Undefined |

### 11.2.8.3 Instruction Breakpoint Address Mask n (IBMn) Register

**Compliance Level:** Implemented only for implemented instruction breakpoints.

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

**Figure 11.4 IBMn Register Format**

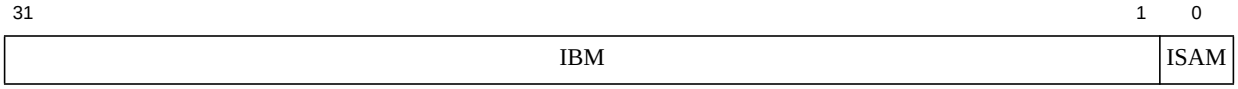



**Table 11.10 IBMn Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| IBM | 31:1 | Instruction breakpoint address mask for condition:<br>0: Corresponding address bit not masked.<br>1: Corresponding address bit masked. | R/W | Undefined |
| ISAM | 0 | Instruction breakpoint ISA mode mask for condition:<br>0: ISA mode considered for match condition<br>1: ISA mode masked | R/W | Undefined |

### 11.2.8.4 Instruction Breakpoint ASID n (IBASIDn) Register

**Compliance Level:** Implemented only for implemented instruction breakpoints.

For processors with a TLB based MMU, this register is used to define an ASID value to be used in the match expression. For cores with a FM MMU, this register is reserved and reads as 0.

**Figure 11.5 IBASIDn Register Format**

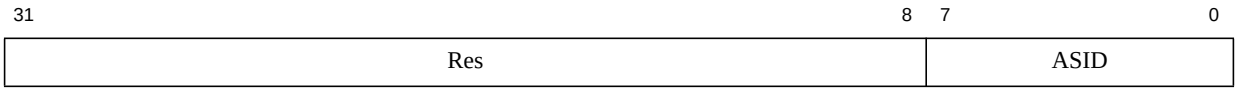



**Table 11.11 IBASIDn Register Field Descriptions**

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

### 11.2.8.5 Instruction Breakpoint Control n (IBCn) Register

**Compliance Level:** Implemented only for implemented instruction breakpoints.

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

**Figure 11.6 IBCn Register Format**

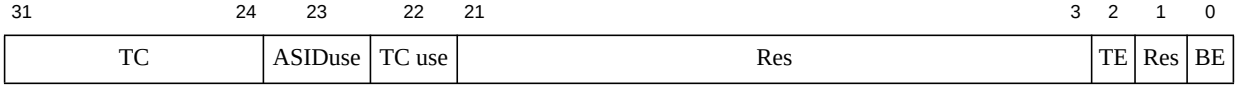


**Table 11.12 IBCn Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| TC | 31:24 | The value of TC (thread context) to match in the comparison to determine if the instruction break is to be taken. TC value is ignored if TCuse is set to 0. | R/W | Undefined |
| ASIDuse | 23 | Use ASID value in compare for instruction breakpoint n:<br>0: Don't use ASID value in compare<br>1: Use ASID value in compare | R/W | Undefined |
| TCuse | 22 | Use TC value in comparison for instruction breakpoint<br>0: Do not use value in compare<br>1: Use TC value in compare. | R/W | Undefined |
| Res | 21:3 | Must be written as zero; returns zero on read. | R | 0 |
| TE | 2 | Use instruction breakpoint n as triggerpoint:<br>0: Don't use it as triggerpoint<br>1: Use it as triggerpoint | R/W | 0 |
| Res | 1 | Must be written as zero; returns zero on read. | R | 0 |
| BE | 0 | Use instruction breakpoint n as breakpoint:<br>0: Don't use it as breakpoint<br>1: Use it as breakpoint | R/W | 0 |

## 11.2.9 Data Breakpoint Registers

The registers for data breakpoints are described below. These registers have implementation information and are used the setup the data breakpoints. All registers are in drseg, and the addresses are shown in Table 11.13.

**Table 11.13 Addresses for Data Breakpoint Registers**

| Offset in drseg | Register Mnemonic | Register Name and Description |
|---|---|---|
| 0x2000 | *DBS* | Data Breakpoint Status |
| 0x2100 + 0x100 * n | *DBAn* | Data Breakpoint Address n |
| 0x2108 + 0x100 * n | *DBMn* | Data Breakpoint Address Mask n |
| 0x2110 + 0x100 * n | *DBASIDn* | Data Breakpoint ASID n |
| 0x2118 + 0x100 * n | *DBCn* | Data Breakpoint Control n |
| 0x2120 + 0x100 * n | *DBVn* | Data Breakpoint Value n |
| 0x2124 + 0x100*n | *DBVHn* | Data Breakpoint Value High n |
| n is breakpoint number as 0 or 1 | | |

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

### 11.2.9.1 Data Breakpoint Status (DBS) Register

**Compliance Level:** Implemented if data breakpoints are implemented.

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

The ASIDsup field indicates whether ASID compares are supported.

**Figure 11.7 DBS Register Format**

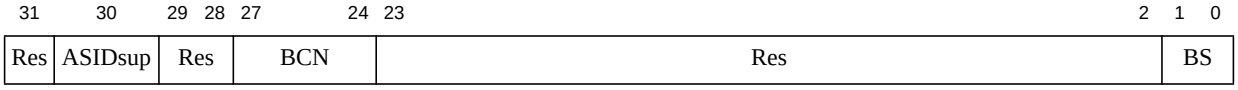



**Table 11.14 DBS Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| Res | 31 | Must be written as zero; returns zero on read. | R | 0 |
| ASID | 30 | Indicates that ASID compares are supported in data breakpoints.<br>0: Not supported<br>1: Supported | R | TLB MMU - 1<br>FM MMU - 0 |
| Res | 29:28 | Must be written as zero; returns zero on read. | R | 0 |
| BCN | 27:24 | Number of data breakpoints implemented. | R | 2 |
| Res | 23:2 | Must be written as zero; returns zero on read. | R | 0 |
| BS | 1:0 | Break status for breakpoint n is at BS[n], with n from 0 to 1. The bit is set to 1 when the condition for the corresponding breakpoint has matched. | R/W0 | Undefined |

### 11.2.9.2 Data Breakpoint Address n (DBAn) Register

**Compliance Level:** Implemented only for implemented data breakpoints.

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

**Figure 11.8 DBAn Register Format**

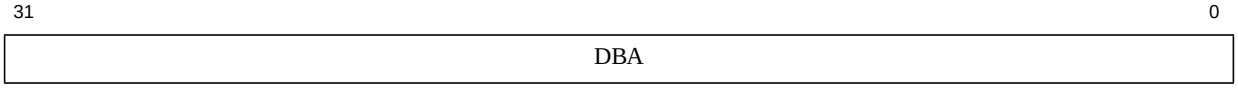



**Table 11.15 DBAn Register Field Descriptions**

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

### 11.2.9.3 Data Breakpoint Address Mask n (DBMn) Register

**Compliance Level:** Implemented only for implemented data breakpoints.

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

**Figure 11.9 DBMn Register Format**

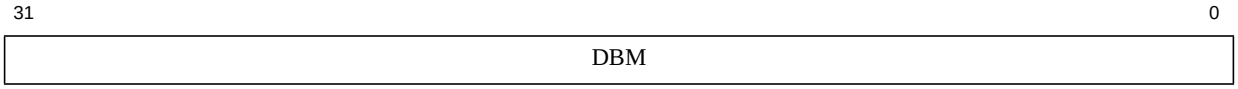



**Table 11.16 DBMn Register Field Descriptions**

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

### 11.2.9.4 Data Breakpoint ASID n (DBASIDn) Register

**Compliance Level:** Implemented only for implemented data breakpoints.

For processors with a TLB based MMU, this register is used to define an ASID value to be used in the match expression. For cores with the FM MMU, this register is reserved and reads as 0.

**Figure 11.10 DBASIDn Register Format**

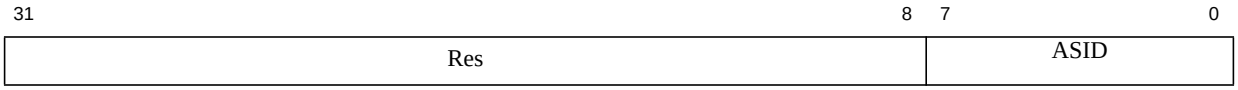



**Table 11.17 DBASIDn Register Field Descriptions**

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

### 11.2.9.5 Data Breakpoint Control n (DBCn) Register

**Compliance Level:** Implemented only for implemented data breakpoints.

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

**Figure 11.11 DBCn Register Format**

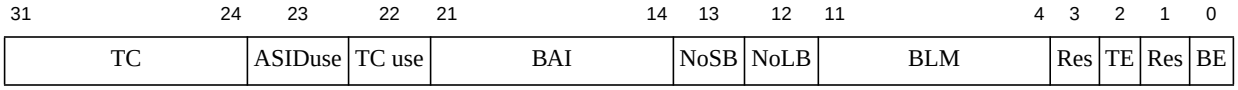


**Table 11.18 DBCn Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| TC | 31:24 | The value of TC to match in the comparison to determine if data break is to be taken. TC value is ignored if TCuse is set to 0 | R/W | Undefined |
| ASIDuse | 23 | Use ASID value in compare for data breakpoint n:<br>0: Don't use ASID value in compare<br>1: Use ASID value in compare | R/W | Undefined |
| TCuse | 22 | Use TC value in comparison for data breakpoint n<br>0: Do not use TC value in compare<br>1: Use TC value in compare | R/W | Undefined |
| BAI | 21:14 | Byte access ignore controls ignore of access to a specific byte. BAI[0] ignores access to byte at bits [7:0] of the data bus, BAI[1] ignores access to byte at bits [15:8], etc.<br>0: Condition depends on access to corresponding byte<br>1: Access for corresponding byte is ignored | R/W | Undefined |
| NoSB | 13 | Controls if condition for data breakpoint is not fulfilled on a store transaction:<br>0: Condition may be fulfilled on store transaction<br>1: Condition is never fulfilled on store transaction | R/W | Undefined |
| NoLB | 12 | Controls if condition for data breakpoint is not fulfilled on a load transaction:<br>0: Condition may be fulfilled on load transaction<br>1: Condition is never fulfilled on load transaction | R/W | Undefined |
| BLM | 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.:<br>0: Compare corresponding byte lane<br>1: Mask corresponding byte lane | R/W | Undefined |
| Res | 3 | Must be written as zero; returns zero on reads. | R | 0 |
| TE | 2 | Use data breakpoint n as triggerpoint:<br>0: Don't use it as triggerpoint<br>1: Use it as triggerpoint | R/W | 0 |
| Res | 1 | Must be written as zero; returns zero on reads. | R | 0 |
| BE | 0 | Use data breakpoint n as breakpoint:<br>0: Don't use it as breakpoint<br>1: Use it as breakpoint | R/W | 0 |

#### 11.2.9.6 Data Breakpoint Value n (DBVn) Register

**Compliance Level:** Implemented only for implemented data breakpoints.

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

**Figure 11.12 DBVn Register Format**

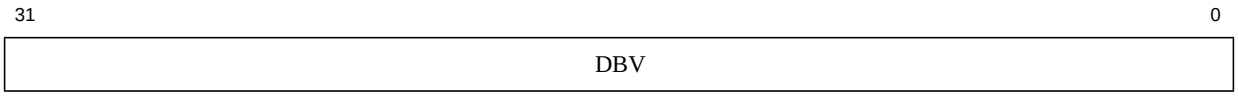



**Table 11.19 DBVn Register Field Descriptions**

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

### 11.2.9.7 Data Breakpoint Value High n (DBVHn) Register

**Compliance Level:** Implemented only for implemented data breakpoints.

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

**Figure 11.13 DBVHn Register Format**


**Table 11.20 DBVHn Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| DBVH | 31:0 | Data breakpoint value high for condition. This register provides the high order bits [63:32] for data value on double-word floating point loads and stores. | R/W | Undefined |

## 11.3 Test Access Port (TAP)

The following main features are supported by the TAP module:

- 5-pin industry standard JTAG Test Access Port (*TCK*, *TMS*, *TDI*, *TDO*, *TRST_N*) interface which is compatible with IEEE Std. 1149.1.
- Target chip and EJTAG feature identification available through the Test Access Port (TAP) controller.
- The processor can access external memory on the EJTAG Probe serially through the EJTAG pins. This is achieved through Processor Access (PA), and is used to eliminate the use of the system memory for debug routines.
- Support for both ROM based debugger and debugging both through TAP.

### 11.3.1 EJTAG Internal and External Interfaces

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

**Table 11.21 EJTAG Interface Pins**

| Pin | Type | Description |
|---|---|---|
| *TCK* | I | Test Clock Input<br>Input clock used to shift data into or out of the Instruction or data registers. The *TCK* clock is independent of the processor clock, so the EJTAG probe can drive *TCK* independently of the processor clock frequency.<br>The core signal for this is called *EJ_TCK* |
| *TMS* | I | Test Mode Select Input<br>The *TMS* input signal is decoded by the TAP controller to control test operation. *TMS* is sampled on the rising edge of *TCK*.<br>The core signal for this is called *EJ_TMS* |
| *TDI* | I | Test Data Input<br>Serial input data (*TDI*) is shifted into the Instruction register or data registers on the rising edge of the *TCK* clock, depending on the TAP controller state.<br>The core signal for this is called *EJ_TDI* |
| *TDO* | O | Test Data Output<br>Serial output data is shifted from the Instruction or data register to the *TDO* pin on the falling edge of the *TCK* clock. When no data is shifted out, the *TDO* is 3-stated.<br>The core signal for this is called *EJ_TDO* with output enable controlled by *EJ_TDOzstate*. |
| *TRST_N* | I | Test Reset Input (Optional pin)<br>The *TRST_N* pin is an active-low signal for asynchronous reset of the TAP controller and instruction in the TAP module, independent of the processor logic. The processor is not reset by the assertion of *TRST_N*.<br>The core signal for this is called *EJ_TRST_N*<br>This signal is optional, but power-on reset must apply a low pulse on this signal at power-on and then leave it high, in case the signal is not available as a pin on the chip. If available on the chip, then it must be low on the board when the EJTAG debug features are unused by the probe. |

### 11.3.2 Test Access Port Operation

The TAP controller is controlled by the Test Clock (*TCK*) and Test Mode Select (*TMS*) inputs. These two inputs determine whether an the Instruction register scan or data register scan is performed. The TAP consists of a small controller, driven by the *TCK* input, which responds to the *TMS* input as shown in the state diagram in Figure 11.14. The TAP uses both clock edges of *TCK*. *TMS* and *TDI* are sampled on the rising edge of *TCK*, while *TDO* changes on the falling edge of *TCK*.

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

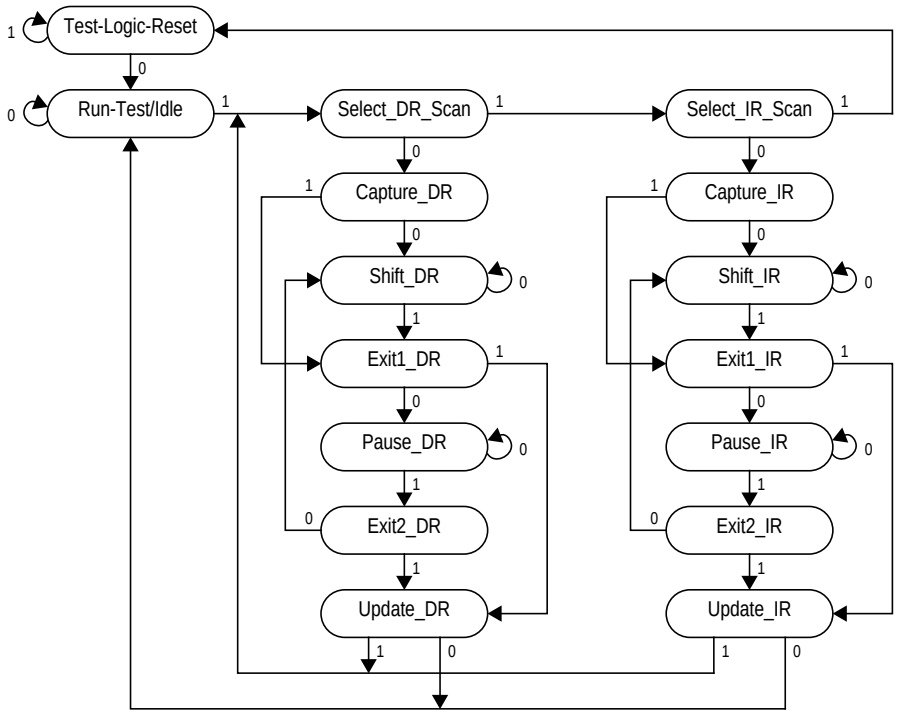
When test access is required, a protocol is applied via the *TMS* and *TCK* inputs, causing the TAP to exit the *Test-Logic-Reset* state and move through the appropriate states. From the *Run-Test/Idle* state, an Instruction register scan or a data register scan can be issued to transition the TAP through the appropriate states shown in Figure 11.14.

The states of the data and instruction register scan blocks are mirror images of each other adding symmetry to the protocol sequences. The first action that occurs when either block is entered is a capture operation. For the data registers, the *Capture-DR* state is used to capture (or parallel load) the data into the selected serial data path. In the Instruction register, the *Capture-IR* state is used to capture status information into the Instruction register.

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

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

**Figure 11.14 TAP Controller State Diagram**




### 11.3.2.1 Test-Logic-Reset State

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

### 11.3.2.2 Run-Test/Idle State

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

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

#### 11.3.2.3 Select_DR_Scan State

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

#### 11.3.2.4 Select_IR_Scan State

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

#### 11.3.2.5 Capture_DR State

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

#### 11.3.2.6 Shift_DR State

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

#### 11.3.2.7 Exit1_DR State

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

#### 11.3.2.8 Pause_DR State

The *Pause_DR* state allows the controller to temporarily halt the shifting of data through the test data register in the serial path between *TDI* and *TDO*. All test data registers selected by the current instruction retain their previous state. If *TMS* is sampled LOW on the rising edge of *TCK*, the controller remains in the *Pause_DR* state. A HIGH on *TMS* causes the controller to transition to the *Exit2_DR* state. The instruction cannot change while the TAP controller is in this state.

#### 11.3.2.9 Exit2_DR State

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

#### 11.3.2.10 Update_DR State

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

If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Run-Test/Idle* state. A HIGH on *TMS* causes the controller to transition to the *Select_DR_Scan* state. The instruction cannot change while the TAP controller is in this state and all shift register stages in the test data registers selected by the current instruction retain their previous state.

#### 11.3.2.11 Capture_IR State

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

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

#### 11.3.2.12 Shift_IR State

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

#### 11.3.2.13 Exit1_IR State

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

#### 11.3.2.14 Pause_IR State

The *Pause_IR* state allows the controller to temporarily halt the shifting of data through the instruction register in the serial path between *TDI* and *TDO*. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller remains in the *Pause_IR* state. A HIGH on *TMS* causes the controller to transition to the *Exit2_IR* state. The instruction cannot change while the TAP controller is in this state.

#### 11.3.2.15 Exit2_IR State

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

#### 11.3.2.16 Update_IR State

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

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

### 11.3.3 Test Access Port (TAP) Instructions

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

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

**Table 11.22 Implemented EJTAG Instructions**

| Value | Instruction | Function |
|---|---|---|
| 0x01 | IDCODE | Select Chip Identification data register |
| 0x03 | IMPCODE | Select Implementation register |
| 0x08 | ADDRESS | Select Address register |
| 0x09 | DATA | Select Data register |
| 0x0A | CONTROL | Select EJTAG Control register |
| 0x0B | ALL | Select the Address, Data and EJTAG Control registers |
| 0x0C | EJTAGBOOT | Set EjtagBrk, ProbEn and ProbTrap to 1 as reset value |
| 0x0D | NORMALBOOT | Set EjtagBrk, ProbEn and ProbTrap to 0 as reset value |
| 0x0E | FASTDATA | Selects the Data and Fastdata registers |
| 0x10 | TCBCONTROLA | Selects the *TCBTCONTROLA* register in the Trace Control Block |
| 0x11 | TCBCONTROLB | Selects the *TCBTCONTROLB* register in the Trace Control Block |
| 0x12 | TCBDATA | Selects the *TCBDATA* register in the Trace Control Block |
| 0x13 | TCBCONTROLC | Selects the *TCBTCONTROLC* register in the Trace Control Block |
| 0x14 | PCSAMPLE | Selects the *PCSAMPLE* register |
| 0x1F | BYPASS | Bypass mode |

#### 11.3.3.1 BYPASS Instruction

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

#### 11.3.3.2 IDCODE Instruction

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

#### 11.3.3.3 IMPCODE Instruction

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

#### 11.3.3.4 ADDRESS Instruction

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

#### 11.3.3.5 DATA Instruction

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

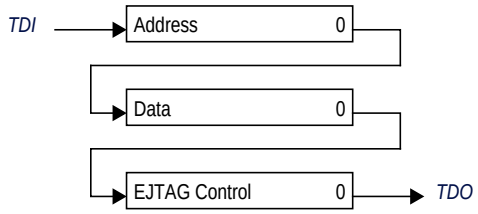
#### 11.3.3.6 CONTROL Instruction

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

#### 11.3.3.7 ALL Instruction

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

**Figure 11.15 Concatenation of the EJTAG Address, Data and Control Registers**




#### 11.3.3.8 EJTAGBOOT Instruction

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

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

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

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

#### 11.3.3.9 NORMALBOOT Instruction

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

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

#### 11.3.3.10 FASTDATA Instruction

This selects the Data and the Fastdata registers at once, as shown in Figure 11.16.

**Figure 11.16 TDI to TDO Path When in Shift-DR State and FASTDATA Instruction is Selected**


#### 11.3.3.11 TCBCONTROLA Instruction

This instruction is used to select the TCBCONTROLA register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace Control Block is present. If no TCB is present, then this instruction will select the Bypass register.

#### 11.3.3.12 TCBCONTROLB Instruction

This instruction is used to select the TCBCONTROLB register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace Control Block is present. If no TCB is present, then this instruction will select the Bypass register.

#### 11.3.3.13 TCBCONTROLC Instruction

This instruction is used to select the TCBCONTROLC register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace Control Block is present. If no TCB is present, then this instruction will select the Bypass register.

#### 11.3.3.14 TCBDATA Instruction

This instruction is used to select the TCBDATA register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace Control Block is present. If no TCB is present, then this instruction will select the Bypass register. It should be noted that the TCBDATA register is only an access register to other TCB registers. The width of the TCBDATA register is dependent on the specific TCB register.

#### 11.3.3.15 PCSAMPLE Instruction

This instruction is used to select the PCSAMPLE register to be connected between *TDI* and *TDO*. This register is always implemented.

## 11.4 EJTAG TAP Registers

The EJTAG TAP Module has one Instruction register and a number of data registers, all accessible through the TAP:

### 11.4.1 Instruction Register

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

$00001_2$, as for the IDCODE instruction. This forces the device into the functional mode and selects the Device ID register. The Instruction register is 5 bits wide. The instruction shifted in takes effect for the following data register scan operation. A list of the implemented instructions are listed in Table 11.22.

## 11.4.2 Data Registers Overview

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

This description applies in general to the following data registers:

- Bypass Register
- Device Identification Register
- Implementation Register
- EJTAG Control Register (ECR)
- Processor Access Address Register
- Processor Access Data Register
- FastData Register

### 11.4.2.1 Bypass Register

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

### 11.4.2.2 Device Identification (ID) Register

The *Device Identification* register is defined by IEEE 1149.1, to identify the device's manufacturer, part number, revision, and other device-specific information. Table 11.23 shows the bit assignments defined for the read-only Device Identification Register, and inputs to the core determine the value of these bits. These bits can be scanned out of the *ID* register after being selected. The register is selected when the Instruction register is loaded with the IDCODE instruction.

**Figure 11.17 Device Identification Register Format**

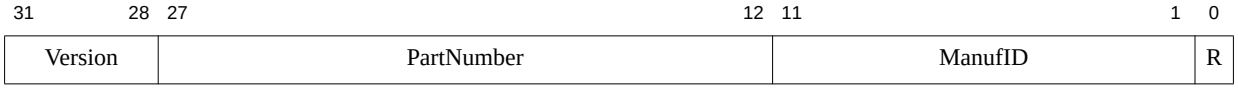


**Table 11.23 Device Identification Register**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| Version | 31:28 | **Version** (4 bits)<br>This field identifies the version number of the processor derivative. | R | *EJ_Version[3:0]* |
| PartNumber | 27:12 | **Part Number** (16 bits)<br>This field identifies the part number of the processor derivative. | R | *EJ_PartNumber[15:0]* |
| ManufID | 11:1 | **Manufacturer Identity** (11 bits)<br>Accordingly to IEEE 1149.1-1990, the manufacturer identity code shall be a compressed form of the JEDEC Publications 106-A. | R | *EJ_ManufID[10:0]* |
| R | 0 | reserved | R | 1 |

### 11.4.2.3 Implementation Register

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

**Figure 11.18 Implementation Register Format**

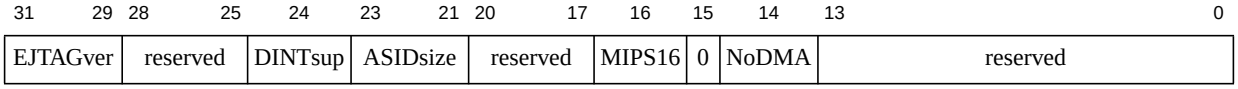



**Table 11.24 Implementation Register Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| EJTAGver | 31:29 | EJTAG Version.<br>3: Version 3.1 | R | 3 |
| reserved | 28:25 | reserved | R | 0 |
| DINTsup | 24 | DINT Signal Supported from Probe<br>This bit indicates if the DINT signal from the probe is supported:<br>0: DINT signal from the probe is not supported<br>1: Probe can use DINT signal to make debug interrupt. | R | *EJ_DINTsup* |
| ASIDsize | 23:21 | Size of ASID field in implementation:<br>0: No ASID in implementation<br>2: 8-bit ASID<br>1,3: Reserved | R | TLB MMU- 2<br>FM MMU- 0 |
| reserved | 20:17 | reserved | R | 0 |
| MIPS16 | 16 | Indicates whether MIPS16 is implemented<br>0: No MIPS16 support<br>1: MIPS16 implemented | R | 1 |
| reserved | 15 | reserved | R | 0 |
| NoDMA | 14 | No EJTAG DMA Support | R | 1 |
| reserved | 13:0 | reserved | R | 0 |

#### 11.4.2.4 EJTAG Control Register

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

The EJTAG Control register is not updated in the *Update-DR* state unless the Reset occurred (Rocc) bit 31, is either 0 or written to 0. This is in order to ensure prober handling of processor accesses.

The value used for reset indicated in the table below takes effect on CPU resets, but not on TAP controller resets by e.g. *TRST_N*. *TCK* clock is not required when the CPU reset occurs, but the bits are still updated to the reset value when the *TCK* applies. The first 5 *TCK* clocks after CPU resets may result in reset of the bits, due to synchronization between clock domains.

**Figure 11.19 EJTAG Control Register Format**

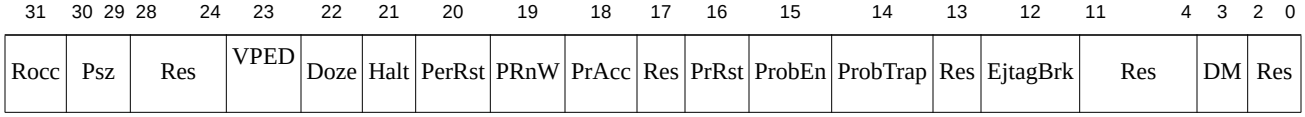



**Table 11.25 EJTAG Control Register Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| Rocc | 31 | Reset Occurred<br>The bit indicates if a CPU reset has occurred:<br>0: No reset occurred since bit last cleared.<br>1: Reset occurred since bit last cleared.<br>The Rocc bit will keep the 1 value as long as reset is applied.<br>This bit must be cleared by the probe, to acknowledge that the incident was detected.<br>The EJTAG Control register is not updated in the *Update-DR* state unless Rocc is 0, or written to 0. This is in order to ensure proper handling of processor access. | R/W | 1 |

**Table 11.25 EJTAG Control Register Descriptions (Continued)**

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>Psz[1:0]</td><td>30:29</td><td>Processor Access Transfer Size<br>These bits are used in combination with the lower two address bits of the Address register to determine the size of a processor access transaction. The bits are only valid when processor access is pending.
<table>
<tr><th>PAA[1:0]</th><th>Psz[1:0]</th><th>Transfer Size</th></tr>
<tr><td>00</td><td>00</td><td>Byte (LE, byte 0; BE, byte 3)</td></tr>
<tr><td>01</td><td>00</td><td>Byte (LE, byte 1; BE, byte 2)</td></tr>
<tr><td>10</td><td>00</td><td>Byte (LE, byte 2; BE, byte 1)</td></tr>
<tr><td>11</td><td>00</td><td>Byte (LE, byte 3; BE, byte 0)</td></tr>
<tr><td>00</td><td>01</td><td>Halfword (LE, bytes 1:0; BE, bytes 3:2)</td></tr>
<tr><td>10</td><td>01</td><td>Halfword (LE, bytes 3:2; BE, bytes 1:0)</td></tr>
<tr><td>00</td><td>10</td><td>Word (LE, BE; bytes 3, 2, 1, 0)</td></tr>
<tr><td>00</td><td>11</td><td>Triple (LE, bytes 2, 1, 0; BE, bytes 3, 2,1)</td></tr>
<tr><td>01</td><td>11</td><td>Triple (LE, bytes 3, 2, 1; BE, bytes 2, 1, 0)</td></tr>
<tr><td colspan="2">All others</td><td>Reserved</td></tr>
</table>
Note: LE=little endian, BE=big endian, the byte# refers to the byte number in a 32-bit register, where byte 3 = bits 31:24; byte 2 = bits 23:16; byte 1 = bits 15:8; byte 0=bits 7:0, independently of the endianess.</td><td>R</td><td>Undefined</td></tr>
<tr><td>Res</td><td>28:24</td><td>reserved</td><td>R</td><td>0</td></tr>
<tr><td>VPED</td><td>23</td><td>VPE Disable state<br>EJTAG state is not valid if this bit is set<br>0: VPE is currently enabled<br>1: VPE is currently disabled</td><td>R</td><td>1</td></tr>
<tr><td>Doze</td><td>22</td><td>Doze state<br>The Doze bit indicates any kind of low power mode. The value is sampled in the Capture-DR state of the TAP controller:<br>0: CPU not in low power mode.<br>1: CPU is in low power mode<br>Doze includes the Reduced Power (RP) and WAIT power-reduction modes.</td><td>R</td><td>0</td></tr>
<tr><td>Halt</td><td>21</td><td>Halt state<br>The Halt bit indicates if the internal system bus clock is running or stopped. The value is sampled in the Capture-DR state of the TAP controller:<br>0: Internal system clock is running<br>1: Internal system clock is stopped</td><td>R</td><td>0</td></tr>
</table>

**Table 11.25 EJTAG Control Register Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| PerRst | 20 | Peripheral Reset<br>When the bit is set to 1, it is only guaranteed that the peripheral reset has occurred in the system when the read value of this bit is also 1. This is to ensure that the setting from the *TCK* clock domain gets effect in the CPU clock domain, and in peripherals.<br>When the bit is written to 0, then the bit must also be read as 0 before it is guaranteed that the indication is cleared in the CPU clock domain also.<br>This bit controls the *EJ_PerRst* signal on the core. | R/W | 0 |
| PRnW | 19 | Processor Access Read and Write<br>This bit indicates if the pending processor access is for a read or write transaction, and the bit is only valid while PrAcc is set:<br>0: Read transaction<br>1: Write transaction | R | Undefined |
| PrAcc | 18 | Processor Access (PA)<br>Read value of this bit indicates if a Processor Access (PA) to the EJTAG memory is pending:<br>0: No pending processor access<br>1: Pending processor access<br>The probe's software must clear this bit to 0 to indicate the end of the PA. Write of 1 is ignored.<br>A pending Processor Access is cleared when Rocc is set, but another PA may occur just after the reset if a debug exception occurs. Finishing a Processor Access is not accepted while the Rocc bit is set. This is to avoid that a Processor Access occurring after the reset is finished due to indication of a Processor Access that occurred before the reset.<br>The FASTDATA access can clear this bit. | R/W0 | 0 |
| Res | 17 | reserved | R | 0 |
| PrRst | 16 | Processor Reset (Implementation dependent behavior)<br>When the bit is set to 1, then it is only guaranteed that this setting has taken effect in the system when the read value of this bit is also 1. This is to ensure that the setting from the *TCK* clock domain gets effect in the CPU clock domain, and in peripherals.<br>When the bit is written to 0, then the bit must also be read as 0 before it is guaranteed that the indication is cleared in the CPU clock domain also.<br>This bit controls the *EJ_PrRst* signal. If the signal is used in the system, then it must be ensured that both the processor and all devices required for a reset are properly reset. Otherwise the system may fail or hang. The bit resets itself, since the EJTAG Control register is reset by a reset. | R/W | 0 |

**Table 11.25 EJTAG Control Register Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| ProbEn | 15 | Probe Enable<br>This bit indicates to the CPU if the EJTAG memory is handled by the probe so processor accesses are answered:<br>0: The probe does not handle EJTAG memory transactions<br>1: The probe does handle EJTAG memory transactions<br>It is an error by the software controlling the probe if it sets the ProbTrap bit to 1, but resets the ProbEn to 0. The operation of the processor is UNDEFINED in this case.<br>The ProbEn bit is reflected as a read-only bit in the ProbEn bit, bit 0, in the Debug Control Register (DCR).<br>The read value indicates the effective value in the DCR, due to synchronization issues between *TCK* and CPU clock domains; however, it is ensured that change of the ProbEn prior to setting the EjtagBrk bit will have effect for the debug handler executed due to the debug exception.<br>The reset value of the bit depends on whether the EJTAGBOOT indication is given or not:<br>No EJTAGBOOT indication given: 0<br>EJTAGBOOT indication given: 1 | R/W | 0 or 1 from EJTAGBOOT |
| ProbTrap | 14 | Probe Trap<br>This bit controls the location of the debug exception vector:<br>0: In normal memory 0xBFC0.0480<br>1: In EJTAG memory at 0xFF20.0200 in dmseg<br>Valid setting of the ProbTrap bit depends on the setting of the ProbEn bit, see comment under ProbEn bit.<br>The ProbTrap should not be set to 1, for debug exception vector in EJTAG memory, unless the ProbEn bit is also set to 1 to indicate that the EJTAG memory may be accessed.<br>The read value indicates the effective value to the CPU, due to synchronization issues between *TCK* and CPU clock domains; however, it is ensured that change of the ProbTrap bit prior to setting the EjtagBrk bit will have effect for the EjtagBrk.<br>The reset value of the bit depends on whether the EJTAGBOOT indication is given or not:<br>No EJTAGBOOT indication given: 0<br>EJTAGBOOT indication given: 1 | R/W | 0 or 1 from EJTAGBOOT |
| Res | 13 | reserved | R | 0 |
| EjtagBrk | 12 | EJTAG Break<br>Setting this bit to 1 causes a debug exception to the processor, unless the CPU was in debug mode or another debug exception occurred. When the debug exception occurs, the processor core clock is restarted if the CPU was in low power mode. This bit is cleared by hardware when the debug exception is taken.<br>The reset value of the bit depends on whether the EJTAGBOOT indication is given or not:<br>No EJTAGBOOT indication given: 0<br>EJTAGBOOT indication given: 1 | R/W1 | 0 or 1 from EJTAGBOOT |
| Res | 11:4 | reserved | R | 0 |

**Table 11.25 EJTAG Control Register Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit(s) | | | |
| DM | 3 | Debug Mode<br>This bit indicates the debug or non-debug mode:<br>0: Processor is in non-debug mode<br>1: Processor is in debug mode<br>The bit is sampled in the *Capture-DR* state of the TAP controller. | R | 0 |
| Res | 2:0 | reserved | R | 0 |

## 11.4.3 Processor Access Address Register

The Processor Access Address (*PAA*) register is used to provide the address of the processor access in the dmseg, and the register is only valid when a processor access is pending. The length of the Address register is 32 bits, and this register is selected by shifting in the ADDRESS instruction.

### 11.4.3.1 Processor Access Data Register

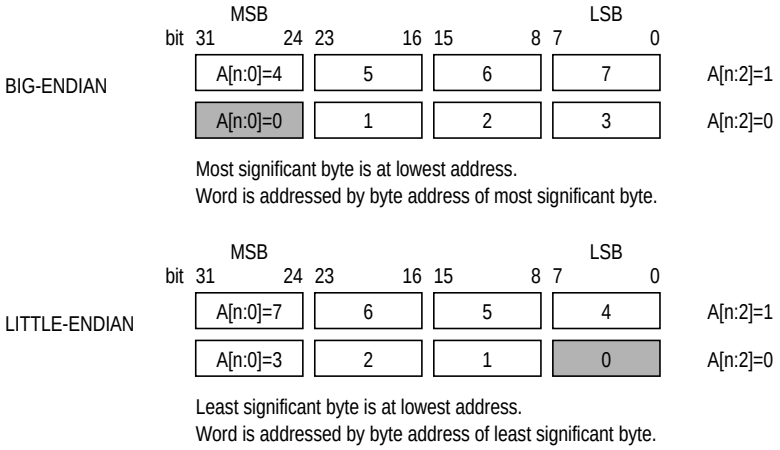
The Processor Access Data (*PAD*) register is used to provide data value to and from a processor access. The length of the Data register is 32 bits, and this register is selected by shifting in the DATA instruction.

The register has the written value for a processor access write due to a CPU store to the dmseg, and the output from this register is only valid when a processor access write is pending. The register is used to provide the data value for a processor access read due to a CPU load or fetch from the dmseg, and the register should only be updated with a new value when a processor access write is pending.

The *PAD* register is 32 bits wide. Data alignment is not used for this register, so the value in the *PAD* register matches data on the internal bus. The undefined bytes for a PA write are undefined, and for a *PAD* read then 0 (zero) must be shifted in for the unused bytes.

The organization of bytes in the *PAD* register depends on the endianess of the core, as shown in Figure 11.20. The endian mode for debug/kernel mode is determined by the state of the *SI_Endian* input at power-up.

**Figure 11.20 Endian Formats for the PAD Register**



The size of the transaction and thus the number of bytes available/required for the *PAD* register is determined by the Psz field in the *ECR*.

### 11.4.4 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 11.21 Fastdata Register Format**

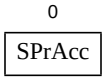



**Table 11.26 Fastdata Register Field Description**

| Fields | | Description | Read / Write | Power-up 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.<br>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 (0xFF20.0000 - 0xFF20.000F) 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 (0xFF20.0000 to 0xFF20.000F).

Table 11.27 shows the values of the PrAcc and SPrAcc bits and the results of a Fastdata access. .

**Table 11.27 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 FAST-DATA | 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 size is a 32-bit word.

The Rocc bit of the Control register is not used for the FASTDATA operation.

## 11.5 TAP Processor Accesses

The TAP modules support handling of fetches, loads and stores from the CPU through the dmseg segment, whereby the TAP module can operate like a *slave unit* connected to the on-chip bus. The core can then execute code taken from the EJTAG Probe and it can access data (via a load or store) which is located on the EJTAG Probe. This occurs in a serial way through the EJTAG interface: the core can thus execute instructions e.g. debug monitor code, without occupying the memory.

Accessing the dmseg segment (EJTAG memory) can only occur when the processor accesses an address in the range from 0xFF20.0000 to 0xFF2F.FFFF, the ProbEn bit is set, and the processor is in debug mode (DM=1). In addition the LSNM bit in the CP0 Debug register controls transactions to/from the dmseg.

When a debug exception is taken, while the ProbTrap bit is set, the processor will start fetching instructions from address 0xFF20.0200.

A pending processor access can only finish if the probe writes 0 to PrAcc or by a reset.

### 11.5.1 Fetch/Load and Store From/To the EJTAG Probe Through dmseg

1. The internal hardware latches the requested address into the PA Address register (in case of the Debug exception: 0xFF20.0200).

2. The internal hardware sets the following bits in the EJTAG Control register:
   PrAcc = 1 (selects Processor Access operation)

PRnW = 0 (selects processor read operation)
Psz[1:0] = value depending on the transfer size

3. The EJTAG Probe selects the EJTAG Control register, shifts out this control register's data and tests the PrAcc status bit (Processor Access): when the PrAcc bit is found 1, it means that the requested address is available and can be shifted out.

4. The EJTAG Probe checks the PRnW bit to determine the required access.

5. The EJTAG Probe selects the PA Address register and shifts out the requested address.

6. The EJTAG Probe selects the PA Data register and shifts in the instruction corresponding to this address.

7. The EJTAG Probe selects the EJTAG Control register and shifts a PrAcc = 0 bit into this register to indicate to the processor that the instruction is available.

8. The instruction becomes available in the instruction register and the processor starts executing.

9. The processor increments the program counter and outputs an instruction read request for the next instruction. This starts the whole sequence again.

Using the same protocol, the processor can also execute a load instruction to access the EJTAG Probe's memory. For this to happen, the processor must execute a load instruction (e.g. a LW, LH, LB) with the target address in the appropriate range.

Almost the same protocol is used to execute a store instruction to the EJTAG Probe's memory through dmseg. The store address must be in the range: 0xFF20.0000 to 0xFF2F.FFFF, the ProbEn bit must be set and the processor has to be in debug mode (DM=1). The sequence of actions is found below:

1. The internal hardware latches the requested address into the PA Address register

2. The internal hardware latches the data to be written into the PA Data register.

3. The internal hardware sets the following bits in the EJTAG Control register:
PrAcc = 1 (selects Processor Access operation)
PRnW = 1 (selects processor write operation)
Psz[1:0] = value depending on the transfer size

4. The EJTAG Probe selects the EJTAG Control register, shifts out this control register's data and tests the PrAcc status bit (Processor Access): when the PrAcc bit is found 1, it means that the requested address is available and can be shifted out.

5. The EJTAG Probe checks the PRnW bit to determine the required access.

6. The EJTAG Probe selects the PA Address register and shifts out the requested address.

7. The EJTAG Probe selects the PA Data register and shifts out the data to be written.

8. The EJTAG Probe selects the EJTAG Control register and shifts a PrAcc = 0 bit into this register to indicate to the processor that the write access is finished.

9. The EJTAG Probe writes the data to the requested address in its memory.

10. The processor detects that PrAcc bit = 0, which means that it is ready to handle a new access.

The above examples imply that no reset occurs during the operations, and that Rocc is cleared.

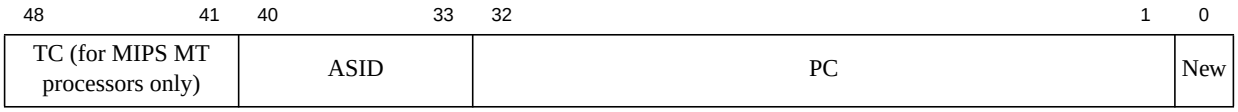
## 11.6 PC Sampling

The PC sampling feature enables sampling of the PC value periodically. This information can be used for statistical profiling of the program akin to gprof. This information is also very useful for detecting hot-spots in the code. PC sampling cannot be turned on or off, that is, the PC value is continually sampled.

The presence or absence of the PC Sampling feature is available in the Debug Control register as bit 9(PCS).The sampled PC values are written into a TAP register. The old value in the TAP register is overwritten by a new value even if this register has not be read out by the debug probe. The sample rate is specified in a manner similar to the PDtrace synchronization period, with three bits. These bits in the Debug Control register are 8:6 and called PCSR (PC Sample Rate). These three bits take the value $2^5$ to $2^{12}$ similar to SyncPeriod. Note that the processor samples PC even when it is asleep, that is, in a WAIT state. This permits an analysis of the amount of time spent by a processor in WAIT state which may be used for example to revert to a low power mode during the non-execution phase of a real-time application.

The sampled values includes a new data bit, the PC, the ASID of the sampled PC as well as the Thread Context id if the processor implements the MIPS MT ASE. Figure shows the format of the sampled values in the TAP register PCsample. The new data bit is used by the probe to determine if the PCsample register data just read out is new or already been read and must be discarded.

**Figure 11.22 TAP Register PCsample Format**




The sampled PC value is the PC of the graduating instruction in the current cycle. If the processor is stalled when the PC sample counter overflows, then the sampled PC is the PC of the next graduating instruction. The processor continues to sample the PC value even when it is in Debug mode.

### 11.6.1 PC Sampling in Wait State

When the processor is in a WAIT state to save power for example, an external agent might want to know how long it stays in the WAIT state. But counting cycles to update the PC sample value is a waste of power. Hence, when in a WAIT state, the processor must simply switch the New bit to 1 every time it is set to 0 by the probe hardware. Hence, the external agent or probe reading the PC value will detect a WAIT instruction for as long as the processor remains in the WAIT state. When the processor leaves the WAIT state, then counting is resumed as before.

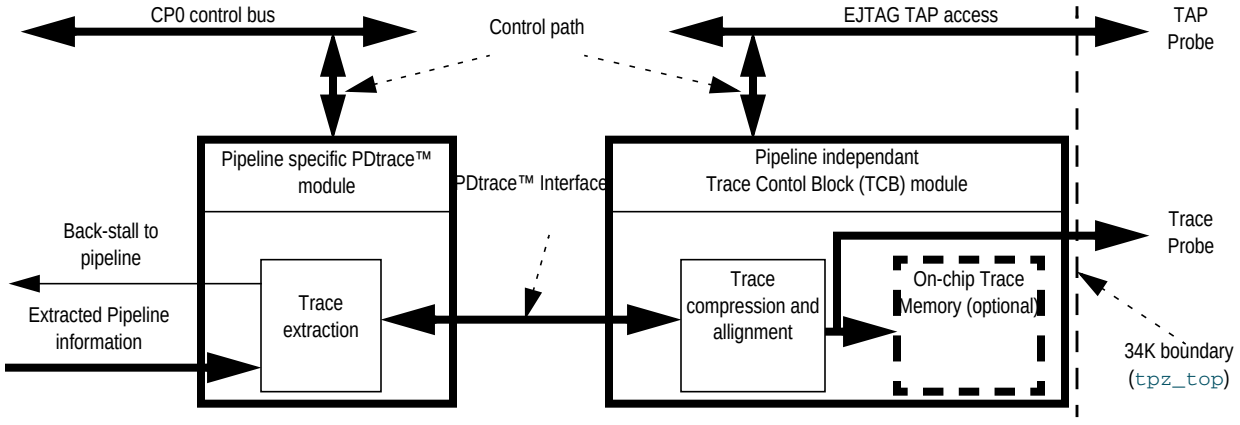
## 11.7 MIPS® Trace

MIPS Trace enables the ability to trace program flow, load/store addresses and load/store data. Several run-time options exist for the level of information which is traced, including tracing only when in specific processor modes (e.g., UserMode or KernelMode). MIPS Trace is an optional block in the 34K core. If MIPS Trace is not implemented, the rest of this chapter is irrelevant. If MIPS Trace is implemented, the *CP0 Config3*$_{TL}$ bit is set.

The pipeline specific part of MIPS Trace is architecturally specified in the *PDtrace™ Interface Specification*. The PDtrace module extracts the trace information from the processor pipeline, and presents it to a pipeline-independent module called the Trace Control Block (TCB). The TCB is specified in the *EJTAG Trace Control Block Specification*. The collective implementation of the two is called *MIPS Trace*.

When MIPS Trace is implemented, the 34K core includes both the PDtrace and the Trace Control Block (TCB) modules. The two modules "talk" to each other on the generic pin-interface called the PDtrace™ Interface. This interface is embedded inside the 34K core, and will not be discussed in detail here (read the *PDtrace™ Interface Specification* for a detailed description). While working closely together, the two parts of MIPS Trace are controlled separately by software. Figure 11.23 shows an overview of the MIPS Trace modules within the core.

**Figure 11.23 MIPS® Trace Modules in the 34K™ Core**




To some extent, the two modules both provide similar trace control features, but the access to these features is quite different. The PDtrace controls can only be reached through access to CP0 registers. The TCB controls can only be reached through EJTAG TAP access. The TCB can then control what is traced through the PDtrace™ Interface.

Before describing the MIPS Trace implemented in the 34K core, some common terminology and basic features are explained. The remaining sections of this chapter will then provide a more thorough explanation.

## 11.7.1 Processor Modes

Tracing can be enabled or disabled based on various processor modes. This section precisely describes these modes. The terminology is then used elsewhere in the document.

```
DebugMode ← (Debug_DM = 1)
ExceptionMode ← (not DebugMode) and ((Status_EXL = 1) or (Status_ERL = 1))
KernelMode ← (not (DebugMode or ExceptionMode)) and (Status_KSU = 2#00)
SupervisorMode ← (not (DebugMode or ExceptionMode)) and (Status_KSU = 2#01)
UserMode ← (not (DebugMode or ExceptionMode)) and (Status_KSU = 2#10)
```

## 11.7.2 Software Versus Hardware Control

In some of the specifications and in this text, the terms "software control" and "hardware control" are used to refer to the method for how trace is controlled. Software control is when the CP0 register *TraceControl* is used to select the modes to trace, etc. Hardware control is when the EJTAG register *TCBCONTROLA* in the TCB, via the PDtrace interface, is used to select the trace modes. The *TraceControl.TS* bit determines whether software or hardware control is active. Even in Software control mode, Trace logic will need TCK to toggle atleast once before it is turned on.

It is assumed that EJTAG probe will be connected while using Trace and probe reset sequence would toggle TCK. In order to extract Trace data out of TCB, TCBCONTROLB.En should be set to 1 even in “software control” mode.

### 11.7.3 Trace Information

The main object of trace is to show the exact program flow from a specific program execution or just a small window of the execution. In MIPS Trace this is done by providing the minimal cycle-by-cycle information necessary on the PDtrace™ interface for trace regeneration software to reproduce the trace. The following is a summary of the type of information traced:

- Only instructions which complete at the end of the pipeline are traced, and indicated with a completion-flag. The PC is implicitly pointing to the next instruction.
- Load instructions are indicated with a load-flag.
- Store instructions are indicated with a store-flag[1].
- Taken branches are indicated with a branch-taken-flag on the target instruction.
- New PC information for a branch is only traced if the branch target is unpredictable from the static program image.
- When branch targets are unpredictable, only the delta value from current PC is traced, if it is dynamically determined to reduce the number of bits necessary to indicate the new PC. Otherwise the full PC value is traced.
- When a completing instruction is executed in a different processor mode from the previous one, the new processor mode is traced.
- The first instruction is always traced as a branch target, with processor mode and full PC.
- Periodic synchronization instructions are identified with a sync-flag, and traced with the processor mode and full PC.

All the instruction flags above are combined into one 3-bit value, to minimize the bit information to trace. The possible processor modes are explained in Section 11.7.1 “Processor Modes”.

The target address is statically predictable for all branch and all jump-immediate instructions. If the branch is taken, then the branch-taken-flag will indicate this. All jump-register instructions and ERET/DERET are instructions which have an unpredictable target address. These will have full/delta PC values included in the trace information. Also treated as unpredictable are PC changes which occur due to exceptions, such as an interrupt, reset, etc.

Trace regeneration software is required to know the static program image in memory, in order to reproduce the dynamic flow with the above information. But this is usually not a problem. Only the virtual value of the PC is used. Physical memory location will typically differ.

It is possible to turn on PC delta/full information for all branches, but this should not normally be necessary. As a safety check for trace regeneration software, a periodic synchronization with a full PC is sent. The period of this synchronization is cycle based and programmable.

1. A SC (Store Conditional) instruction is not flagged as a store instruction if the load-locked bit prevented the actual store.

### 11.7.4 Load/Store Address and Data Trace Information

In addition to PC flow, it is possible to get information on the load/store addresses, as well as the data read/written. When enabled, the following information is optionally added to the trace.

- When load-address tracing is on, the full load address of the first load instruction is traced (indicated by the load-flag). For subsequent loads, a dynamically-determined delta to the previous load address is traced to compress the information which must be sent.
- When store-address tracing is on, the full store address of the first store instruction is traced (indicated by the store-flag). For subsequent stores, a dynamically-determined delta to the previous store address is traced.
- When load-data tracing is on, the full load data read by each load instruction is traced (indicated by the load-flag). Only actual read bytes are traced.
- When store-data tracing is on, the full store data written by each store instruction is traced (indicated by the store-flag). Only written bytes are traced.

After each synchronization instruction, the first load address and the first store address following this are both traced with the full address if load/store address tracing is enabled.

### 11.7.5 Programmable Processor Trace Mode Options

To enable tracing, a global Trace On signal must be set. When trace is on, it is possible to enable tracing in any combination of the processor modes described in Section 11.7.1 “Processor Modes”. In addition to this, trace can be turned on globally for all process, or only for specific processes by tracing only specific masked values of the ASID found in $EntryHi_{ASID}$.

Additionally, an EJTAG Simple Break trigger point can override the processor mode and ASID selection and turn them all on. Another trigger point can disable this override again.

### 11.7.6 Programmable Trace Information Options

The processor mode changes are always traced:

- On the first instruction.
- On any synchronization instruction.
- When the mode changes and either the previous or the current processor mode is selected for trace.

The amount of extra information traced is programmable to include:

- PC information only.
- PC and cross product of load/store address/data

If the full internal state of the processor is known prior to trace start, PC and load data are the only information needed to recreate all register values on an instruction by instruction basis.

#### 11.7.6.1 User Data Trace

In addition to the above, a special CP0 register, *UserTraceData*, can generate a data trace. When this register is written, and the global Trace On is set, then the 32-bit data written is put in the trace as special User Data information.

*Remark*: The User Data is sent even if the processor is operating in an un-traced processor mode.

### 11.7.7 Enable Trace to Probe On-chip Memory

When trace is On, based on the options listed in Section 11.7.5 "Programmable Processor Trace Mode Options", the trace information is continuously sent on the PDtrace™ interface to the TCB. The TCB must, however, be enabled to transmit the trace information to the Trace probe or to on-chip trace memory, by having the $TCBCONTROLB_{EN}$ bit set. It is possible to enable and disable the TCB in two ways:

- Set/clear the $TCBCONTROLB_{EN}$ bit via an EJTAG TAP operation.
- Initialize a TCB trigger to set/clear the $TCBCONTROLB_{EN}$ bit.

### 11.7.8 TCB Trigger

The TCB can optionally include 0 to 8 triggers. A TCB trigger can be programmed to fire from any combination of:

- Probe Trigger Input to the TCB.
- Chip-level Trigger Input to the TCB.
- Processor entry into DebugMode.

When a trigger fires it can be programmed to have any combination of actions:

- Create Probe Trigger Output from TCB.
- Create Chip-level Trigger Output from TCB.
- Set, clear, or start countdown to clear the $TCBCONTROLB_{EN}$ bit (start/end/about trigger).
- Put an information byte into the trace stream.

### 11.7.9 Cycle-by-Cycle Information

All of the trace information listed in Section 11.7.3 "Trace Information" and Section 11.7.4 "Load/Store Address and Data Trace Information", will be collected from the PDtrace™ interface by the TCB. The trace will then be compressed and aligned to fit in 64 bit trace words, with no loss of information. It is possible to exclude/include the exact cycle-by-cycle relationship between each instruction. If excluded, the number of bits required in the trace information from the TCB is reduced, and each trace word will only contain information from completing instructions.

### 11.7.10 Trace Message Format

The TCB collects trace information every cycle from the PDtrace™ interface. This information is collected into six different Trace Formats (TF1 to TF6). One important feature is that all Trace Formats have at least one non-zero bit.

### 11.7.11 Trace Word Format

After the PDtrace™ data has been turned into Trace Formats, the trace information must be streamed to either on-chip trace memory or to the trace probe. Each of the major Trace Formats are of different size. This complicates how to store this information into an on-chip memory of fixed width without too much wasted space. It also complicates how to transmit data through a fixed-width trace probe interface to off-chip memory. To minimize memory overhead and or bandwidth-loss, the Trace Formats are collected into Trace Words of fixed width.

A Trace Word (TW) is defined to be 64 bits wide. An empty/invalid TW is built of all zeros. A TW which contains one or more valid TF's is guaranteed to have a non-zero value on one of the four least significant bits [3:0]. During operation of the TCB, each TW is built from the TF's generated each clock cycle. When all 64 bits are used, the TW is full and can be sent to either on-chip trace memory or to the trace probe.

## 11.8 PDtrace™ Registers (Software Control)

The CP0 registers associated with PDtrace are listed in Table 11.28 and described in Chapter 7, "CP0 Registers of the 34K™ Core"

**Table 11.28 A List of Coprocessor 0 Trace Registers**

| Register Number | Sel | Register Name | Reference |
|---|---|---|---|
| 23 | 1 | TraceControl | Section 7.2.46 "Trace Control Register (CP0 Register 23, Select 1)" |
| 23 | 2 | TraceControl2 | Section 7.2.47 "Trace Control2 Register (CP0 Register 23, Select 2)" |
| 23 | 3 | UserTraceData | Section 7.2.48 "User Trace Data Register (CP0 Register 23, Select 3)" |
| 23 | 4 | TraceBPC | Section 7.2.49 "TraceIBPC Register (CP0 Register 23, Select 4)" |

## 11.9 Trace Control Block (TCB) Registers (Hardware Control)

The TCB registers used to control its operation are listed in Table 11.29 and Table 11.30. These registers are accessed via the EJTAG TAP interface.

**Table 11.29 TCB EJTAG Registers**

| EJTAG Register | Name | Description | Implemented |
|---|---|---|---|
| 0x10 | TCBCONTROLA | Control register in the TCB mainly used for controlling the trace input signals to the core on the PDtrace interface. See Section 11.9.1 "TCBCONTROLA Register". | Yes |
| 0x11 | TCBCONTROLB | Control register in the TCB that is mainly used to specify what to do with the trace information. The REG [25:21] field in this register specifies the number of the TCB internal register accessed by the TCBDATA register. A list of all the registers that can be accessed by the TCBDATA register is shown in Table 11.30. See Section 11.9.2 "TCBCONTROLB Register". | Yes |
| 0x12 | TCBDATA | This is used to access registers specified by the REG field in the TCBCONTROLB register. See Section 11.9.3 "TCBDATA Register". | Yes |
| 0x13 | TCBCONTROLC | Control Register in the TCB used to control and hold tracing information. See Section 11.9.4 "TCBCONTROLC Register". | Yes |

**Table 11.30 Registers Selected by *TCBCONTROLB*$_{REG}$**

| *TCBCONTROLB*$_{REG}$ field | Name | Reference | Implemented |
|---|---|---|---|
| 0 | TCBCONFIG | Section 11.9.5 “TCBCONFIG Register (Reg 0)” | Yes |
| 4 | TCBTW | Section 11.9.6 “TCBTW Register (Reg 4)” | Yes if on-chip memory exists. Otherwise No |
| 5 | TCBRDP | Section 11.9.7 “TCBRDP Register (Reg 5)” | |
| 6 | TCBWRP | Section 11.9.8 “TCBWRP Register (Reg 6)” | |
| 7 | TCBSTP | Section 11.9.9 “TCBSTP Register (Reg 7)” | |
| 16-23 | TCBTRIGx | Section 11.9.10 “TCBTRIGx Register (Reg 16-23)” | Only the number indicated by *TCBCONFIG*$_{TRIG}$ are implemented. |

## 11.9.1 TCBCONTROLA Register

The TCB is responsible for asserting or de-asserting the trace input control signals on the PDtrace interface to the core’s tracing logic. Most of the control is done using the *TCBCONTROLA* register.

The *TCBCONTROLA* register is written by an EJTAG TAP controller instruction, *TCBCONTROLA* (0x10).

The format of the *TCBCONTROLA* register is shown below, and the fields are described in Table 11.31.

**Figure 11.24 TCBCONTROLA Register Format**

| 31 … 26 | 25 … 24 | 23 | 22 … 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 … 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | VModes | ADW | SyP | TB | IO | D | E | S | K | U | ASID | G | TFCR | TLSM | TIM | On |

**Table 11.31 TCBCONTROLA Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| 0 | 31:26 | Reserved. Must be written as zero; returns zero on read. | R | 0 |
| VModes | 25:24 | This field specifies the type of tracing that is supported by the processor, as follows:<br><br>**Encoding** / **Meaning**<br>00 — PC tracing only<br>01 — PC and Load and store address tracing only<br>10 — PC, load and store address, and load and store data.<br>11 — Reserved<br><br>This field is preset to the value of *PDO_ValidModes*. | R | 10 |
| ADW | 23 | *PDO_AD* bus width.<br>0: The *PDO_AD* bus is 16 bits wide.<br>1: The *PDO_AD* bus is 32 bits wide. | R | 1 |

**Table 11.31 TCBCONTROLA Register Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| SyP | 22:20 | Used to indicate the synchronization period.<br>The period (in cycles) between which the periodic synchronization information is to be sent is defined as shown in the table below.<br><br>**SyP** / **Sync Period**<br>000 — $2^5$<br>001 — $2^6$<br>010 — $2^7$<br>011 — $2^8$<br>100 — $2^9$<br>101 — $2^{10}$<br>110 — $2^{11}$<br>111 — $2^{12}$<br><br>This field defines the value on the *PDI_SyncPeriod* signal. | R/W | 000 |
| TB | 19 | Trace All Branches. When set to one, this field indicates that the core must trace either full or incremental PC values for all branches. When set to zero, only the unpredictable branches are traced.<br>This field defines the value on the *PDI_TraceAllBranch* signal. | R/W | Undefined |
| IO | 18 | Inhibit Overflow. This bit is used to indicate to the core trace logic that slow but complete tracing is desired. Hence, the core tracing logic must not allow a FIFO overflow and discard trace data. This is achieved by stalling the pipeline when the FIFO is nearly full so that no trace records are ever lost.<br>This field defines the value on the *PDI_InhibitOverflow* signal. | R/W | Undefined |
| D | 17 | When set to one, this enables tracing in Debug mode, i.e., when the DM bit is one in the *Debug* register. For trace to be enabled in Debug mode, the On bit must be one and either the G bit must be one, or the current process must match the ASID field in this register.<br>When set to zero, trace is disabled in Debug mode, irrespective of other bits.<br>This field defines the value on the *PDI_DM* signal. | R/W | Undefined |
| E | 16 | This controls when tracing is enabled. When set, tracing is enabled when either of the EXL or ERL bits in the *Status* register is one, provided that the On bit (bit 0) is also set, and either the G bit is set, or the current process ASID matches the ASID field in this register.<br>This field defines the value on the *PDI_E* signal. | R/W | Undefined |
| S | 15 | When set, this enables tracing when the core is in Supervisor mode as defined in the MIPS32 or MIPS64 architecture specification. This is provided the On bit (bit 0) is also set, and either the G bit is set, or the current process ASID matches the ASID field in this register.<br>This field defines the value on the *PDI_S* signal. | R/W | Undefined |

**Table 11.31 TCBCONTROLA Register Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| K | 14 | When set, this enables tracing when the On bit is set and the core is in Kernel mode. Unlike the usual definition of Kernel Mode, this bit enables tracing only when the ERL and EXL bits in the *Status* register are zero. This is provided the On bit (bit 0) is also set, and either the G bit is set, or the current process ASID matches the ASID field in this register.<br>This field defines the value on the *PDI_K* signal. | R/W | Undefined |
| U | 13 | When set, this enables tracing when the core is in User mode as defined in the MIPS32 or MIPS64 architecture specification. This is provided the On bit (bit 0) is also set, and either the G bit is set, or the current process ASID matches the ASID field in this register.<br>This field defines the value on the *PDI_U* signal. | R/W | Undefined |
| ASID | 12:5 | The ASID field to match when the G bit is zero. When the G bit is one, this field is ignored.<br>This field defines the value on the *PDI_ASID* signal. | R/W | Undefined |
| G | 4 | When set, this implies that tracing is to be enabled for all processes, provided that other enabling functions (like U, S, etc.,) are also true.<br>This field defines the value on the *PDI_G* signal. | R/W | Undefined |
| TFCR | 3 | When set, this indicates to the PDtrace interface that complete information about instruction if it can be a function call or return should be traced, that is signal *PDI_TraceFuncCR* is asserted as long as this value is set to 1. It also indicates to the TCB that the optional Fcr bit must be traced in the appropriate trace formats | R/W | Undefined |
| TLSM | 2 | When set, this indicates to the PDtrace interface that complete information about Load and Store data cache miss should be traced, that is signal *PDI_TraceLSMiss* is asserted as long as this value is set to 1. It also indicates to the TCB that the optional LSm bit must be traced in the appropriate trace formats. | R/W | Undefined |
| TIM | 1 | When set, this indicates to the PDtrace interface that complete information about instruction cache miss should be traced, that is signal *PDI_TraceIMiss* is asserted as long as this value is set to 1. It also indicates to the TCB that the optional Im bit must be traced in the appropriate trace formats. | R/W | Undefined |
| On | 0 | This is the global trace enable switch to the core. When zero, tracing from the core is always disabled, unless enabled by core internal software override of the *PDI_** input pins.<br>When set to one, tracing is enabled whenever the other enabling functions are also true.<br>This field defines the value on the *PDI_TraceOn* signal. | R/W | 0 |

## 11.9.2 TCBCONTROLB Register

The TCB includes a second control register, *TCBCONTROLB* (0x11). This register generally controls what to do with the trace information received.

The format of the *TCBCONTROLB* register is shown below, and the fields are described in Table 11.32.

**Figure 11.25 TCBCONTROLB Register Format**

| 31 | 30–28 | 27–26 | 25–21 | 20 | 19–17 | 16 | 15 | 14 | 13–12 | 11–10 | 8 | 7 | 6–3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| WE | 0 | TWSrcWidth | REG | WR | 0 | RM | TR | BF | TM | TLSIF | CR | Cal | TWSrcVal | CA | OfC | EN |

**Table 11.32 TCBCONTROLB Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| WE | 31 | Write Enable.<br>Only when set to 1 will the other bits be written in *TCBCONTROLB*.<br>This bit will always read 0. | R | 0 |
| 0 | 30:28 | Reserved. Must be written as zero; returns zero on read. | R | 0 |
| TWSrc-Width | 27:26 | Used to indicate the number of bits used in the source field of the Trace Word, this is a configuration option of the core that cannot be modified by software.<br>00 - zero source field width<br>01 - two bit source field width<br>10 - four bit source field width<br>11 - reserved for future use<br>This field can either be 00 or 01 for the 34K core. | R | 00 |
| REG | 25:21 | Register select: This field select the registers accessible through the *TCBDATA* register. Legal values are shown in Table 11.30. | R/W | 0 |
| WR | 20 | Write Registers: When set, the register selected by REG field is read and written when *TCBDATA* is accessed. Otherwise the selected register is only read. | R/W | 0 |
| 0 | 19:17 | Reserved. Must be written as zero; returns zero on read. | R | 0 |
| RM | 16 | Read on-chip trace memory.<br>When written to 1, the read address-pointer of the on-chip memory is set to point to the oldest memory location written since the last reset of pointers.<br>Subsequent access to the *TCBTW* register (through the *TCBDATA* register), will automatically increment the read pointer (*TCBRDP* register) after each read. [Note: The read pointer does not auto-increment if the WR field is one.]<br>When the write pointer is reached, this bit is automatically reset to 0, and the *TCBTW* register will read all zeros.<br>Once set to 1, writing 1 again will have no effect. The bit is reset by setting the TR bit or by reading the last Trace word in *TCBTW*.<br>This bit is reserved if on-chip memory is not implemented. | R/W1 | 0 |
| TR | 15 | Trace memory reset.<br>When written to one, the address pointers for the on-chip trace memory are reset to zero. Also the RM bit is reset to 0.<br>This bit is automatically de-asserted back to 0, when the reset is completed.<br>This bit is reserved if on-chip memory is not implemented. | R/W1 | 0 |
| BF | 14 | Buffer Full indicator that the TCB uses to communicate to external software in the situation that the on-chip trace memory is being deployed in the **trace-from** and **trace-to** mode. (See Section 11.13 “TCB On-Chip Trace Memory”)<br>This bit is cleared when writing 1 to the TR bit<br>This bit is reserved if on-chip memory is not implemented. | R | 0 |

**Table 11.32 TCBCONTROLB Register Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| TM | 13:12 | Trace Mode. This field determines how the trace memory is filled when using the simple-break control in the PDtrace™ interface to start or stop trace.<br><br>TM / Trace Mode:<br>00 – Trace-To<br>01 – Trace-From<br>10 – Reserved<br>11 – Reserved<br><br>In Trace-To mode, the on-chip trace memory is filled, continuously wrapping around and overwriting older Trace Words, as long as there is trace data coming from the core.<br>In Trace-From mode, the on-chip trace memory is filled from the point that *PDO_IamTracing* is asserted, and until the on-chip trace memory is full.<br>In both cases, de-asserting the EN bit in this register will also stop fill to the trace memory.<br>If a *TCBTRIGx* trigger control register is used to start/stop tracing, then this field should be set to Trace-To mode.<br>This bit is reserved if on-chip memory is not implemented. | R/W | 0 |
| TLSIF | 11 | When set, this indicates to the TCB that information about Load and Store data cache miss, instruction cache miss, and function call are to be taken from the PDtrace interface and trace them out in the appropriate trace formats as the three optional bits LSm, Im, and Fcr. | R/W | 0 |
| CR | 10:8 | Off-chip Clock Ratio. Writing this field, sets the ratio of the core clock to the off-chip trace memory interface clock. The clock-ratio encoding is shown in Table 11.33.<br>**Remark:** As the Probe interface works in double data rate (DDR) mode, a 1:2 ratio indicates one data packet sent per core clock rising edge.<br>This bit is reserved if off-chip trace option is not implemented. | R/W | 100 |

**Table 11.32 TCBCONTROLB Register Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| Cal | 7 | Calibrate off-chip trace interface.<br>If set to one, the off-chip trace pins will produce the following pattern in consecutive trace clock cycles. If more than 4 data pins exist, the pattern is replicated for each set of 4 pins. The pattern repeats from top to bottom until the Cal bit is de-asserted.<br>**Note:** The clock source of the TCB and PIB must be running.<br>This bit is reserved if off-chip trace option is not implemented. | R/W | 0 |
| TWSrcVal | 6:3 | These bits are used to indicate the value of the TW source field that will be traced if TWSrcWidth indicates a source bit field width of 2 or 4 bits. Note that if the field is 2 bits, then only bits 4:3 of this field will be used in the TW. | R | 0 |
| CA | 2 | Cycle accurate trace.<br>When set to 1, the trace will include stall information.<br>When set to 0, the trace will exclude stall information, and remove bit zero from all transmitted TF's.<br>The stall information included/excluded is:<br>• TF6 formats with TCBcode 0001 and 0101.<br>• All TF1 formats. | R/W | 0 |
| OfC | 1 | If set to 1, trace is sent to off-chip memory using *TR_DATA* pins.<br>If set to 0, trace info is sent to on-chip memory.<br>This bit is read only if a single memory option exists (either off-chip or on-chip only). | R/W | Preset |

Calibrations pattern

| This pattern is replicated for every 4 bits of *TR_DATA* pins. | 3 | 2 | 1 | 0 |
|---|---|---|---|---|
| | 0 | 0 | 0 | 0 |
| | 1 | 1 | 1 | 1 |
| | 0 | 0 | 0 | 0 |
| | 0 | 1 | 0 | 1 |
| | 1 | 0 | 1 | 0 |
| | 1 | 0 | 0 | 0 |
| | 0 | 1 | 0 | 0 |
| | 0 | 0 | 1 | 0 |
| | 0 | 0 | 0 | 1 |
| | 1 | 1 | 1 | 0 |
| | 1 | 1 | 0 | 1 |
| | 1 | 0 | 1 | 1 |
| | 0 | 1 | 1 | 1 |

**Table 11.32 TCBCONTROLB Register Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| EN | 0 | Enable trace.<br>This is the master enable for trace to be generated from the TCB. This bit can be set or cleared, either by writing this register or from a start/stop/about trigger.<br>When set to 1, trace information is sampled on the *PDO_** pins. Trace Words are generated and sent to either on-chip memory or to the Trace Probe. The target of the trace is selected by the OfC bit.<br>When set to 0, trace information on the *PDO_** pins is ignored. A potential TF6-stop (from a stop trigger) is generated as the last information, the TCB pipe-line is flushed, and trace output is stopped. | R/W | 0 |

**Table 11.33 Clock Ratio encoding of the CR field**

| CR/CRMin/CRMax | Clock Ratio |
|---|---|
| 000 | 8:1 (Trace clock is eight times that of core clock) |
| 001 | 4:1 (Trace clock is four times that of core clock) |
| 010 | 2:1 (Trace clock is double that of core clock) |
| 011 | 1:1 (Trace clock is same as core clock) |
| 100 | 1:2 (Trace clock is one half of core clock) |
| 101 | 1:4 (Trace clock is one fourth of core clock) |
| 110 | 1:6 (Trace clock is one sixth of core clock) |
| 111 | 1:8 (Trace clock is one eighth of core clock) |

## 11.9.3 TCBDATA Register

The *TCBDATA* register (0x12) is used to access the registers defined by the $TCBCONTROLB_{REG}$ field; see Table 11.30. Regardless of which register or data entry is accessed through *TCBDATA*, the register is only written if the $TCBCONTROLB_{WR}$ bit is set. For read-only registers, the $TCBCONTROLB_{WR}$ is a don't care.

The format of the *TCBDATA* register is shown below, and the field is described in Table 11.34. The width of *TCBDATA* is 64 bits when on-chip trace words (TWs) are accessed (*TCBTW* access).

**Figure 11.26 TCBDATA Register Format**

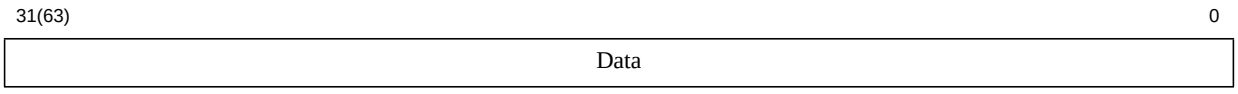


**Table 11.34 TCBDATA Register Field Descriptions**

| Fields | | Description | Read/Write | Reset State |
|---|---|---|---|---|
| Names | Bits | | | |
| Data | 31:0<br>63:0 | Register fields or data as defined by the *TCBCONTROLB*$_{REG}$ field | Only writable if *TCBCONTROLB*$_{WR}$ is set | 0 |

## 11.9.4 TCBCONTROLC Register

The trace output from the processor on the PDtrace interface can be controlled by the trace input signals to the processor from the TCB. The TCB uses a control register, TCBCONTROLC, whose values are used to change the signal values on the PDtrace input interface. External software (i.e., debugger), can therefore manipulate the trace output by writing the TCBCONTROLC register.

The *TCBCONTROLC* register is written by an EJTAG TAP controller instruction, *TCBCONTROLC* (0x13).

The format of the *TCBCONTROLC* register is shown below, and the fields are described in Table 11.31.

**Figure 11.27 TCBCONTROLC Register Format**

| 31 28 | 27 23 | 22 | 21 15 | 14 | 13 | 12 9 | 8 5 | 4 2 | 1 0 |
|---|---|---|---|---|---|---|---|---|---|
| Res | Mode | CPUvalid | Res | CPUid | TCvalid | Res | TCnum | TCbits | MTtraceType | MTtraceTC |

**Table 11.35 TCBCONTROLC Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| Res | 31:28 | Reserved for future use. Must be written as zero; returns zero on read. | 0 | 0 |
| Mode | 27:23 | When tracing is turned on, this signal specifies what information is to be traced by the core. It uses 5 bits, where each bit turns on a tracing of a specific tracing mode.<br><br>**Bit # Set / Trace The Following**<br>0 PC<br>1 Load address<br>2 Store address<br>3 Load data<br>4 Store data<br><br>The table shows what trace value is turned on when that bit value is a 1. If the corresponding bit is 0, then the Trace Value shown in column two is not traced by the processor.<br>On the 34K core PC tracing is always enabled, regardless of the value on bit 23.<br>This field defines the value on the *PDI_TraceMode* signal. | R/W | 0 |

**Table 11.35 TCBCONTROLC Register Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| CPUvalid | 22 | This bit enables VPE based tracing. This bit is ignored if TCvalid field is set<br>0: Instructions for all VPEs are traced<br>1: Instructions from only one VPE specified in CPUId field are traced<br>This field defines the value on the PDI_CPUIdValid signal | R/W | 0 |
| Res | 21:15 | Reserved for future use. | R/W | 0 |
| CPUId | 14 | This bit indicates the value of the VPEid to be traced if CPUValid field is set<br>0: Instructions from VPE0 are traced<br>1: Instructions from VPE1 are traced<br>This field defines the value on the PDI_CPUId signal | R/W | Undefined |
| TCvalid | 13 | This bit enables TC based tracing<br>0: Instructions are traced based on CPUValid/CPUId settings<br>1: Instructions from only one TC specified in TCnum field are traced<br>This field defines the value on the PDI_TCNumValid | R/W | 0 |
| Res | 12:9 | Reserved for future use. | R/W | Undefined |
| TCnum | 8:5 | This field indicates the value of the TC to be traced if TCvalid is set<br>This field defines the value on the PDI_TCNum signal | R/W | Undefined |
| TCbits | 4:2 | This value is used by the TCB to determine the number of bits needed to represent TC value in a Trace Format(TF). Returns 3 on reads indicating 4 bits are needed to represent 9 TC value. | R | Preset |
| MTtrace-Type | 1 | This bit indicates the type of implemented multi-threading<br>0: Fine grained, i.e., switch threads every cycle. If MTtraceTC field is set then each Trace format is augmented by TC information<br>1: Coarse-grained, also known as block multi-threading. If MTtraceTC field is set then TF7 is used and each TF is not augmented.<br>Returns 0 on read indicating that processor may switch threads every cycle if needed. | 0 | Preset |
| MTtraceTC | 0 | This bit controls TC value tracing.<br>0: TC value is not traced<br>1: TC value is tracing by augmenting TCId on each Trace format | R/W | Undefined |

## 11.9.5 TCBCONFIG Register (Reg 0)

The *TCBCONFIG* register holds information about the hardware configuration of the TCB. The format of the *TCBCONFIG* register is shown below, and the field is described in Table 11.36.

**Figure 11.28 TCBCONFIG Register Format**

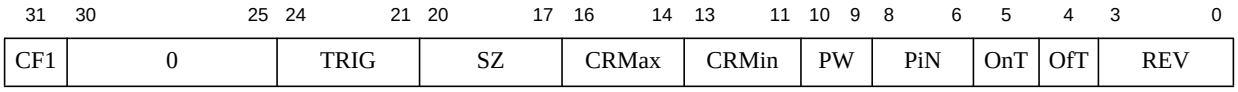


**Table 11.36 TCBCONFIG Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| CF1 | 31 | This bit is set if a *TCBCONFIG1* register exists. In this revision, *TCBCONFIG1* does not exist and this bit always reads zero. | R | 0 |
| 0 | 30:25 | Reserved. Must be written as zero; returns zero on read. | R | 0 |
| TRIG | 24:21 | Number of triggers implemented. This also indicates the number of *TCBTRIGx* registers that exist. | R | Preset<br>Legal values are 0 - 8 |
| SZ | 20:17 | On-chip trace memory size. This field holds the encoded size of the on-chip trace memory.<br>The size in bytes is given by $2^{(SZ+8)}$, implying that the minimum size is 256 bytes and the largest is 8Mb.<br>This bit is reserved if on-chip memory is not implemented. | R | Preset |
| CRMax | 16:14 | Off-chip Maximum Clock Ratio.<br>This field indicates the maximum ratio of the core clock to the off-chip trace memory interface clock. The clock-ratio encoding is shown in Table 11.33.<br>This bit is reserved if off-chip trace option is not implemented. | R | Preset |
| CRMin | 13:11 | Off-chip Minimum Clock Ratio.<br>This field indicates the minimum ratio of the core clock to the off-chip trace memory interface clock.The clock-ratio encoding is shown in Table 11.33.<br>This bit is reserved if off-chip trace option is not implemented. | R | Preset |
| PW | 10:9 | Probe Width: Number of bits available on the off-chip trace interface *TR_DATA* pins. The number of TR_DATA pins is encoded, as shown in the table.<br>**PW** / **Number of bits used on *TR_DATA***<br>00 — 4 bits<br>01 — 8 bits<br>10 — 16 bits<br>11 — reserved<br>This field is preset based on input signals to the TCB and the actual capability of the TCB.<br>This bit is reserved if off-chip trace option is not implemented. | R | Preset |
| PiN | 8:6 | Pipe number.<br>Indicates the number of execution pipelines. | R | 0 |
| OnT | 5 | When set, this bit indicates that on-chip trace memory is present. This bit is preset based on the selected option when the TCB is implemented. | R | Preset |
| OfT | 4 | When set, this bit indicates that off-chip trace interface is present. This bit is preset based on the selected option when the TCB is implemented, and on the existence of a PIB module (*TC_PibPresent* asserted). | R | Preset |
| REV | 3:0 | Revision of TCB. An implementation that conforms to PDtrace version 4.x must has a value of 1 for this field. | R | 1 |

## 11.9.6 TCBTW Register (Reg 4)

The *TCBTW* register is used to read Trace Words from the on-chip trace memory. The TW read is the one pointed to by the *TCBRDP* register. A side effect of reading the *TCBTW* register is that the *TCBRDP* register increments to the next TW in the on-chip trace memory. If *TCBRDP* is at the max size of the on-chip trace memory, the increment wraps back to address zero.

This register is reserved if on-chip trace memory is not implemented.

The format of the *TCBTW* register is shown below, and the field is described in Table 11.37.

**Figure 11.29 TCBTW Register Format**

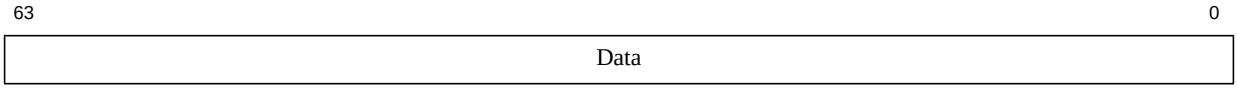



**Table 11.37 TCBTW Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Names | Bits | | | |
| Data | 63:0 | Trace Word | R/W | 0 |

## 11.9.7 TCBRDP Register (Reg 5)

The *TCBRDP* register is the address pointer to on-chip trace memory. It points to the TW read when reading the *TCBTW* register. When writing the $TCBCONTROLB_{RM}$ bit to 1, this pointer is reset to the current value of *TCBSTP*.

This register is reserved if on-chip trace memory is not implemented.

The format of the *TCBRDP* register is shown below, and the field is described in Table 11.38. The value of n depends on the size of the on-chip trace memory. As the address points to a 64-bit TW, lower three bits are always zero.

**Figure 11.30 TCBRDP Register Format**

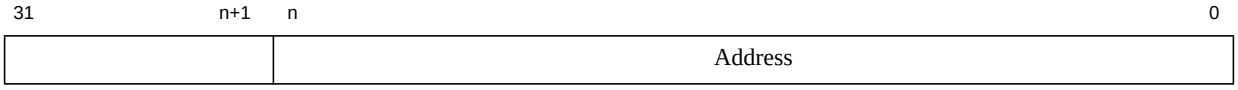



**Table 11.38 TCBRDP Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Names | Bits | | | |
| Data | 31:(n+1) | Reserved. Must be written zero, reads back zero. | 0 | 0 |
| Address | n:0 | Byte address of on-chip trace memory word. | R/W | 0 |

## 11.9.8 TCBWRP Register (Reg 6)

The *TCBWRP* register is the address pointer to on-chip trace memory. It points to the location where the next new TW for on-chip trace will be written.

This register is reserved if on-chip trace memory is not implemented.

The format of the *TCBWRP* register is shown below, and the fields are described in Table 11.39. The value of n depends on the size of the on-chip trace memory. As the address points to a 64-bit TW, the lower three bits are always zero.

**Figure 11.31 TCBWRP Register Format**

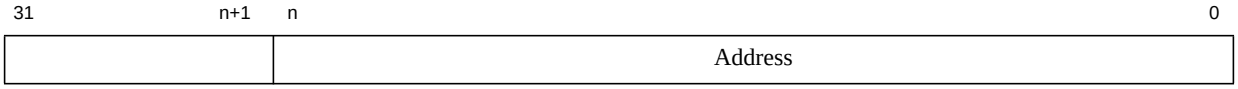



**Table 11.39 *TCBWRP* Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Names | Bits | | | |
| Data | 31:(n+1) | Reserved. Must be written zero, reads back zero. | 0 | 0 |
| Address | n:0 | Byte address of on-chip trace memory word. | R/W | 0 |

## 11.9.9 TCBSTP Register (Reg 7)

The *TCBSTP* register is the start pointer register. This register points to the on-chip trace memory address at which the oldest TW is located. This pointer is reset to zero when the $TCBCONTROLB_{TR}$ bit is written to 1. If a continuous trace to on-chip memory wraps around the on-chip memory, *TSBSTP* will have the same value as *TCBWRP*.

This register is reserved if on-chip trace memory is not implemented.

The format of the *TCBSTP* register is shown below, and the fields are described in Table 11.40. The value of n depends on the size of the on-chip trace memory. As the address points to a 64-bit TW, lower three bits are always zero.

**Figure 11.32 TCBSTP Register Format**

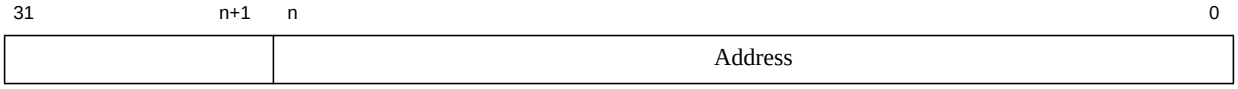



**Table 11.40 TCBSTP Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Names | Bits | | | |
| Data | 31:(n+1) | Reserved. Must be written zero, reads back zero. | 0 | 0 |
| Address | n:0 | Byte address of on-chip trace memory word. | R/W | 0 |

### 11.9.10 TCBTRIGx Register (Reg 16-23)

Up to eight Trigger Control registers are possible. Each register is named *TCBTRIGx*, where *x* is a single digit number from 0 to 7 (*TCBTRIG0* is Reg 16). The actual number of trigger registers implemented is defined in the $TCBCONFIG_{TRIG}$ field. An unimplemented register will read all zeros and writes are ignored.

Each Trigger Control register controls when an associated trigger is fired, and the action to be taken when the trigger occurs. Please also read Section 11.11 "TCB Trigger Logic", for detailed description of trigger logic issues.

The format of the *TCBTRIGx* register is shown below, and the fields are described in Table 11.41.

**Figure 11.33 TCBTRIGx Register Format**

| 31 24 | 23 | 22 16 | 15 | 14 | 13 7 | 6 | 5 | 4 | 3 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| TCBinfo | Trace | 0 | CHTro | PDTro | 0 | DM | CHTri | PDTri | Type | FO | TR |

**Table 11.41 TCBTRIGx Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Names | Bits | | | |
| TCBinfo | 31:24 | TCBinfo to be used in a possible TF6 trace format when this trigger fires. | R/W | 0 |
| Trace | 23 | When set, generate TF6 trace information when this trigger fires. Use TCBinfo field for the TCBinfo of TF6 and use Type field for the two MSB of the TCBtype of TF6. The two LSB of TCBtype are 00. The write value of this bit always controls the behavior of this trigger.<br>When this trigger fires, the read value will change to indicate if the TF6 format was ever suppressed by a simultaneous trigger. If so, the read value will be 0. If the write value was 0, the read value is always 0. This special read value is valid until the *TCBTRIGx* register is written. | R/W | 0 |
| 0 | 22:16 | Reserved. Must be written as zero; returns zero on read. | R | 0 |
| CHTro | 15 | When set, generate a single cycle strobe on *TC_ChipTrigOut* when this trigger fires. | R/W | 0 |
| PDTro | 14 | When set, generate a single cycle strobe on *TC_ProbeTrigOut* when this trigger fires. | R/W | 0 |
| 0 | 13:7 | Reserved. Must be written as zero; returns zero on read. | R | 0 |
| DM | 6 | When set, this Trigger will fire when a rising edge on the Debug mode indication from the core is detected.<br>The write value of this bit always controls the behavior of this trigger.<br>When this trigger fires, the read value will change to indicate if this source was ever the cause of a trigger action (even if the action was suppressed). If so the read value will be 1. If the write value was 0 the read value is always 0. This special read value is valid until the *TCBTRIGx* register is written. | R/W | 0 |

**Table 11.41 TCBTRIGx Register Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Names | Bits | | | |
| CHTri | 5 | When set, this Trigger will fire when a rising edge on *TC_ChipTrigIn* is detected.<br>The write value of this bit always controls the behavior of this trigger.<br>When this trigger fires, the read value will change to indicate if this source was ever the cause of a trigger action (even if the action was suppressed). If so the read value will be 1. If the write value was 0 the read value is always 0. This special read value is valid until the *TCBTRIGx* register is written. | R/W | 0 |
| PDTri | 4 | When set, this Trigger will fire when a rising edge on *TC_ProbeTrigIn* is detected.<br>The write value of this bit always controls the behavior of this trigger.<br>When this trigger fires, the read value will change to indicate if this source was ever the cause of a trigger action (even if the action was suppressed). If so the read value will be 1. If the write value was 0 the read value is always 0. This special read value is valid until the *TCBTRIGx* register is written. | R/W | 0 |
| Type | 3:2 | Trigger Type: The Type indicates the action to take when this trigger fires. The table below show the Type values and the Trigger action.<br><br>**Type** / **Trigger action**<br>00 — **Trigger Start:** Trigger start-point of trace.<br>01 — **Trigger End:** Trigger end-point of trace.<br>10 — **Trigger About:** Trigger center-point of trace.<br>11 — **Trigger Info:** No action trigger, only for trace info.<br><br>The actual action is to set or clear the $TCBCONTROLB_{EN}$ bit. A Start trigger will set $TCBCONTROLB_{EN}$, a End trigger will clear $TCBCONTROLB_{EN}$. The About trigger will clear $TCBCONTROLB_{EN}$ half way through the trace memory, from the trigger. The size determined by the $TCBCONFIG_{SZ}$ field for on-chip memory. Or from the $TCBCONTROLA_{SyP}$ field for off-chip trace.<br>If Trace is set, then a TF6 format is added to the trace words. For Start and Info triggers this is done before any other TF's in that same cycle. For End and About triggers, the TF6 format is added after any other TF's in that same cycle.<br>If the $TCBCONTROLB_{TM}$ field is implemented it must be set to Trace-To mode (00), for the Type field to control on-chip trace fill.<br>The write value of this bit always controls the behavior of this trigger.<br>When this trigger fires, the read value will change to indicate if the trigger action was ever suppressed. If so the read value will be 11. If the write value was 11 the read value is always 11. This special read value is valid until the *TCBTRIGx* register is written. | R/W | 0 |
| FO | 1 | Fire Once. When set, this trigger will not re-fire until the TR bit is de-asserted. When de-asserted this trigger will fire each time one of the trigger sources indicates trigger. | R/W | 0 |

**Table 11.41 TCBTRIGx Register Field Descriptions (Continued)**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Names | Bits | | | |
| TR | 0 | Trigger happened. When set, this trigger fired since the TR bit was last written 0.<br>This bit is used to inspect whether the trigger fired since this bit was last written zero.<br>When set, all the trigger source bits (bit 4 to 13) will change their read value to indicate if the particular bit was the source to fire this trigger. Only enabled trigger sources can set the read value, but more than one is possible.<br>Also when set the Type field and the Trace field will have read values which indicate if the trigger action was ever suppressed by a higher priority trigger. | R/W0 | 0 |

### 11.9.11 Register Reset State

Reset state for all register fields is entered when either of the following occur:

1. TAP controller enters/is in Test-Logic-Reset state.
2. *EJ_TRST_N* input is asserted low.

## 11.10 Enabling MIPS Trace

As there are several ways to enable tracing, it can be quite confusing to figure out how to turn tracing on and off. This section should help clarify the enabling of trace.

### 11.10.1 Trace Trigger from EJTAG Hardware Instruction/Data Breakpoints

If hardware instruction/data simple breakpoints are implemented in the 34K core, then these breakpoint can be used as triggers to start/stop trace. When used for this, the breakpoints need not also generate a debug exception, but are capable of only generating an internal trigger to the trace logic. This is done by only setting the TE bit and not the BE bit in the Breakpoint Control register. Please see Section 11.2.8.5 "Instruction Breakpoint Control n (IBCn) Register" and Section 11.2.9.5 "Data Breakpoint Control n (DBCn) Register", for details on breakpoint control.

In connection with the breakpoints, the Trace BreakPoint Control (*TraceBPC*) register is used to define the trace action when a trigger happens. When a breakpoint is enabled as a trigger (TE = 1), it can be selected to be either a start or a stop trigger to the trace logic. Please see Section 7.2.49 "TraceIBPC Register (CP0 Register 23, Select 4)" for detail in how to define a start/stop trigger.

### 11.10.2 Turning On PDtrace™ Trace

Trace enabling and disabling from software is similar to the hardware method, with the exception that the bits in the control register are used instead of the input enable signals from the TCB. The $TraceControl_{TS}$ bit controls whether hardware (via the TCB), or software (via the *TraceControl* register) controls tracing functionality.

Trace is turned on when the following expression evaluates true:

(

```
    (
        (TraceControl_TS and TraceControl_On) or
        ((not TraceControl_TS) and TCBCONTROLA_On)
    )
    and
    (MatchEnable or TriggerEnable)
)
```

where,

```
MatchEnable ←
(
    TraceControl_TS
    and
        ((TraceControl2_TCV and (TraceControl2_TCNUM equal TCIDofCompletedInst)) or
         ((not TraceControl2_TCV) and TraceControl2_CPUIdV and
            (TraceControl2_CPUId equal VPEIDofCompletedInst )) or
         (TraceControl2_TCV nor TraceControl2_CPUIdV ))
    and
    (
       TraceControl_G or
       (((TraceControl_ASID xor EntryHi_ASID) and (not TraceControl_ASID_M)) = 0)
    )
    and
    (
        (TraceControl_U and UserMode)      or
        (TraceControl_S and SupervisorMode) or
        (TraceControl_K and KernelMode)    or
        (TraceControl_E and ExceptionMode) or
        (TraceControl_D and DebugMode)
    )
)
or
(
    (not TraceControl_TS)
    and
        ((TCBCONTROLC_TCV and (TCBCONTROLC_TCNUM equal TCIDofCompletedInst)) or
         ((not TCBCONTROLC_TCV) and TCBCONTROLC_CPUIdV and
            (TCBCONTROLC_CPUId equal VPEIDofCompletedIns )) or
         (TCBCONTROLC_TCV nor TCBCONTROLC_CPUIdV ))
    and
    (TCBCONTROLA_G or (TCBCONTROLA_ASID = EntryHi_ASID))
    and
    (
        (TCBCONTROLA_U and UserMode)       or
        (TCBCONTROLA_S and SupervisorMode) or
        (TCBCONTROLA_K and KernelMode)     or
        (TCBCONTROLA_E and ExceptionMode)  or
        (TCBCONTROLA_DM and DebugMode)
    )
)
```

and where,

```
TriggerEnable ←
(
    DBCi_TE        and
    DBS_BS[i]      and
```

```
    TraceBPC_DE     and
    (TraceBPC_DBPOn[i] = 1)
)
or
(
    IBCi_TE         and
    IBS_BS[i]       and
    TraceBPC_IE     and
    (TraceBPC_IBPOn[i] = 1)
)
```

As seen in the expression above, trace can be turned on only if the master switch *TraceControl*$_{On}$ or *TCBCONTROLA*$_{On}$ is first asserted.

Once this is asserted, there are two ways to turn on tracing. The first way, the *MatchEnable* expression, uses the input enable signals from the TCB or the bits in the *TraceControl* register. This tracing is done over general program areas. For example, all of the user-level code for a particular process (if ASID is specified), and so on.

The second way to turn on tracing, the *TriggerEnable* expression, is from the processor side using the EJTAG hardware breakpoint triggers. If EJTAG is implemented, and hardware breakpoints can be set, then using this method enables finer grain tracing control. It is possible to send a trigger signal that turns on tracing at a particular instruction. For example, it would be possible to trace a single procedure in a program by triggering on trace at the first instruction, and triggering off trace at the last instruction.

The easiest way to unconditionally turn on trace is to assert either hardware or software tracing and the corresponding trace on signal with other enables. For example, with *TraceControl*$_{TS}$=0, i.e., hardware controlled tracing, assert *TCBCONTROLA*$_{On}$, *TCBCONTROLA*$_{G}$, and all the other signals in the second part of expression *MatchEnable*. To only trace when a particular process with a known ASID is executing, assert *TCBCONTROLA*$_{On}$, the correct *TCBCONTROLA*$_{ASID}$ value, and all of *TCBCONTROLA*$_{U}$, *TCBCONTROLA*$_{K}$, *TCBCONTROLA*$_{E}$, and *TCBCONTROLA*$_{DM}$. (If it is known that the particular process is a user-level process, then it would be sufficient to only assert *TCBCONTROLA*$_{U}$ for example). When using the EJTAG hardware triggers to turn trace on and off, it is best if *TCBCONTROLA*$_{On}$ is asserted and all the other processor mode selection bits in *TCBCONTROLA* are turned off. This would be the least confusing way to control tracing with the trigger signals. Tracing can be controlled via software with the *TraceControl* register in a similar manner.

### 11.10.3 Turning Off PDtrace™ Trace

Trace is turned off when the following expression evaluates true:

```
(
    (TraceControl_TS and (not TraceControl_On)) or
    ((not TraceControl_TS) and (not TCBCONTROLA_On))
)
or
(
    (not MatchEnable)       and
    (not TriggerEnable)     and
    TriggerDisable
)
```

where,

```
TriggerDisable ←
```

```
(
    DBCi_TE          and
    DBS_BS[i]        and
    TraceBPC_DE      and
    (TraceBPC_DBPOn[i] = 0)
)
or
(
    IBCi_TE          and
    IBS_BS[i]        and
    TraceBPC_IE      and
    (TraceBPC_IBPOn[i] = 0)
)
```

Tracing can be unconditionally turned off by de-asserting the *TraceControl*$_{On}$ bit or the *TCBCONTROLA*$_{On}$ signal. When either of these are asserted, tracing can be turned off if all of the enables are de-asserted, irrespective of the TraceControl$_{G}$ bit (*TCBCONTROLA*$_{G}$) and TraceControl$_{ASID}$ (*TCBCONTROLA*$_{ASID}$) values. EJTAG hardware breakpoints can be used to trigger trace off as well. Note that if simultaneous triggers are generated, and even one of them turns on tracing, then even if all of the others attempt to trigger trace off, then tracing will still be turned on. This condition is reflected in presence of the “(not TriggerEnable)” term in the expression above.

### 11.10.4 TCB Trace Enabling

The TCB must be enabled in order to produce a trace on the probe or to on-chip memory, when trace information is sent on the PDtrace™ interface. The main switch for this is the *TCBCONTROLB*$_{EN}$ bit. When set, the TCB will send trace information to either on-chip trace memory or to the Trace Probe, controlled by the setting of the *TCBCONTROLB*$_{OfC}$ bit.

The TCB can optionally include trigger logic, which can control the *TCBCONTROLB*$_{EN}$ bit. Please see Section 11.11 “TCB Trigger Logic” for details.

### 11.10.5 Tracing a Reset Exception

Tracing a reset exception is possible. However, the *TraceControl*$_{TS}$ bit is reset to 0 at core reset, so all the trace control must be from the TCB (using *TCBCONTROLA* and *TCBCONTROLB*). The PDtrace fifo and the entire TCB are reset based on an EJTAG reset. It is thus possible to set up the trace modes, etc., using the TAP controller, and then reset the processor core.

## 11.11 TCB Trigger Logic

The TCB is optionally implemented with trigger unit. If this is the case, then the *TCBCONFIG*$_{TRIG}$ field is non-zero. This section will explain some of the issues around triggers in the TCB.

### 11.11.1 Trigger Units Overview

TCB trigger logic features three main parts:

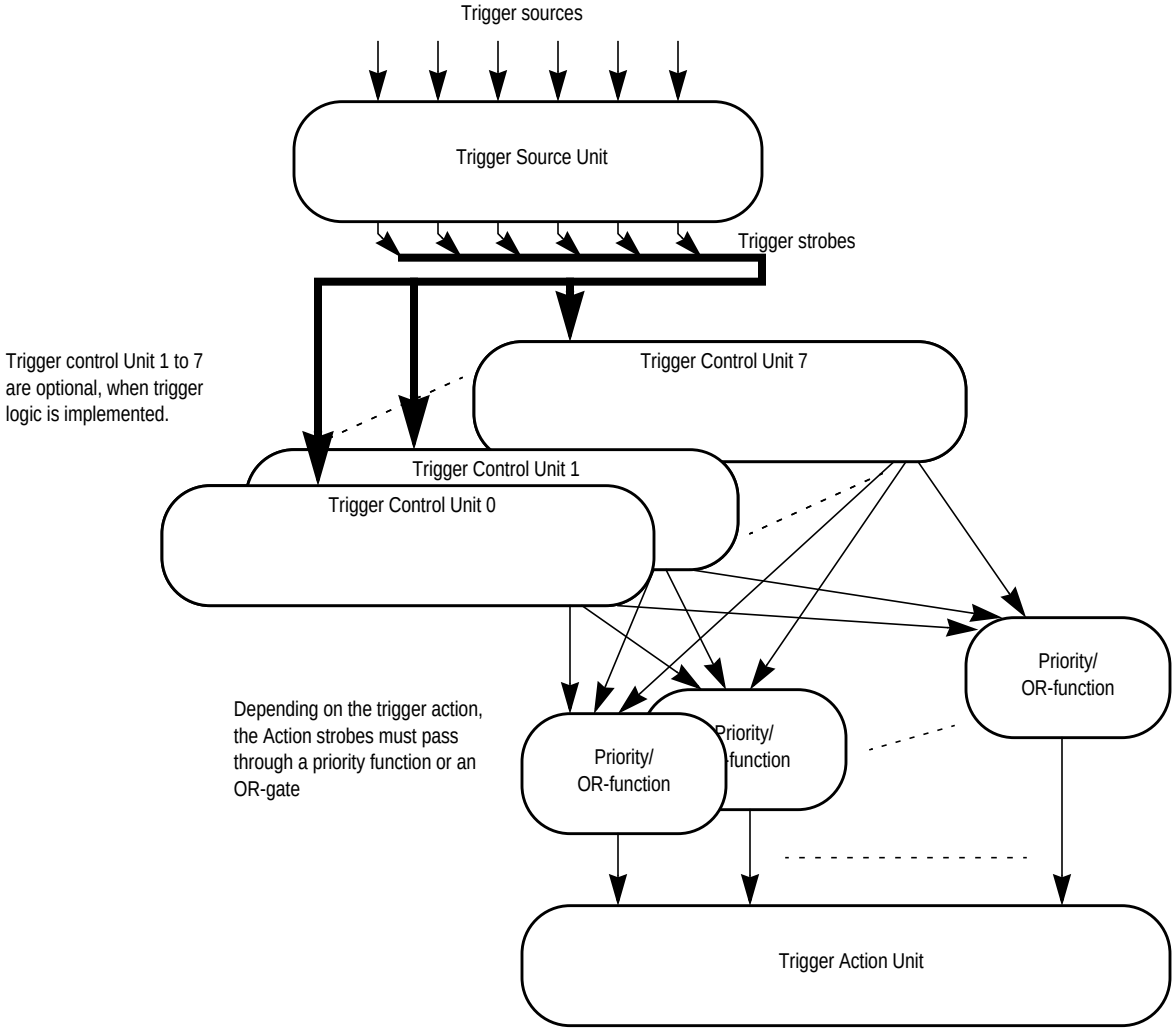
1. A common Trigger Source detection unit.

2. 1 to 8 separate Trigger Control units.

3. A common Trigger Action unit.

Figure 11.34 show the functional overview of the trigger flow in the TCB.

**Figure 11.34 TCB Trigger Processing Overview**




## 11.11.2 Trigger Source Unit

The TCB has three trigger sources:

1. Chip-level trigger input (*TC_ChipTrigIn*).
2. Probe trigger input (*TR_TRIGIN)*.
3. Debug Mode (DM) entry indication from the processor core.

The input triggers are all rising-edge triggers, and the Trigger Source Units convert the edge into a single cycle strobe to the Trigger Control Units.

### 11.11.3 Trigger Control Units

Up to eight Trigger Control Units are possible. Each of them has its own Trigger Control Register (*TCBTRIGx, x={0..7}*). Each of these registers controls the trigger fire mechanism for the unit. Each unit has all of the Trigger Sources as possible trigger event and they can fire one or more of the Trigger Actions. This is all defined in the Trigger Control register *TCBTRIGx* (see Section 11.9.10 "TCBTRIGx Register (Reg 16-23)").

### 11.11.4 Trigger Action Unit

The TCB has four possible trigger actions:

1. Chip-level trigger output (*TC_ChipTrigOut*).
2. Probe trigger output (*TR_TRIGOUT*).
3. Trace information. Put a programmable byte into the trace stream from the TCB.
4. Start, End or About (delayed end) control of the $TCBCONTROLB_{EN}$ bit.

The basic function of the trigger actions is explained in Section 11.9.10 "TCBTRIGx Register (Reg 16-23)". Please also read the next Section 11.11.5 "Simultaneous Triggers".

### 11.11.5 Simultaneous Triggers

Two or more triggers can fire simultaneously. The resulting behavior depends on trigger action set for each of them, and whether they should produce a TF6 trace information output or not. There are two groups of trigger actions: Prioritized and OR'ed.

#### 11.11.5.1 Prioritized Trigger Actions

For prioritized simultaneous trigger actions, the trigger control unit which has the lowest number takes precedence over the higher numbered units. The *x* in *TCBTRIGx* registers defines the number. The oldest trigger takes precedence over everything.

The following trigger actions are prioritized when two or more units fire simultaneously:

- Trigger Start, End and About type triggers ($TCBTRIGx_{Type}$ field set to 00, 01 or 10), which will assert/de-assert the $TCBCONTROLB_{EN}$ bit. The About trigger is delayed and will always change $TCBCONTROLB_{EN}$ because it is the oldest trigger when it de-asserts $TCBCONTROLB_{EN}$. An About trigger will not start the countdown if an even older About trigger is using the Trace Word counter.
- Triggers which produce TF6 trace information in the trace flow (Trace bit is set).

Regardless of priority, the $TCBTRIGx_{TR}$ bit is set when the trigger fires. This is so even if a trigger action is suppressed by a higher priority trigger action. If the trigger is set to only fire once (the $TCBTRIGx_{FO}$ bit is set), then the suppressed trigger action will not happen until after $TCBTRIGx_{TR}$ is written 0.

If a Trigger action is suppressed by a higher priority trigger, then the read value, when the $TCBTRIGx_{TR}$ bit is set, for the $TCBTRIGx_{Trace}$ field will be 0 for suppressed TF6 trace information actions. The read value in the $TCBTRIGx_{Type}$ field for suppressed Start/End/About triggers will be 11. This indication of a suppressed action is

sticky. If any of the two actions (Trace and Type) are ever suppressed for a multi-fire trigger (the *TCBTRIGx*$_{FO}$ bit is zero), then the read values in Trace and/or Type are set to indicate any suppressed action.

#### *About Trigger*

The About triggers delayed de-assertion of the *TCBCONTROLB*$_{EN}$ bit is always executed, regardless of priority from another Start trigger at the time of the *TCBCONTROLB*$_{EN}$ change. This means that if a simultaneous About trigger action on the *TCBCONTROLB*$_{EN}$ bit (n/2 Trace Words after the trigger) and a Start trigger hit the same cycle, then the About trigger wins, regardless of which trigger number it is. The oldest trigger takes precedence.

However, if an About trigger has started the count down from n/2, but not yet reached zero, then a new About trigger, will NOT be executed. Only one About trigger can have the cycle counter. This second About trigger will store 11 in the *TCBTRIGx*$_{Type}$ field. But, if the *TCBTRIGx*$_{Trace}$ bit is set, a TF6 trace information will still go in the trace.

### 11.11.5.2 OR'ed Trigger Actions

The simple trigger actions CHTro and PDTro from each trigger unit, are effectively OR'ed together to produce the final trigger. One or more expected trigger strobes on i.e. *TC_ChipTrigOut* can thus disappear. External logic should not rely on counting of strobes, to predict a specific event, unless simultaneous triggers are known not to occur.

## 11.12 MIPS Trace Cycle-by-Cycle Behavior

A key reason for using trace, and not single stepping to debug a software problem, is often to get a picture of the real-time behavior. However the trace logic itself can, when enabled, affect the exact cycle-by-cycle behavior,

### 11.12.1 FIFO Logic in PDtrace and TCB Modules

Both the PDtrace module and the TCB module contain a fifo. This might seem like extra overhead, but there are good reasons for this. The vast majority of the information compression happens in the PDtrace module. Any data information, like PC and load/store address values (delta or full), load/store data and processor mode changes, are all sent on the same 32-bit data bus to the TCB on the internal PDtrace™ interface. When an instruction requires more than 32 bits of information to be traced properly, the PDtrace fifo will buffer the information, and send it on subsequent clock cycles.

In the TCB, the on-chip trace memory is defined as a 64-bit wide synchronous memory running at core-clock speed. In this case the fifo is not needed. For off-chip trace through the Trace Probe, the fifo comes into play, because only a limited number of pins (4, 8 or 16) exist. Also the speed of the Trace Probe interface can be different (either faster or slower) from that of the 34K core. So for off-chip tracing, a specific TCB TW fifo is needed.

### 11.12.2 Handling of FIFO Overflow in the PDtrace Module

Depending on the amount of trace information selected for trace, and the frequency with which the 32-bit data interface is needed, it is possible for the PDtrace fifo overflow from time to time. There are two ways to handle this case:

1. Allow the overflow to happen, and thereby lose some information from the trace data.

2. Prevent the overflow by back-stalling the core, until the fifo has enough empty slots to accept new trace data.

The PDtrace fifo option is controlled by either the *TraceControl*$_{IO}$ or the *TCBCONTROLA*$_{IO}$ bit, depending on the setting of *TraceControl*$_{TS}$ bit.

The first option is free of any cycle-by-cycle change whether trace is turned on or not. This is achieved at the cost of potentially losing trace information. After an overflow, the fifo is completely emptied, and the next instruction is traced as if it was the start of the trace (processor mode and full PC are traced). This guarantees that only the un-traced fifo information is lost.

The second option guarantees that all the trace information is traced to the TCB. In some cases this is then achieved by back-stalling the core pipeline, giving the PDtrace fifo time to empty enough room in the fifo to accept new trace information from a new instruction. This option can obviously change the real-time behavior of the core when tracing is turned on.

If PC trace information is the only thing enabled (in *TraceControl2*$_{MODE}$ or *TCBCONTROLC*$_{MODE}$, depending on the setting of *TraceControl*$_{TS}$), and Trace of all branches is turned off (via *TraceControl*$_{TB}$ or *TCBCONTROLA*$_{TB}$, depending on the setting of *TraceControl*$_{TS}$), then the fifo is unlikely to overflow very often, if at all. This is of course very dependent on the code executed, and the frequency of exception handler jumps, but with this setting there is very little information overhead.

## 11.12.3 Handling of FIFO Overflow in the TCB

The TCB also holds a fifo, used to buffer the TW's which are sent off-chip through the Trace Probe. The data width of the probe can be either 4, 8 or 16 pins, and the speed of these data pins can be from 16 times the core-clock to 1/4 of the core clock (the trace probe clock always runs at a double data rate multiple to the core-clock). See Section 11.12.3.1 "Probe Width and Clock-ratio Settings" for a description of probe width and clock-ratio options. The combination between the probe width (4, 8 or 16) and the data speed, allows for data rates through the trace probe from 256 bits per core-clock cycle down to only 1 bit per core-clock cycle. The high extreme is not likely to be supported in any implementation, but the low one might be.

The data rate is an important figure when the likelihood of a TCB fifo overflow is considered. The TCB will at maximum produce one full 64-bit TW per core-clock cycle. This is true for any selection of trace mode in *TraceControl2*$_{MODE}$ or *TCBCONTROLC*$_{MODE}$. The PDtrace module will guarantee the limited amount of data. If the TCB data rate cannot be matched by the off-chip probe width and data speed, then the TCB fifo can possibly overflow. There is only one way to handle this:

1. Prevent the overflow by asserting a stall-signal back to the core (*PDI_StallSending*). This will in turn stall the core pipeline.

There is no way to guarantee that this back-stall from the TCB is never asserted, unless the effective data rate of the Trace Probe interface is at least 64-bits per core-clock cycle.

As a practical matter, the amount of data to the TCB can be minimized by only tracing PC information and excluding any cycle accurate information. This is explained in Section 11.12.2 "Handling of FIFO Overflow in the PDtrace Module" and below in Section 11.12.4 "Adding Cycle Accurate Information to the Trace". With this setting, a data rate of 8-bits per core-clock cycle is usually sufficient. No guarantees can be given here, however, as heavy interrupt activity can increase the number of unpredictable jumps considerably.

### 11.12.3.1 Probe Width and Clock-ratio Settings

The actual number of data pins (4, 8 or 16) is defined by the *TCBCONFIG*$_{PW}$ field. Furthermore, the frequency of the Trace Probe can be different from the core-clock frequency. The trace clock (*TR_CLK*) is a double data rate clock. This means that the data pins (*TR_DATA*) change their value on both edges of the trace clock. When the trace clock is running at clock ratio of 1:2 (one half) of core clock, the data output registers are running a core-clock frequency. The clock ratio is set in the *TCBCONTROLB*$_{CR}$ field. The legal range for the clock ratio is defined in *TCBCONFIG*$_{CRMax}$ and *TCBCONFIG*$_{CRMin}$ (both values inclusive). If *TCBCONTROLB*$_{CR}$ is set to an unsup-

ported value, the result is UNPREDICABLE. The maximum possible value for *TCBCONFIG*$_{CRMax}$ is 8:1 (*TR_CLK* is running 8 times faster than core-clock). The minimum possible value for *TCBCONFIG*$_{CRMin}$ is 1:8 (*TR_CLK* is running at one eighth of the core-clock). See Table 11.33 for a description of the encoding of the clock ratio fields.

### 11.12.4 Adding Cycle Accurate Information to the Trace

Depending on the trace regeneration software, it is possible to obtain the exact cycle time relationship between each instruction in the trace. This information is added to the trace, when the *TCBCONTROLB*$_{CA}$ bit is set. The overhead on the trace information is a little more than one extra bit per core-clock cycle.

This setting only affects the TCB module and not the PDtrace module. The extra bit therefore only affects the likelihood of the TCB fifo overflowing.

## 11.13 TCB On-Chip Trace Memory

When on-chip trace memory is available (*TCBCONFIG*$_{OnT}$ is set) the memory is typically of smaller size than if it were external in a trace probe. The assumption is that it is of some value to trace a smaller piece of the program.

With on-chip trace memory, the TCB can work in three possible modes:

1. Trace-From mode.
2. Trace-To mode.
3. Under Trigger unit control.

Software can select this mode using the *TCBCONTROLB*$_{TM}$ field. If one or more trigger control registers (*TCBTRIGx*) are implemented, and they are using Start, End or About triggers, then the trace mode in *TCBCONTROLB*$_{TM}$ should be set to Trace-To mode.

### 11.13.1 On-Chip Trace Memory Size

The supported On-chip trace memory size can range from 256 byte to 8Mbytes, in powers of 2. The actual size is shown in the *TCBCONFIG*$_{SZ}$ field.

### 11.13.2 Trace-From Mode

In the Trace-From mode, tracing begins when the processor enters into a processor mode/ASID value which is defined to be traced or when an EJTAG hardware breakpoint trace trigger turns on tracing. Trace collection is stopped when the buffer is full. The TCB then signals buffer full using *TCBCONTROLB*$_{BF}$. When external software polling this register finds the *TCBCONTROLB*$_{BF}$ bit set, it can then read out the internal trace memory. Saving the trace into the internal buffer will re-commence again only when the *TCBCONTROLB*$_{BF}$ bit is reset and if the core is sending valid trace data (i.e., *PDO_IamTracing* not equal 0).

### 11.13.3 Trace-To Mode

In the Trace-To mode, the TCB keeps writing into the internal trace memory, wrapping over and overwriting the oldest information, until the processor is reaches an end of trace condition. End of trace is reached by leaving the processor mode/ASID value which is traced, or when an EJTAG hardware breakpoint trace trigger turns tracing off. At this

point, the on-chip trace buffer is then dumped out in a manner similar to that described above in Section 11.13.2 “Trace-From Mode”.

*Chapter 12*

# Inter-Thread Communication Unit of the 34K™ Core

This chapter describes the 34K Inter-Thread Communication Unit (ITU) included in the 34Kf core. This chapter contains the following sections:



## 12.1 Features Overview

Inter-Thread Communication (ITC) Storage is a Gating Storage mechanism designed for low-level thread synchronization. Loads and stores to and from gating storage may block until the state of the storage location corresponds to some set of conditions required for completion. A blocked load or store can be precisely aborted if necessary, and restarted later on.

In the 34K core, the ITC storage is provided by the Inter-Thread Communication Unit (ITU). This block of logic resides outside of the core and connects to the core through the gating storage interface. The SoC integrators are free to use the MIPS-supplied reference module, to implement their own ITU module, or to not use ITC at all. This chapter describes the features of the sample ITU block supplied with the 34K core. This block only supports synchronization of TCs within a single 34K core.

## 12.2 ITC Storage

References to memory pages which map to ITC storage resolve not to main memory, but to storage locations, or cells, with special attributes. In general, it is possible that behind each ITC storage cell there is more than one memory location. This is useful for mapping hardware queues, stacks, and other structures. The reference ITU supports two kinds of storage cells: four entry FIFO queues and single entry Semaphore cells. All ITC cells are composed of the tag and data portions. In the single entry cells, the data is 32 bits wide. The FIFO cells store four 32 bit data values.Although the memory space allows for 64bit ITC cells, only the least significant 32bit words are present in this implementation. All ITC cells should be accessed as 32bit memory. Partial-word access such as LH or SB will result in undefined behavior.

The tag of each ITC cell contains a number of control bits that regulate accesses to that cell. The format for the ITC tag is shown in Figure 12.1. In addition to the *E* (Empty) and *F* (Full) fields specified by the MT ASE, the tag contains four implementation specific fields: *T, FIFO*, *FIFODepth*, and *FIFOPtr*. The *FIFO* and *FIFODepth* fields indicate whether a cell is a FIFO and what its depth is. The *FIFOPtr* indicates how many elements are currently in a FIFO. This field is always zero for single entry cells. The *FIFOPtr* can be reset by writing 1 into the *E* field of a

FIFO. Finally, the *T* field indicates whether a Gating Storage exception should be signaled on an E/F or P/V view access to the cell.

**Table 12.1 ITC Storage Cell Tag Format**

| Fields | | Description | Read/Write | Reset State |
|---|---|---|---|---|
| Name | Bit | | | |
| FIFODepth | 31:28 | $Log_2$ of the cell depth. This field is set to 0x0 for single entry cells, and to 0x2 for four entry FIFO cells. | R | Preset |
| FIFOPtr | 20:18 | This field indicates the number of elements in a FIFO cell, and always reads zero for single entry Semaphore cells. | R | 0 |
| FIFO | 17 | 1 for FIFO cells and 0 for single entry Semaphore cells. | R | Preset |
| T | 16 | Trap Bit. When set, this bit causes the processor to take a Gating Storage Exception on PV or EF accesses. | R/W | Undefined |
| F | 1 | Full Bit. This bit indicates that the cell is full | R/W | Undefined |
| E | 0 | Empty Bit. This bit indicates that the cell is empty. Writing 1 to this bit also reset FIFOPtr. | R/W | Undefined |
| 0 | 27:21, 15:2 | Must be written as zeros; return zeros on read | 0 | 0 |

The number and type of ITC cells implemented in the ITU is configurable. The possible configurations are: 0, 1, 2, 4, 8, or 16 four entry FIFOs and 0, 1, 2, 4, 8, or 16 single entry Semaphores. The N FIFO cells are located at addresses 0:N-1, and the M Semaphore cells are located at address N:N+M-1.

## 12.3 ITC Views

All ITC cells can be accessed in one of 16 ways, called views, using standard load and store instructions. The view is encoded in bits 6:3 of the memory address, such that the successive views of a cell correspond to successive 64-bit-aligned addresses. Table 12.2 shows the addresses for the various views and the following sections describe the effects of using each of the views If the ITC location is of type FIFO, the behavior of some of the views changes and this is noted in the description of each view.

**Table 12.2 ITC View Addresses**

| Address[6:3] | View |
|---|---|
| 0x0 | Bypass View |
| 0x1 | Control View |
| 0x2 | Empty/Full Synchronized View |
| 0x3 | Empty/Full Try View |
| 0x4 | P/V Synchronized View |
| 0x5 | P/V Try View |
| 0x6-0xF | Reserved Views |

### 12.3.1 Bypass View

This view of the ITC location implies that a load or a store does not cause the issuing thread to block and does not affect any of the cells state bits.The operation of SC using this view is undefined.

Accesses using Bypass view never result in Gating Storage exceptions.

A Bypass view store to a FIFO ITC location overwrites the newest FIFO entry, while a Bypass view load returns the contents of the oldest entry.

### 12.3.2 Control View

This view of the ITC location can be used to manipulate the tag of the ITC cell. Loads and stores access the entire 32b tag value. Table 12.1 shows the fields within that 32b tag.

Accesses using Control view never cause the issuing thread to block and never result in Gating Storage exceptions.

A Control view store to a FIFO location with the E bit set will cause the FIFO to reset its read pointer.

### 12.3.3 Empty/Full Synchronized View

This view of the ITC location implies that a load causes the issuing thread to block if the cell is Empty. Similarly, a store blocks if the cell is full. Accesses using this view cause an automatic update of the Empty and Full bits to reflect the new state of the cell. The operation of SC using this view is undefined.

If the T bit is set, then all E/F Synchronized view accesses, success or failure, cause a gated exception trap.

### 12.3.4 Empty/Full Try View

This view of the ITC location is similar in nature to the previous E/F Synchronized view in most respects other than the waiting policy on an access failure. It is to be used if the issuing thread can potentially find something else to do and does not wish to be blocked if the access fails. A load with this view returns a value of zero if the cell is Empty, regardless of actual data contained. Otherwise the load behaves as in the E/F Synchronized case. Normal Stores to Full locations through the E/F Try view fail silently to update the contents of the cell, rather than block the thread. SC (Store Conditional) instructions referencing the EF Try view will indicate success or failure based on whether the ITC store succeeds or fails.

If the T bit is set, then all E/F Try view accesses, success or failure, cause a gated exception trap.

### 12.3.5 P/V Synchronized View

This view of the ITC location does not modify the Empty and Full bits, both of which are assumed to be cleared as part of the cell initialization routine. Loads with this view return the current cell data value if the value is non-zero, and cause an atomic post-decrement of the value. If the cell value is zero, loads block until the cell takes a non-zero value. Normal Stores cause an atomic increment of the cell value, up to a maximum of 0xffff at which point the value saturates. Loads check the least significant 16bits of the cell for a 0x0 irrespective of load size. The operation of SC using this view is undefined.

If the T bit is set, then all P/V Synchronized view accesses, success or failure, cause a gated exception trap.

P/V Synchronized view accesses are not allowed to FIFO ITC locations.

### 12.3.6 P/V Try View

This view of the ITC location is similar in nature to the previous P/V Synchronized view in most respects other than the waiting policy on an access failure. It is to be used if the issuing thread can potentially find something else to do

and does not wish to be blocked if the access fails. A load with this view returns a value of zero even if the cell contains a data value of 0x0. Otherwise the load behaves as in the E/F Synchronized case. Normal stores using this view cause a saturating atomic increment of the cell value (saturating to 0xffff), as described for the P/V Synchronized view, and cannot fail. The operation of SC using this view is undefined.

If the T bit is set, then all PV Try view accesses, success or failure, will cause a gated exception trap.

P/V Try view accesses are not allowed to FIFO ITC locations.

### 12.3.7 Reserved Views

These views are reserved and should not be used by software.

## 12.4 ITC Address Space

The ITC physical address space is defined by two, 32 bit registers: *ITCAddressMap0* and *ITCAddressMap1*. Together these two registers specify a $2^N$ aligned block of uncached memory. The *BaseAddress* field of the *ITCAddressMap0* register specifies the starting address of the ITC memory block. The *AddrMask* of the *ITCAddressMap1* register determines the size of the memory block which can be varied from 1KB to 128KB. Within this address space, ITC cells are spread out with a stride specified by the *EntryGrain* field. Tightly spaced cells save on memory space, but widely spaced cells spread across a number of TLB pages, permitting different cells to be mapped to different processes. The number of cells is specified by the *NumEntries* field. See Table 12.5 and Table 12.6 for a detailed description of the *AddressMap* registers.

**Table 12.3 ITC AddressMap0 Register Format**

| 31 … 10 | 9 … 1 | 0 |
|---|---|---|
| BaseAddress | 0 | En |

**Table 12.4 ITCAddressMap1 Register Format**

| 31 | 30 … 20 | 19 … 17 | 16 … 10 | 9 … 3 | 2 … 0 |
|---|---|---|---|---|---|
| M | NumEntries | 0 | AddrMask | 0 | EntryGrain |

**Table 12.5 AddressMap0 Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit | | | |
| BaseAddress | 31:10 | The top [31:10] bits of the ITC Physical Memory Mapped Block | R/W | Undefined |
| En | 0 | ITC enable | R/W | 0 |
| 0 | 9:1 | Must be written as zeros; return zeros on read | 0 | 0 |

**Table 12.6 AddressMap1 Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bit | | | |
| M | 31 | This bit indicates if another ITC block is defined along with another pair of ITCAddressMap registers. On the 34K, this value is hardcoded to 0 | R | 0 |
| NumEntries | 30:20 | Number of ITC cells present | R | Preset |
| AddrMask | 16:10 | Indicates which bits of the BaseAddress field should not participate in determining an ITC memory hit. This field effectively defines the size of the ITC memory block. AddrMask set to zero implies a 1KB ITC address space, and AddrMask set to 0x3f implies a 128KB address space. | R/W | Undefined |
| EntryGrain | 2:0 | Cells are spaced at intervals of $128x2^{EntryGrain}$ bytes, or:<br>0x0 - 128B<br>0x1 - 256B<br>0x2 - 512B<br>0x3 - 1KB<br>0x4 - 2KB<br>0x5 - 4KB<br>0x6 - 8KB<br>0x7 - 16KB | R/W | Undefined |
| 0 | 19:17, 9:3 | Must be written as zeros; return zeros on read | 0 | 0 |

Depending on the setting of the *AddrMask*, *NumEntries*, and *EntryGrain*, it is possible that ITC cells do not fill up the entire ITC address block. If for example, two cells are mapped to a 1KB area with a stride of 256B (*EntryGrain* equal to 0x1), the first cell starts at offset 0x000 and the second at offset 0x100. The remaining two 256B regions starting at offsets 0x200 and 0x300 do not map to any storage. Any access to an address that does not map to an ITC entry will result in undefined behavior. It is also possible to set the ITC registers in a way that makes some of the cells unavailable.

*Chapter 13*

# Policy Manager in the 34K™ Core

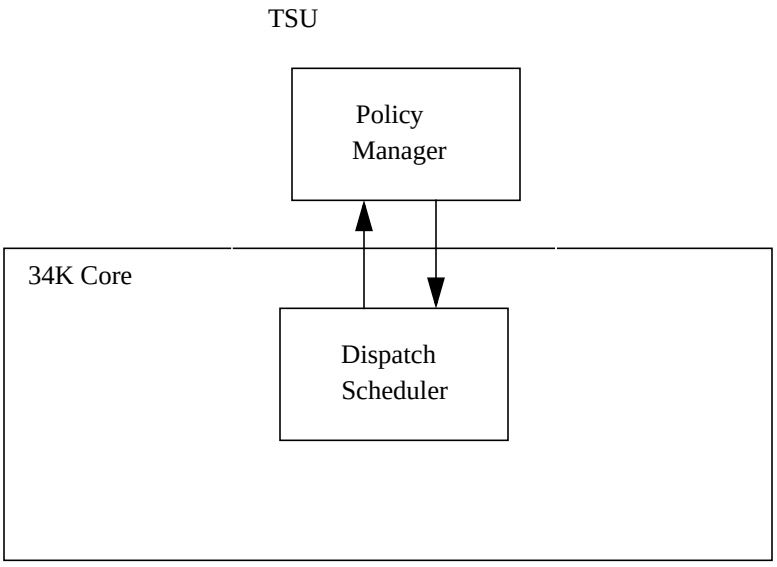
The 34K processor core includes a Policy Manager (PM) that is tasked with giving longer-term hints to the Dispatch Scheduler so as to achieve whatever performance allocation is desired in the system. The Policy Manager will be external to the core. MIPS will deliver choices of the policy managers. In addition, the customer may design their own Policy Manager.



## 13.1 Thread Scheduling Unit

The 34K core contains a unit called the Thread Scheduling Unit (TSU) which has 2 submodules - an internal Dispatch Scheduler and an external Policy Manager.

**Figure 13.1 TSU Block Diagram**




The Dispatch Scheduler (DS) will make cycle-by-cycle choices on which instructions to issue/dispatch. Since it is internal to the processor core, it will not be modifiable by the customer. The DS is designed to be as simple as possible and the system-specific complexity should be put into the policy manager so as not to burden the core with extra area/power which is not needed in all configurations.

The Policy Manager Interface incorporates a 4-level priority scheme. Each TC will be assigned to one of 4 groups and each group will have a unique priority level. Each cycle the dispatch scheduler will choose to dispatch an instruction from a TC in the highest priority group that contains any runnable TCs. If there are multiple TCs in the selected group, then it will choose among them using a round-robin algorithm. For more details about the Policy Manager Interface, please refer to Chapter 9, “Policy Manager Interface on the 34K™ Core” in the *MIPS32® 34K™ Processor Core Family Integrator’s Guide*.

## 13.2 Policy Managers

MIPS will provide a couple of reference PM designs with core deliverables. These designs will support a basic thread-scheduling programmability. The user can implement their own policy manager for more complex scheme.

### 13.2.1 Basic Round-Robin Policy Manager

All TCs are assigned to the same priority level. Since the internal Dispatch Scheduler implements a simple round-robin among TCs in the same priority level, all TCs are statically given the same weight and bandwidth, and will be fairly allocated amongst all runnable TCs.

This PM does not implement any thread-scheduling CP0 registers. Writes to these registers will be ignored. Reads from these registers will return -1.

When a new TC is forked, it will begin to participate in the round robin pool. This will thus cause the older TCs to get lower bandwidth allocations.

### 13.2.2 Weighted Round-Robin Policy Manager

The main difference between the basic round-robin policy and the weighted round-robin policy manager is the software controllability. With the WRR PM, TCs are scheduled round-robin style, but bandwidth given to individual TC can be adjusted or “weighted” by software so that a TC can get more or less than its fair share of the processor bandwidth.

This PM will implement the *TCSchedule* CP0 register, but only the *STP*, and *GRP* fields, and the *VPESchedule* register, but only the *GPO* field. The Group rotation schedule will of course be implemented. See details in Section 13.2.2.4 “Group Rotation Schedule”.

Also WRR PM will implement the full functionality of the *TCScheFBack* register. It will, however, not implement the *VPEScheFBack* register.

When a new TC is forked, the GRP of the new TC will be set to be the same as that of its parent.

#### 13.2.2.1 TCSchedule Register

The WRR policy manager will define the *TCSchedule* register for each TC as follows.

Figure 13.2 and Table 13.1 shows the format of the *TCSchedule* register.

**Figure 13.2 TCSchedule Register (CP0 Register2, Select 6)**

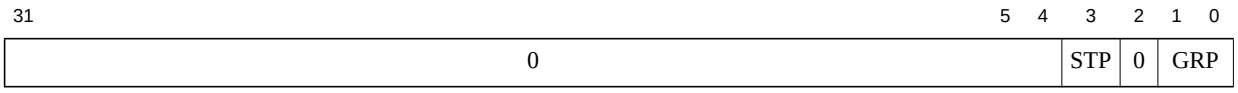


**Table 13.1 TCSchedule Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| 0 | 31:4,2 | Must be written as 0. Returns zero on reads. | 0 | 0 |
| STP | 3 | Stop Priority. Software sets this if this TC should never issue any instructions. | R/W | 0 |
| GRP | 1:0 | Group of the TC. Software sets this value to the group the TC should belong to. | R/W | 0 |

### 13.2.2.2 TCScheFBack Register

The WRR policy manager will define the *TCScheFBack* register for each TC as follows.

Figure 13.3 and Table 13.2 shows the format of the *TCScheFBack* register.

**Figure 13.3 TCScheFBack Register (CP0 Register2, Select 7)**

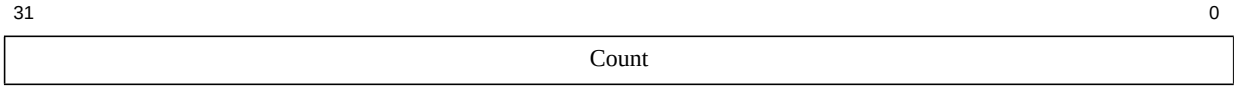



**Table 13.2 TCScheFBack Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| COUNT | 31:0 | This is a count of the number of instructions completed by this TC. The value will saturate at 32'hffff_ffff rather than rolling over to 0. | R/W | Undefined |

### 13.2.2.3 VPESchedule Register

The WRR policy manager will define the *VPESchedule* register for each VPE as follows.

Figure 13.4 and Table 13.3 shows the format of the *VPESchedule* register.

**Figure 13.4 VPESchedule Register (CP0 Register1, Select 5)**

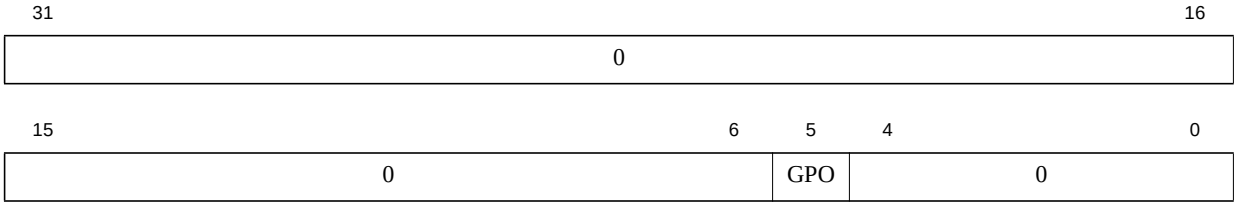


**Table 13.3 VPESchedule Register Field Descriptions**

| Fields | | Description | Read / Write | Reset State |
|---|---|---|---|---|
| Name | Bits | | | |
| 0 | 31:6,4:0 | Must be written as 0. Returns zero on reads. | 0 | 0 |
| GPO | 5 | Group priority override. When set, the priorities of the groups will be fixed as follows:<br>group3 will be priority3<br>group2 will be priority2<br>group1 will be priority1<br>group0 will be priority0<br>When cleared, the priorities of the groups will be rotated as described in Section 13.2.2.4 "Group Rotation Schedule"<br>NOTE: GPO is a per-processor field. There is only one GPO register, which is accessible from both GPO fields in a dual-VPE system. | R/W | 1 |

### 13.2.2.4 Group Rotation Schedule

When VPESchedule.GPO is cleared, the group priorities are rotated as described in this section. These rotations will enable a TC in a higher group to be prioritized higher than TCs in lower groups. The exact weighting function is complicated due to the interplay of non-runnable TCs, but generally, TCs in the next higher group will get at least twice the bandwidth of the TCs in the lower group.

The rotation schedule is described in Table 13.4.

**Table 13.4 Rotation of Group Priority Levels**

| Rotation Count | Group3 Priority | Group2 Priority | Group1 Priority | Group0 Priority |
|---|---|---|---|---|
| 4'b0001 | P3 | P2 | P1 | P0 |
| 4'b0010 | P2 | P3 | P0 | P1 |
| 4'b0011 | P3 | P2 | P1 | P0 |
| 4'b0100 | P2 | P0 | P3 | P1 |
| 4'b0101 | P3 | P2 | P1 | P0 |
| 4'b0110 | P2 | P3 | P0 | P1 |
| 4'b0111 | P3 | P1 | P2 | P0 |
| 4'b1000 | P0 | P2 | P1 | P3 |
| 4'b1001 | P3 | P1 | P2 | P0 |
| 4'b1010 | P2 | P3 | P0 | P1 |
| 4'b1011 | P3 | P2 | P1 | P0 |
| 4'b1100 | P2 | P0 | P3 | P1 |
| 4'b1101 | P3 | P2 | P1 | P0 |
| 4'b1110 | P2 | P3 | P0 | P1 |
| 4'b1111 | P3 | P1 | P2 | P0 |

The group priorities can easily be generated using a 4b counter and a priority encoder.

```
G3_priority = { Cnt[0] | Cnt[1] | Cnt[2], Cnt[0]};

G2_priority =
{(~Cnt[3]&~Cnt[2] | ~Cnt[2]&Cnt[1] | Cnt[1]&~Cnt[0] | ~Cnt[1]&(Cnt[2]^~Cnt[0])),
(Cnt[2]&Cnt[1] | Cnt[1]&~Cnt[0] | Cnt[3]&~Cnt[2]&~Cnt[1]&Cnt[0])};
```

The priority values for the other groups can easily be calculated from the above as follows:

```
G1_priority = ~G2_priority;
G0_priority = ~G3_priority;
```

With this mechanism, each group gets successively more slots at the highest priority. Group0 is the highest priority 1/15 slots, Group1 - 2/15, Group2 - 4/15, and Group3 - 8/15. Here are some of the properties of this rotation schedule.

For adjacent groups:

- Group3 is higher priority than group2 10/15 cycles. (G3 has 100% more bandwidth than G2)
- Group2 is higher priority than group1 10/15 cycles. (G2 has 100% more bandwidth than G1)
- Group1 is higher priority than group0 10/15 cycles. (G1 has 100% more bandwidth than G0)

For groups 2 levels apart:

- Group3 is higher priority than group1 12/15 cycles (G3 has 300% more bandwidth than G1)
- Group2 is higher priority than group0 12/15 cycles (G2 has 300% more bandwidth than G0)

And finally, for groups 3 levels apart:

- Group3 is higher priority than group0 14/15 cycles (G3 has 1300% more bandwidth than G0)

The priorities are rotated potentially every cycle. However, when the highest priority group in a given cycle has multiple runnable TCs in it, then that rotation is held for as many cycles as there are TCs in that highest priority group. This mechanism enables the relative bandwidth between groups to be maintained even when one group contains more TCs than another group.

For instance, assume we have a 4 TC system with 3 TCs in group1 and 1 TC in group0. The exact cycle by cycle priority is described in Table 13.5.

**Table 13.5 Priority Level Rotation (3TCs in group1, 1 TC in group0)**

| Cycle Count | Rotation Count | Group3 Priority | Group2 Priority | Group1 Priority | Group0 Priority |
|---|---|---|---|---|---|
| 1 | 4’b0001 | P3 | P2 | P1 | P0 |
| 2 | | P3 | P2 | P1 | P0 |
| 3 | | P3 | P2 | P1 | P0 |
| 4 | 4’b0010 | P2 | P3 | P0 | P1 |
| 5 | 4’b0011 | P3 | P2 | P1 | P0 |
| 6 | | P3 | P2 | P1 | P0 |
| 7 | | P3 | P2 | P1 | P0 |

**Table 13.5 Priority Level Rotation (3TCs in group1, 1 TC in group0)**

| Cycle Count | Rotation Count | Group3 Priority | Group2 Priority | Group1 Priority | Group0 Priority |
|---|---|---|---|---|---|
| 8 | 4’b0100 | P2 | P0 | P3 | P1 |
| 9 | | P2 | P0 | P3 | P1 |
| 10 | | P2 | P0 | P3 | P1 |
| 11 | 4’b0101 | P3 | P2 | P1 | P0 |
| 12 | | P3 | P2 | P1 | P0 |
| 13 | | P3 | P2 | P1 | P0 |
| 14 | 4’b0110 | P2 | P3 | P0 | P1 |
| 15 | 4’b0111 | P3 | P1 | P2 | P0 |
| 16 | | P3 | P1 | P2 | P0 |
| 17 | | P3 | P1 | P2 | P0 |
| 18 | 4’b1000 | P0 | P2 | P1 | P3 |
| 19 | 4’b1001 | P3 | P1 | P2 | P0 |
| 20 | | P3 | P1 | P2 | P0 |
| 21 | | P3 | P1 | P2 | P0 |
| 22 | 4’b1010 | P2 | P3 | P0 | P1 |
| 23 | 4’b1011 | P3 | P2 | P1 | P0 |
| 24 | | P3 | P2 | P1 | P0 |
| 25 | | P3 | P2 | P1 | P0 |
| 26 | 4’b1100 | P2 | P0 | P3 | P1 |
| 27 | | P2 | P0 | P3 | P1 |
| 28 | | P2 | P0 | P3 | P1 |
| 29 | 4’b1101 | P3 | P2 | P1 | P0 |
| 30 | | P3 | P2 | P1 | P0 |
| 31 | | P3 | P2 | P1 | P0 |
| 32 | 4’b1110 | P2 | P3 | P0 | P1 |
| 33 | 4’b1111 | P3 | P1 | P2 | P0 |
| 34 | | P3 | P1 | P2 | P0 |
| 35 | | P3 | P1 | P2 | P0 |

As can be seen in the table, the full rotation actually requires 35 cycles to complete. Out of these 35 cycles, group1 is higher priority than group0 for 30 cycles. However, since group1 contains 3 TCs, these will be round-robin’d by the DS, so on average, each of these TCs will get 33% of this group’s bandwidth, or 10cycles. (29% of all the issue slots for each of those TCs in group1). The one TC in group0 gets 5 issue slots, or 14%. As can be seen, each of the TCs in group1 gets about double the issue slots of the TC in group0.

*Chapter 14*

# Instruction Set Overview

This chapter provides a general overview on the three CPU instruction set formats of the MIPS architecture: Immediate, Jump, and Register. Refer to Chapter 15, "34K™ Processor Core Instructions" for a complete listing and description of instructions.

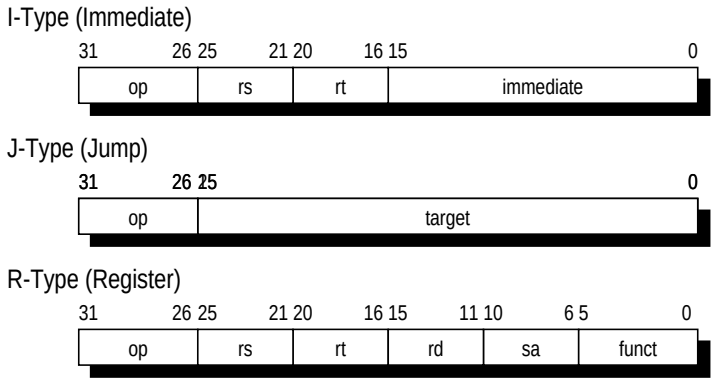
This chapter discusses the following topics



## 14.1 CPU Instruction Formats

Each CPU instruction consists of a single 32-bit word, aligned on a word boundary. There are three instruction formats immediate (I-type), jump (J-type), and register (R-type)—as shown in Figure 14.1. The use of a small number of instruction formats simplifies instruction decoding, allowing the compiler to synthesize more complicated (and less frequently used) operations and addressing modes from these three formats as needed.

**Figure 14.1 Instruction Formats**




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

## 14.2 Load and Store Instructions

Load and store instructions are immediate (I-type) instructions that move data between memory and the general registers. The only addressing mode that integer load and store instructions directly support is *base register plus 16-bit signed immediate offset*. Floating point load and store instructions can use either that addressing mode or *register plus register* indexed addressing.

### 14.2.1 Scheduling a Load Delay Slot

A load instruction that does not allow its result to be used by the instruction immediately following is called a *delayed load instruction*. The instruction slot immediately following this delayed load instruction is referred to as the *load delay slot*.

In a 34K core, the instruction immediately following a load instruction can use the contents of the loaded register; however in such cases hardware interlocks insert additional real cycles. Although not required, the scheduling of load delay slots can be desirable, both for performance and R-Series processor compatibility.

### 14.2.2 Defining Access Types

*Access type* indicates the size of a core data item to be loaded or stored, set by the load or store instruction opcode.

Regardless of access type or byte ordering (endianness), the address given specifies the low-order byte in the addressed field. For a big-endian configuration, the low-order byte is the most-significant byte; for a little-endian configuration, the low-order byte is the least-significant byte.

The access type, together with the three low-order bits of the address, define the bytes accessed within the addressed word as shown in Table 14.1. Only the combinations shown in Table 14.1 are permissible; other combinations cause address error exceptions.

Instruction fetches are either halfword accesses (MIPS16e™ code) or word accesses (32b code). These references will be impacted by endianness the same as load/store references of those sizes.

**Table 14.1 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 |
| 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 | | | | | | | |

## 14.3 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
- Count Leading Zeros/Ones
- Multiply
- Divide

These operations fit in the following four categories of computational instructions:

- ALU Immediate instructions
- Three-operand Register-type Instructions
- Shift Instructions
- Multiply And Divide Instructions

### 14.3.1 Cycle Timing for Multiply and Divide Instructions

Any multiply instruction in the integer pipeline is transferred to the multiplier as remaining instructions continue through the pipeline; the product of the multiply instruction is saved in the HI and LO registers. If the multiply instruction is followed by an MFHI or MFLO before the product is available, the pipeline interlocks until this product does become available. Refer to Chapter 2, "Pipeline of the 34K™ Core" for more information on instruction latency and repeat rates.

## 14.4 Jump and Branch Instructions

Jump and branch instructions change the control flow of a program. All jump and branch instructions occur with a delay of one instruction: that is, the instruction immediately following the jump or branch (this is known as the instruction in the *delay slot*) always executes while the target instruction is being fetched from storage.

### 14.4.1 Overview of Jump Instructions

Subroutine calls in high-level languages are usually implemented with Jump or Jump and Link instructions, both of which are J-type instructions. In J-type format, the 26-bit target address shifts left 2 bits and combines with the high-order 4 bits of the current program counter to form an absolute address.

Returns, dispatches, and large cross-page jumps are usually implemented with the Jump Register or Jump and Link Register instructions. Both are R-type instructions that take the 32-bit byte address contained in one of the general purpose registers.

For more information about jump instructions, refer to the individual instructions in *MIPS32® Architecture Reference Manual, Volume II: The MIPS32® Instruction Set*.

### 14.4.2 Overview of Branch Instructions

All branch instruction target addresses are computed by adding the address of the instruction in the delay slot to the 16-bit *offset* (shifted left 2 bits and sign-extended to 32 bits). All branches occur with a delay of one instruction.

If a conditional branch likely is not taken, the instruction in the delay slot is nullified.

Branches, jumps, ERET, and DERET instructions should not be placed in the delay slot of a branch or jump.

## 14.5 Control Instructions

Control instructions allow the software to initiate traps; they are always R-type.

## 14.6 Coprocessor Instructions

CP0 instructions perform operations on the System Control Coprocessor registers to manipulate the memory management and exception handling facilities of the processor. Refer to Chapter 15, “34K™ Processor Core Instructions” for a listing of CP0 instructions.

*Chapter 15*

# 34K™ Processor Core Instructions

This chapter supplements the MIPS32 Architecture Reference Manual by describing instruction behavior that is specific to a 34K processor core. The chapter is divided into the following sections:

- Section 15.1 "Understanding the Instruction Descriptions"
- Section 15.2 "34K™ Opcode Map"
- Section 15.3 "Floating Point Unit Instruction Format Encodings"
- Section 15.4 "MIPS32® Instruction Set for the 34K™ Core"

The 34K processor core also supports the MIPS16e ASE to the MIPS32 architecture. The MIPS16e ASE instruction set is described in Chapter 16, "MIPS16e™ Application-Specific Extension to the MIPS32® Instruction Set".

## 15.1 Understanding the Instruction Descriptions

Refer to Volume II of the MIPS32 Architecture Reference Manual for more information about the instruction descriptions. There is a description of the instruction fields, definition of terms, and a description function notation available in that document.

## 15.2 34K™ Opcode Map

**Table 15.1 Symbols Used in the Instruction Encoding Tables**

| Symbol | Meaning |
|---|---|
| * | Operation or field codes marked with this symbol are reserved for future use, are valid encodings for a higher-order MIPS ISA level, or are part of an application specific extension not implemented on this core. Executing such an instruction will cause 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 only legal if 64-bit floating point operations are enabled. In other cases, executing such an instruction will cause a Reserved Instruction Exception (non-coprocessor encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encodings for a coprocessor to which access is not allowed). |
| ϕ | Operation or field codes marked with this symbol are obsolete and will be removed from a future revision of the MIPS32 ISA. Software should avoid using these operation or field codes. |

### Table 15.2 MIPS32 Encoding of the Opcode Field

| **opcode** | | *bits 28..26* | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| *bits 31..29* | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 0 | 000 | *SPECIAL* 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 | *COP1*X | BEQL ϕ | BNEL ϕ | BLEZL ϕ | BGTZL ϕ |
| 3 | 011 | * | * | * | * | *SPECIAL2* δ | JALX | * | *SPECIAL3* δ |
| 4 | 100 | LB | LH | LWL | LW | LBU | LHU | LWR | * |
| 5 | 101 | SB | SH | SWL | SW | * | * | SWR | CACHE |
| 6 | 110 | LL | LWC1 | LWC2 | PREF | * | LDC1 | LDC2 | * |
| 7 | 111 | SC | SWC1 | SWC2 | * | * | SDC1 | SDC2 | * |

### Table 15.3 MIPS32 *SPECIAL* Opcode Encoding of Function Field

| **function** | | *bits 2..0* | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| *bits 5..3* | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 0 | 000 | SLL[1] | *MOVCI* δ | *SRL* δ | SRA | SLLV | * | *SRLV* δ | SRAV |
| 1 | 001 | JR[2] | JALR[2] | MOVZ | MOVN | SYSCALL | BREAK | * | SYNC |
| 2 | 010 | MFHI | MTHI | MFLO | MTLO | * | * | * | * |
| 3 | 011 | MULT | MULTU | DIV | DIVU | * | * | * | * |
| 4 | 100 | ADD | ADDU | SUB | SUBU | AND | OR | XOR | NOR |
| 5 | 101 | * | * | SLT | SLTU | * | * | * | * |
| 6 | 110 | TGE | TGEU | TLT | TLTU | TEQ | * | TNE | * |
| 7 | 111 | * | * | * | * | * | * | * | * |

1. Specific encodings of the *rt*, *rd*, and *sa* fields are used to distinguish among the SLL, NOP, SSNOP and EHB functions.
2. Specific encodings of the hint field are used to distinguish JR from JR.HB and JALR from JALR.HB

### Table 15.4 MIPS32 *REGIMM* Encoding of rt Field

| **rt** | | *bits 18..16* | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| *bits 20..19* | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 0 | 00 | BLTZ | BGEZ | BLTZL ϕ | BGEZL ϕ | * | * | * | * |
| 1 | 01 | TGEI | TGEIU | TLTI | TLTIU | TEQI | * | TNEI | * |
| 2 | 10 | BLTZAL | BGEZAL | BLTZALL ϕ | BGEZALL ϕ | * | * | * | * |
| 3 | 11 | * | * | * | * | * | * | * | SYNCI |

### Table 15.5 MIPS32 *SPECIAL2* Encoding of Function Field

| **function** | | *bits 2..0* | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| *bits 5..3* | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 0 | 000 | MADD | MADDU | MUL | * | MSUB | MSUBU | * | * |
| 1 | 001 | * | * | * | * | * | * | * | * |
| 2 | 010 | CorExtend | | | | | | | |
| 3 | 011 | | | | | | | | |
| 4 | 100 | CLZ | CLO | * | * | * | * | * | * |
| 5 | 101 | * | * | * | * | * | * | * | * |
| 6 | 110 | * | * | * | * | * | * | * | * |
| 7 | 111 | * | * | * | * | * | * | * | SDBBP |

### Table 15.6 MIPS32 *Special3* Encoding of Function Field for Release 2 of the Architecture

| **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 | EXT | * | * | * | INS | * | * | * |
| 1 | 001 | * | * | * | * | * | * | * | * |
| 2 | 010 | * | * | * | * | * | * | * | * |
| 3 | 011 | * | * | * | * | * | * | * | * |
| 4 | 100 | *BSHFL* δ | * | * | * | * | * | * | * |
| 5 | 101 | * | * | * | * | * | * | * | * |
| 6 | 110 | * | * | * | * | * | * | * | * |
| 7 | 111 | * | * | * | RDHWR | * | * | * | * |

### Table 15.7 MIPS32 *MOVCI* Encoding of tf Bit

| **tf** | *bit 16* | |
|---|---|---|
| | 0 | 1 |
| | MOVF | MOVT |

### Table 15.8 MIPS32 *SRL* Encoding of Shift/Rotate

| **tf** | *bit 21* | |
|---|---|---|
| | 0 | 1 |
| | SRL | ROTR |

### Table 15.9 MIPS32 *SRLV* Encoding of Shift/Rotate

| **tf** | *bit 6* | |
|---|---|---|
| | 0 | 1 |
| | SRLV | ROTRV |

### Table 15.10 MIPS32 *BSHFL* Encoding of sa Field[1]

| sa | | *bits 8..6* | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| *bits 10..9* | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 0 | 00 | | | WSBH | | | | | |
| 1 | 01 | | | | | | | | |
| 2 | 10 | SEB | | | | | | | |
| 3 | 11 | SEH | | | | | | | |

1. The sa field is sparsely decoded to identify the final instructions. Entries in this table with no mnemonic are reserved for future use by MIPS Technologies and may or may not cause a Reserved Instruction exception.

### Table 15.11 MIPS32 *COP0* Encoding of rs Field

| rs | | *bits 23..21* | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| *bits 25..24* | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 0 | 00 | MFC0 | * | * | * | MTC0 | * | * | * |
| 1 | 01 | * | * | RDPGPR | *MFMC0*[1] δ | * | * | WRPGPR | * |
| 2 | 10 | C0 δ | | | | | | | |
| 3 | 11 | | | | | | | | |

1. Release 2 of the Architecture added the MFMC0 function, which is further decoded as the DI and EI instructions.

### Table 15.12 MIPS32 *COP0* Encoding of Function Field When rs=*CO*

| function | | *bits 2..0* | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| *bits 5..3* | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 0 | 000 | * | TLBR | TLBWI | * | * | * | TLBWR | * |
| 1 | 001 | TLBP | * | * | * | * | * | * | * |
| 2 | 010 | * | * | * | * | * | * | * | * |
| 3 | 011 | ERET | * | * | * | * | * | * | DERET |
| 4 | 100 | WAIT | * | * | * | * | * | * | * |
| 5 | 101 | * | * | * | * | * | * | * | * |
| 6 | 110 | * | * | * | * | * | * | * | * |
| 7 | 111 | * | * | * | * | * | * | * | * |

### Table 15.13 MIPS32 *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 | * | CFC1 | MFHC1 | MTC1 | * | CTC1 | MTHC1 |
| 1 | 01 | *BC1* δ | * | * | * | * | * | * | * |
| 2 | 10 | *S* δ | *D* δ | * | * | *W* δ | *L* δ | * | * |
| 3 | 11 | * | * | * | * | * | * | * | * |

### Table 15.14 MIPS32 *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 | * | * | * | * | * | * | * | * |
| 7 | 111 | * | * | * | * | * | * | * | * |

### Table 15.15 MIPS32 *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 | * | * | * | * | * | * | * | * |
| 7 | 111 | * | * | * | * | * | * | * | * |

### Table 15.16 MIPS32 *COP1* Encoding of Function Field When rs=*W or L*[1]

| **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 | * | * | * | * | * | * | * | * |

1. Format type L is legal only if 64-bit floating point operations are enabled.

**Table 15.17 MIPS32 *COP1* Encoding of tf Bit When rs=*S or D*, Function=*MOVCF***

| **tf** | *bit 16* | |
|---|---|---|
| | 0 | 1 |
| | MOVF.fmt | MOVT.fmt |

**Table 15.18 MIPS64 *COP1X* Encoding of Function Field[1]**

| **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 ∇ | * | * | 24k * | * | ⁎ | * |
| 7 | 111 | NMSUB.S ∇ | NMSUB.D ∇ | * | * | * | * | ⁎ | * |

1. COP1X instructions are legal only if 64-bit floating point operations are enabled.

**Table 15.19 MIPS32 *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 | ⁎ | CFC2 | MFHC2 | MTC2 | ⁎ | CTC2 | MTHC2 |
| 1 | 01 | *BC2*δ | ⁎ | ⁎ | * | * | * | * | * |
| 2 | 10 | C2 | | | | | | | |
| 3 | 11 | | | | | | | | |

# 15.3 Floating Point Unit Instruction Format Encodings

Instruction format encodings for the floating point unit are presented in this section. This information is a tabular presentation of the encodings described in tables Table 15.13 and Table 15.18 above.

**Table 15.20 Floating Point Unit Instruction Format Encodings**

| ***fmt* field (bits 25..21 of COP1 opcode)** | | ***fmt3* field (bits 2..0 of COP1X opcode)** | | | | | |
|---|---|---|---|---|---|---|---|
| **Decimal** | **Hex** | **Decimal** | **Hex** | **Mnemonic** | **Name** | **Bit Width** | **Data Type** |
| 0..15 | 00..0F | — | — | Used to encode Coprocessor 1 interface instructions (MFC1, CTC1, etc.). Not used for format encoding. | | | |
| 16 | 10 | 0 | 0 | S | Single | 32 | Floating Point |
| 17 | 11 | 1 | 1 | D | Double | 64 | Floating Point |

**Table 15.20 Floating Point Unit Instruction Format Encodings**

| *fmt* field (bits 25..21 of COP1 opcode) | | *fmt3* field (bits 2..0 of COP1X opcode) | | Mnemonic | Name | Bit Width | Data Type |
|---|---|---|---|---|---|---|---|
| Decimal | Hex | Decimal | Hex | | | | |
| 18..19 | 12..13 | 2..3 | 2..3 | Reserved for future use by the architecture. | | | |
| 20 | 14 | 4 | 4 | W | Word | 32 | Fixed Point |
| 21 | 15 | 5 | 5 | L | Long | 64 | Fixed Point |
| 22 | 16 | 6 | 6 | PS | Paired Single | $2 \times 32$ | Floating Point |
| 23 | 17 | 7 | 7 | Reserved for future use by the architecture. | | | |
| 24..31 | 18..1F | — | — | Reserved for future use by the architecture. Not available for *fmt3* encoding. | | | |

# 15.4 MIPS32® Instruction Set for the 34K™ Core

This section describes the MIPS32 instructions for the 34K cores. Table 15.21 lists the instructions in alphabetical order. Instructions that have implementation dependent behavior are described afterwards. The descriptions for other instructions exist in the architecture reference manual and are not duplicated here.

**Table 15.21 34K™ Core Instruction Set**

| Instruction | Description | Function |
|---|---|---|
| `ABS.fmt` | Floating Point Absolute Value<br>fmt = s,d | `Fd = abs(Fs)` |
| `ADD` | Integer Add | `Rd = Rs + Rt` |
| `ADD.fmt` | Floating Point Add<br>fmt = s,d | `Fd = Fs + Ft` |
| `ADDI` | Integer Add Immediate | `Rt = Rs + Immed` |
| `ADDIU` | Unsigned Integer Add Immediate | `Rt = Rs +`$_U$ `Immed` |
| `ADDIUPC` | Unsigned Integer Add Immediate to PC (MIPS16 only) | `Rt = PC +`$_u$ `Immed` |
| `ADDU` | Unsigned Integer Add | `Rd = Rs +`$_U$ `Rt` |
| `AND` | Logical AND | `Rd = Rs & Rt` |
| `ANDI` | Logical AND Immediate | `Rt = Rs & (`$0_{16}$ `\|\| Immed)` |
| `B` | Unconditional Branch<br>(Assembler idiom for: `BEQ r0, r0, offset`) | `PC += (int)offset` |
| `BAL` | Branch and Link<br>(Assembler idiom for: `BGEZAL r0, offset`) | `GPR[31] = PC + 8`<br>`PC += (int)offset` |
| `BC1F` | Branch On Floating Point False | `if (cc[i] == 0) then`<br>`  PC += (int)offset` |
| `BC1FL` | Branch On Floating Point False Likely | `if (cc[i] == 0)then`<br>`  PC += (int)offset`<br>`else`<br>`  Ignore Next Instruction` |

**Table 15.21 34K™ Core Instruction Set (Continued)**

| Instruction | Description | Function |
|---|---|---|
| BC1T | Branch On Floating Point True | `if(cc[i] == 1) then`<br>`  PC += (int)offset` |
| BC1TL | Branch On Floating Point True Likely | `if (cc[i] == 1) then`<br>`  PC += (int)offset`<br>`else`<br>`  Ignore Next Instruction` |
| BC2F | Branch On CP2 False | `if (cc[i] == 0) then`<br>`  PC += (int)offset` |
| BC2FL | Branch On CP2 False Likely | `if (cc[i] == 0)then`<br>`  PC += (int)offset`<br>`else`<br>`  Ignore Next Instruction` |
| BC2T | Branch On CP2 True | `if(cc[i] == 1) then`<br>`  PC += (int)offset` |
| BC2TL | Branch On CP2 True Likely | `if (cc[i] == 1) then`<br>`  PC += (int)offset`<br>`else`<br>`  Ignore Next Instruction` |
| BEQ | Branch On Equal | `if Rs == Rt`<br>`  PC += (int)offset` |
| BEQL | Branch On Equal Likely | `if Rs == Rt`<br>`  PC += (int)offset`<br>`else`<br>`  Ignore Next Instruction` |
| BGEZ | Branch on Greater Than or Equal To Zero | `if !Rs[31]`<br>`  PC += (int)offset` |
| BGEZAL | Branch on Greater Than or Equal To Zero And Link | `GPR[31] = PC + 8`<br>`if !Rs[31]`<br>`  PC += (int)offset` |
| BGEZALL | Branch on Greater Than or Equal To Zero And Link Likely | `GPR[31] = PC + 8`<br>`if !Rs[31]`<br>`  PC += (int)offset`<br>`else`<br>`  Ignore Next Instruction` |
| BGEZL | Branch on Greater Than or Equal To Zero Likely | `if !Rs[31]`<br>`  PC += (int)offset`<br>`else`<br>`  Ignore Next Instruction` |
| BGTZ | Branch on Greater Than Zero | `if !Rs[31] && Rs != 0`<br>`  PC += (int)offset` |
| BGTZL | Branch on Greater Than Zero Likely | `if !Rs[31] && Rs != 0`<br>`  PC += (int)offset`<br>`else`<br>`  Ignore Next Instruction` |
| BLEZ | Branch on Less Than or Equal to Zero | `if Rs[31] \|\| Rs == 0`<br>`  PC += (int)offset` |

**Table 15.21 34K™ Core Instruction Set (Continued)**

| Instruction | Description | Function |
|---|---|---|
| BLEZL | Branch on Less Than or Equal to Zero Likely | `if Rs[31] \|\| Rs == 0`<br>`  PC += (int)offset`<br>`else`<br>`  Ignore Next Instruction` |
| BLTZ | Branch on Less Than Zero | `if Rs[31]`<br>`  PC += (int)offset` |
| BLTZAL | Branch on Less Than Zero And Link | `GPR[31] = PC + 8`<br>`if Rs[31]`<br>`  PC += (int)offset` |
| BLTZALL | Branch on Less Than Zero And Link Likely | `GPR[31] = PC + 8`<br>`if Rs[31]`<br>`  PC += (int)offset`<br>`else`<br>`  Ignore Next Instruction` |
| BLTZL | Branch on Less Than Zero Likely | `if Rs[31]`<br>`  PC += (int)offset`<br>`else`<br>`  Ignore Next Instruction` |
| BNE | Branch on Not Equal | `if Rs != Rt`<br>`  PC += (int)offset` |
| BNEL | Branch on Not Equal Likely | `if Rs != Rt`<br>`  PC += (int)offset`<br>`else`<br>`  Ignore Next Instruction` |
| BREAK | Breakpoint | Break Exception |
| C.cond.fmt | Floating Point Compare<br>fmt = s,d | cc[i] = Fs compare_cond Ft |
| CACHE | Cache Operation | See Software User's Manual |
| CEIL.L.fmt | Floating Point Ceiling to Long Fixed Point | `Fd = convert_and_round(Fs)` |
| CEIL.W.fmt | Floating Point Ceiling to Word Fixed Point | `Fd = convert_and_round(Fs)` |
| CFC1 | Move Control Word From Floating Point | `Rt = FP_Control[Fs]` |
| CFC2 | Move Control Word From CP2 | `Rt = CP2_Control[Fs]` |
| CLO | Count Leading Ones | `Rd = NumLeadingOnes(Rs)` |
| CLZ | Count Leading Zeroes | `Rd = NumLeadingZeroes(Rs)` |
| COP0 | Coprocessor 0 Operation | See Software User's Manual |
| COP2 | Coprocessor 2 Operation | Implementation dependent |
| CTC1 | Move Control Word To Floating Point | `FP_Control[Fs] = Rt` |
| CTC2 | Move Control Word to CP2 | `CP2 Control[Fs] = Rt` |
| CVT.D.fmt | Floating Point Convert to Double Floating Point<br>fmt = S,W,L | `Fd = convert_and_round(Fs)` |

**Table 15.21 34K™ Core Instruction Set (Continued)**

| Instruction | Description | Function |
|---|---|---|
| CVT.D.fmt | Floating Point Convert to Double Floating Point<br>fmt = S,W,L | Fd = convert_and_round(Fs) |
| CVT.L.fmt | Floating Point Convert to Long Fixed Point<br>fmt = S,D | Fd = convert_and_round(Fs) |
| CVT.S.fmt | Floating Point Convert to Single Floating Point<br>fmt = W,D,L | Fd = convert_and_round(Fs) |
| CVT.W.fmt | Floating Point Convert to Word Fixed Point<br>fmt = S,D | Fd = convert_and_round(Fs) |
| DERET | Return from Debug Exception | PC = DEPC<br>Exit Debug Mode |
| DI | Atomically Disable Interrupts | Rt = Status; $Status_{IE}$ = 0 |
| DIV | Divide | LO = (int)Rs / (int)Rt<br>HI = (int)Rs % (int)Rt |
| DIV.fmt | Floating Point Divide<br>fmt = S,D | Fd = Fs/Ft |
| DIVU | Unsigned Divide | LO = (uns)Rs / (uns)Rt<br>HI = (uns)Rs % (uns)Rt |
| DMT | Clear *VPEControl[TE]*, which suspends execution of all other TCs affiliates to the same VPE. | The *rt* register receives the original value of *VPEControl*; if you don't specify a register rt it receives the previous contents of the *MVPControl* register. |
| DVPE | Disable all multithreading, including any other TCs affiliated to other VPEs, leaving this thread running alone. | Implemented as an atomic clear of the *MVPContorl[VEP]* bit. If you specify a register rt it receives the previous contents of the *MVPControl* register. |
| EHB | Execution Hazard Barrier | Stop instruction execution until execution hazards are cleared |
| EI | Atomically Enable Interrupts | Rt = Status; $Status_{IE}$ = 1 |
| EMT | Atomically sets the *VPEControl[TE]* bit and returns the old value. | VPEControl[TE] = 1 |
| ERET | Return from Exception | if SR[2]<br>  PC = ErrorEPC<br>else<br>  PC = EPC<br>  SR[1] = 0<br>SR[2] = 0<br>LL = 0 |
| EVPE | Returns the previous value of the *MVPControl* register and enable multi-VPE execution. | GPR[rt] <- MVPControl;<br>$MVPControl_{EVP}$ <- 1 |
| EXT | Extract Bit Field | Rt = ExtractField(Rs, pos, size) |
| FLOOR.L.fmt | Floating Point Floor to Long Fixed Point<br>fmt = S,D | Fd = convert_and_round(Fs) |

**Table 15.21 34K™ Core Instruction Set (Continued)**

| Instruction | Description | Function |
|---|---|---|
| FLOOR.W.fmt | Floating Point Floor to Word Fixed Point<br>fmt = S,D | Fd = convert_and_round(Fs) |
| FORK | Find a TC and activate it, so it starts at rs. | The new thread's rd register will be set to the value provided in rt. |
| INS | Insert Bit Field | Rt = InsertField(Rs, Rt, pos, size) |
| J | Unconditional Jump | PC = PC[31:28] \|\| offset<<2 |
| JAL | Jump and Link | GPR[31] = PC + 8<br>PC = PC[31:28] \|\| offset<<2 |
| JALR | Jump and Link Register | Rd = PC + 8<br>PC = Rs |
| JALR.HB | Jump and Link Register with Hazard Barrier | Like JALR, but also clears execution and instruction hazards |
| JALRC | Jump and Link Register Compact - do not execute instruction in jump delay slot(MIPS16 only) | Rd = PC + 2<br>PC = Rs |
| JR | Jump Register | PC = Rs |
| JR.HB | Jump Register with Hazard Barrier | Like JR, but also clears execution and instruction hazards |
| JRC | Jump Register Compact - do not execute instruction in jump delay slot (MIPS16 only) | PC = Rs |
| LB | Load Byte | Rt = (byte)Mem[base+offset] |
| LBU | Unsigned Load Byte | Rt = (ubyte)Mem[base+offset] |
| LDC1 | Load Doubleword to Floating Point | Ft = memory[base+offset] |
| LDC2 | Load Doubleword to CP2 | Ft = memory[base+offset] |
| LDXC1 | Load Doubleword Indexed to Floating Point | Fd = memory[base+index] |
| LH | Load Halfword | Rt = (half)Mem[base+offset] |
| LHU | Unsigned Load Halfword | Rt = (uhalf)Mem[base+offset] |
| LL | Load Linked Word | Rt = Mem[base+offset]<br>LL = 1 |
| LUI | Load Upper Immediate | Rt = immediate << 16 |
| LUXC1 | Load Doubleword Indexed Unaligned to Floating Point | Fd = memory[(base+index)psize-1..3 |
| LW | Load Word | Rt = Mem[Rs+offset] |
| LWC1 | Load Word to Floating Point | Ft = memory[base+offset] |
| LWC2 | Load Word to CP2 | Ft = memory[base+offset] |
| LWPC | Load Word, PC relative | Rt = Mem[PC+offset] |

**Table 15.21 34K™ Core Instruction Set (Continued)**

| Instruction | Description | Function |
|---|---|---|
| LWXC1 | Load Word Indexed to Floating Point | Fd = memory[base+index] |
| LWL | Load Word Left | See Architecture Reference Manual |
| LWR | Load Word Right | See Architecture Reference Manual |
| MADD | Multiply-Add | HI \| LO += (int)Rs * (int)Rt |
| MADD.fmt | Floating Point Multiply Add<br>fmt = S,D | Fd = Fs * Ft + Fr |
| MADDU | Multiply-Add Unsigned | HI \| LO += (uns)Rs * (uns)Rt |
| MFC0 | Move From Coprocessor 0 | Rt = CPR[0, Rd, sel] |
| MFC1 | Move From FPR | Rt = $Fs_{31..0}$ |
| MFC2 | Move From CP2 Register | Rt = $Fs_{31..0}$ |
| MFHC1 | Move From High Half of FPR | Rt = $Fs_{63..32}$ |
| MFHC2 | Move From High Half of CP2 Register | Rt = $Fs_{63..32}$ |
| MFHI | Move From HI | Rd = HI |
| MFLO | Move From LO | Rd = LO |
| MFTR | Move from thread register belonging to some other TC. | Rd = Rt |
| MOV.fmt | Floating Point Move | Fd = Fs |
| MOVF | GPR Conditional Move on Floating Point False | if (cc[i] == 0) then Rd = Rs |
| MOVF.fmt | FPR Conditional Move on Floating Point False | if (cc[i] == 0) then Fd = Fs |
| MOVN | GPR Conditional Move on Not Zero | if Rt ≠ 0 then<br>Rd = Rs |
| MOVN.fmt | FPR Conditional Move on Not Zero | if Rt ≠ 0 then<br>Fd = Fs |
| MOVT | GPR Conditional Move on Floating Point True | if (cc[i] == 1) then Rd = Rs |
| MOVT.fmt | FPR Conditional Move on Floating Point True | if (cc[i] == 1) then Fd = Fs |
| MOVZ | GPR Conditional Move on Zero | if Rt = 0 then<br>Rd = Rs |
| MOVZ.fmt | FPR Conditional Move on Zero | if (Rt == 0) then Fd = Fs |
| MSUB | Multiply-Subtract | HI \| LO -= (int)Rs * (int)Rt |
| MSUB.fmt | Floating Point Multiply Subtract<br>fmt = S,D | Fd = Fs * Ft - Fr |
| MSUBU | Multiply-Subtract Unsigned | HI \| LO -= (uns)Rs * (uns)Rt |
| MTC0 | Move To Coprocessor 0 | CPR[0, n, Sel] = Rt |
| MTC1 | Move To FPR | Fs = Rt |
| MTC2 | Move to CP2 register | Fs = Rt |

**Table 15.21 34K™ Core Instruction Set (Continued)**

| Instruction | Description | Function |
|---|---|---|
| MTHC1 | Move To High Half of FPR | Fd = Rt \|\| $Fs_{31..0}$ |
| MTHC2 | Move to High Half of CP2 register | Fd = Rt \|\| $Fs_{31..0}$ |
| MTHI | Move To HI | HI = Rs |
| MTLO | Move To LO | LO = Rs |
| MTTR | Move to thread register belonging to some other TC. | Rt = Rd |
| MUL | Multiply with register write | HI \| LO =Unpredictable<br>Rd = ((int)Rs * (int)$Rt)_{31..0}$ |
| MUL.fmt | Floating Point Multiply<br>fmt = S,D | Fd = Fs * Ft |
| MULT | Integer Multiply | HI \| LO = (int)Rs * (int)Rd |
| MULTU | Unsigned Multiply | HI \| LO = (uns)Rs * (uns)Rd |
| NEG.fmt | Floating Point Negate<br>fmt = S,D | Fd = neg(Fs) |
| NMADD.fmt | Floating Point Negative Multiply Add<br>fmt = S,D | Fd = neg(Fs * Ft + Fr) |
| NMSUB.fmt | Floating Point Negative Multiply Subtract<br>fmt = S,D | Fd = neg(Fs * Ft - Fr) |
| NOP | No Operation<br>(Assembler idiom for: SLL r0, r0, r0) | |
| NOR | Logical NOR | Rd = ~(Rs \| Rt) |
| OR | Logical OR | Rd = Rs \| Rt |
| ORI | Logical OR Immediate | Rt = Rs \| Immed |
| PREF | Prefetch | Load Specified Line into Cache |
| PREFX | Prefetch Indexed | Load Specified Line into Cache |
| RDHWR | Read Hardware Register | Allows unprivileged access to registers enabled by *HWREna* register |
| RDPGPR | Read GPR from Previous Shadow Set | Rt = $SGPR[SRSCtl_{PSS}, Rd]$ |
| RECIP.fmt | Floating Point Reciprocal Approximation<br>fmt = S,D | Fd = recip(Fs) |
| RESTORE | Restore registers and deallocate stack frame (MIPS16 only) | See Architecture Reference Manual |
| ROTR | Rotate Word Right | Rd = $Rt_{sa-1..0}$ \|\| $Rt_{31..sa}$ |
| ROTRV | Rotate Word Right Variable | Rd = $Rt_{Rs-1..0}$ \|\| $Rt_{31..Rs}$ |
| ROUND.L.fmt | Floating Point Round to Long Fixed Point<br>fmt = S,D | Fd = convert_and_round(Fs) |

**Table 15.21 34K™ Core Instruction Set (Continued)**

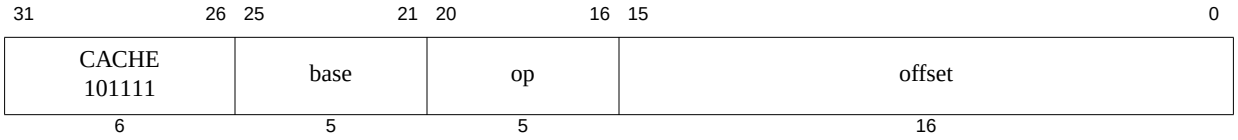
| Instruction | Description | Function |
|---|---|---|
| ROUND.W.fmt | Floating Point Round to Word Fixed Point<br>fmt = S,D | Fd = convert_and_round(Fs) |
| RSQRT.fmt | Floating Point Reciprocal Square Root Approximation<br>fmt = S,D | Fd = rsqrt(Fs) |
| SAVE | Save registers and allocate stack frame (MIPS16 only) | See Architecture Reference Manual |
| SB | Store Byte | (byte)Mem[base+offset] = Rt |
| SC | Store Conditional Word | if LL = 1<br>  mem[base+offset] = Rt<br>Rt = LL |
| SDBBP | Software Debug Break Point | Trap to SW Debug Handler |
| SDC1 | Store Doubleword from Floating Point | memory[base+offset] = Ft |
| SDC2 | Store Doubleword from CP2 | memory[base+offset] = Ft |
| SDXC1 | Store Word Indexed from Floating Point | memory[base+index] = Fs |
| SEB | Sign Extend Byte | Rd = (byte)Rs |
| SEH | Sign Extend Half | Rd = (half)Rs |
| SH | Store Half | (half)Mem[base+offset] = Rt |
| SLL | Shift Left Logical | Rd = Rt << sa |
| SLLV | Shift Left Logical Variable | Rd = Rt << Rs[4:0] |
| SLT | Set on Less Than | if (int)Rs < (int)Rt<br>  Rd = 1<br>else<br>  Rd = 0 |
| SLTI | Set on Less Than Immediate | if (int)Rs < (int)Immed<br>  Rt = 1<br>else<br>  Rt = 0 |
| SLTIU | Set on Less Than Immediate Unsigned | if (uns)Rs < (uns)Immed<br>  Rt = 1<br>else<br>  Rt = 0 |
| SLTU | Set on Less Than Unsigned | if (uns)Rs < (uns)Immed<br>  Rd = 1<br>else<br>  Rd = 0 |
| SQRT.fmt | Floating Point Square Root<br>fmt = S,D | Fd = sqrt(Fs) |
| SRA | Shift Right Arithmetic | Rd = (int)Rt >> sa |
| SRAV | Shift Right Arithmetic Variable | Rd = (int)Rt >> Rs[4:0] |
| SRL | Shift Right Logical | Rd = (uns)Rt >> sa |

**Table 15.21 34K™ Core Instruction Set (Continued)**

| Instruction | Description | Function |
|---|---|---|
| SRLV | Shift Right Logical Variable | Rd = (uns)Rt >> Rs[4:0] |
| SSNOP | Superscalar Inhibit No Operation | NOP |
| SUB | Integer Subtract | Rt = (int)Rs - (int)Rd |
| SUB.fmt | Floating Point Subtract<br>fmt = S,D | Fd = Fs - Ft |
| SUBU | Unsigned Subtract | Rt = (uns)Rs - (uns)Rd |
| SUXC1 | Store Doubleword Indexed Unaligned from Floating Point | memory[(base+index)psize-1..3] = Fs |
| SW | Store Word | Mem[base+offset] = Rt |
| SWC1 | Store Word From Floating Point | Mem[base+offset] = Fs |
| SWC2 | Store Word From CP2 Register | Mem[base+offset] = Fs |
| SWL | Store Word Left | See Architecture Reference Manual |
| SWR | Store Word Right | See Architecture Reference Manual |
| SWXC1 | Store Word Indexed to Floating Point | memory[base+index] = Fs |
| SYNC | Synchronize | See Software User's Manual |
| SYNCI | Synchronize Caches to Make Instruction Writes Effective | For D-cache writeback and I-cache invalidate on specified address |
| SYSCALL | System Call | SystemCallException |
| TEQ | Trap if Equal | if Rs == Rt<br>TrapException |
| TEQI | Trap if Equal Immediate | if Rs == (int)Immed<br>TrapException |
| TGE | Trap if Greater Than or Equal | if (int)Rs >= (int)Rt<br>TrapException |
| TGEI | Trap if Greater Than or Equal Immediate | if (int)Rs >= (int)Immed<br>TrapException |
| TGEIU | Trap if Greater Than or Equal Immediate Unsigned | if (uns)Rs >= (uns)Immed<br>TrapException |
| TGEU | Trap if Greater Than or Equal Unsigned | if (uns)Rs >= (uns)Rt<br>TrapException |
| TLBWI | Write Indexed TLB Entry | See Software Users Manual |
| TLBWR | Write Random TLB Entry | See Software Users Manual |
| TLBP | Probe TLB for Matching Entry | See Software Users Manual |
| TLBR | Read Index for TLB Entry | See Software Users Manual |
| TLT | Trap if Less Than | if (int)Rs < (int)Rt<br>TrapException |

**Table 15.21 34K™ Core Instruction Set (Continued)**

| Instruction | Description | Function |
|---|---|---|
| TLTI | Trap if Less Than Immediate | `if (int)Rs < (int)Immed TrapException` |
| TLTIU | Trap if Less Than Immediate Unsigned | `if (uns)Rs < (uns)Immed TrapException` |
| TLTU | Trap if Less Than Unsigned | `if (uns)Rs < (uns)Rt TrapException` |
| TNE | Trap if Not Equal | `if Rs != Rt TrapException` |
| TNEI | Trap if Not Equal Immediate | `if Rs != (int)Immed TrapException` |
| TRUNC.L.fmt | Floating Point Truncate to Long Fixed Point | `Fd = convert_and_round(Fs)` |
| TRUNC.W.fmt | Floating Point Truncate to Word Fixed Point | `Fd = convert_and_round(Fs)` |
| WAIT | Wait for Interrupts | Stall until interrupt occurs |
| WRPGPR | Write to GPR in Previous Shadow Set | $SGPR[SRSCtl_{PSS}, Rd] = Rt$ |
| WSBH | Word Swap Bytes Within HalfWords | $Rd = Rt_{23..16} \mid\mid Rt_{31..24} \mid\mid Rt_{7..0} \mid\mid Rt_{15..8}$ |
| XOR | Exclusive OR | `Rd = Rs ^ Rt` |
| XORI | Exclusive OR Immediate | `Rt = Rs ^ (uns)Immed` |
| YIELD | A multipurpose instruction whose action depends on Rs. | `Rs==0;` It terminates the thread and makesthe TC available for a subsequent fork. `Rs==-1;` paused while other threads run and any scheduling policy change filters through. `Rs==-2;` is just done to poll yield inputs. Rs > 0; you wait for one of the yield input signals, but only one for which there's a corresponding bit set in Rs. |
| ZEB | Zero extend byte (MIPS16 only) | `Rt = (ubyte) Rs` |
| ZEH | Zero extend half (MIPS16 only) | `Rt = (uhalf) Rs` |




**Format:** `CACHE op, offset(base)` **MIPS32**

**Purpose:** Perform Cache Operation

To perform the cache operation specified by op.

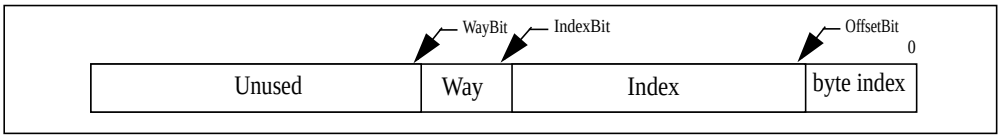
**Description:**

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 15.22 Usage of Effective Address**

| Operation Requires an | Type of Cache | Usage of Effective Address |
|---|---|---|
| Address | Physical | The effective address is translated by the MMU to a physical address. The physical address is then used to address the cache |
| Index | N/A | The effective address is used to index the cache.<br><br>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:<br><br>`OffsetBit ← Log2(BPT)`<br>`IndexBit ← Log2(CS / A)`<br>`WayBit ← IndexBit + Ceiling(Log2(A))`<br>$Way \leftarrow Addr_{WayBit-1..IndexBit}$<br>$Index \leftarrow Addr_{IndexBit-1..OffsetBit}$ |

**Figure 15.1 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.

The effective address may be an arbitrarily-aligned by address. The CACHE instruction never causes an Address Error Exception due to an non-aligned address.

A Cache Error exception may occur as a by-product 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. However, cache error exceptions should not be triggered by an Index Load Tag or Index Store

tag operation, as these operations are used for initialization and diagnostic purposes.

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.

Bits [17:16] of the instruction specify the cache on which to perform the operation, as follows:

**Table 15.23 Encoding of Bits[17:16] of CACHE Instruction**

| Code | Name | Cache | Cop0 Registers Used |
|---|---|---|---|
| 2#00 | I | Primary Instruction | TagLo0, DataLo0, DataHi0, ErrCtl |
| 2#01 | D | Primary Data | TagLo1, DataLo1, ErrCtl |
| 2#10 | T | Tertiary - Not supported | |
| 2#11 | S | Secondary | TagLo2, DataLo2, DataHi2 |

Some of the operations use coprocessor0 registers as either sources or destinations. Each of the caches has a separate set of Tag and Data registers. The last column in Table 15.25 lists which registers are used by operations to each cache. In the description of the operations, these may be explicitly listed or referred to in general, such as *TagLoN*, which would refer to the TagLo register corresponding to that cache.

Bits [20:18] of the instruction specify the operation to perform. On Index Load Tag and Index Store Data operations, the specific word (primary D) or double-word (primary I, secondary) that is addressed is loaded into / read from the *DataLo1* (primary D), *DataLo2* and *DataHi2* (secondary), or *DataLo0* and *DataHi0* (primary I) registers. All other cache instructions are line-based and the word and byte indexes will not affect their operation.

**Table 15.24 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST,SPR] Cleared**

| Code | Caches | Name | Effective Address Operand Type | Operation | Implemented? |
|---|---|---|---|---|---|
| 2#000 | I | Index Invalidate | Index | Set the state of the cache line at the specified index to invalid.<br>This encoding may be used by software to invalidate the entire instruction cache by stepping through all valid indices. | Yes |
| | D, S, T | Index Writeback Invalidate | Index | If the state of the cache line at the specified index is valid and dirty, write the line back to the memory address specified by the cache tag. After that operation is completed, set the state of the cache line to invalid. If the line is valid but not dirty, set the state of the line to invalid.<br><br>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 |

**Table 15.24 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST,SPR] Cleared (Continued)**

| Code | Caches | Name | Effective Address Operand Type | Operation | Implemented? |
|---|---|---|---|---|---|
| 2#001 | I | Index Load Tag | Index | • Read the tag for the cache line at the specified index into the *TagLo0* Coprocessor 0 register.<br>• Read the data corresponding to the dword index into the *DataLo0* and *DataHi0* registers.<br>• Precode bits and data array parity bits are also read into the *ErrCtl* register. | Yes |
| 2#001 | D | Index Load Tag | Index | • Read the tag for the cache line at the specified index into the *TagLo0* Coprocessor 0 register.<br>• Read the data corresponding to the word index into the *DataLo1* register.<br>• Data array parity bits are also read into the *ErrCtl* register. | Yes |
| 2#001 | S | Index Load Tag | Index | • Read the tag for the cache line at the specified index into the *TagLo2* Coprocessor 0 register.<br>• Read the data corresponding to the dword index into the *DataLo2* and *DataHi2* registers. | Yes |
| 2#010 | All | Index Store Tag | Index | Write the tag for the cache line at the specified index from the associated *TagLoN* Coprocessor 0 register.<br><br>By default, the tag parity value will be automatically calculated. For test purposes, the parity/ECC bits from the *TagLoN* register will be used if $ErrCtl_{PO}$ is set.<br><br>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* register associated with the cache be initialized first. | Yes |
| 2#011 | I,D,T | Reserved | Unspecified | Executed as a no-op | No |
| 2#011 | S | Index Store Data | Index | Write the *DataHi2* and *DataLo2* Coprocessor 0 register contents at the way and dword index specified.<br><br>The ECC bits are always generated by the hardware (if present) | Yes |

**Table 15.24 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST,SPR] Cleared (Continued)**

| Code | Caches | Name | Effective Address Operand Type | Operation | Implemented? |
|---|---|---|---|---|---|
| 2#100 | All | Hit Invalidate | Address | If the cache line contains the specified address, set the state of the cache line 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 |
| 2#101 | I | Fill | Address | Fill the cache from the specified address.<br><br>The cache line is refetched even if it is already in the cache. | Yes |
| | D, S, T | Hit WriteBack Invalidate | Address | If the cache line 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 line to invalid. If the line is valid but not dirty, set the state of the line to invalid.<br><br>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 |
| 2#110 | D, S, T | Hit WriteBack | Address | If the cache line 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 |
| 2#111 | All | 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.<br><br>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 *TagLoN* register.<br><br>It is illegal to lock all ways at a given cache index. If all ways are locked, subsequent references to that index will displace one of the locked lines. | Yes |

**Table 15-1 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST] Set. ErrCtl[SPR] Cleared**

| Code | Caches | Name | Effective Address Operand Type | Operation | Implemented? |
|---|---|---|---|---|---|
| 2#001 | All | Index Load WS | Index | Read the WS RAM at the specified index into the *TagLoN* Coprocessor 0 register. | Yes |
| 2#010 | I | Index Store WS | Index | Update the WS RAM at the specified index from the *TagLo0* Coprocessor 0 register. | Yes |
| 2#010 | D | Index Store WS | Index | Update the WS RAM at the specified index from the *TagLo1* Coprocessor 0 register.<br>If $ErrCtl_{PO}$ is set, the dirty parity values in the *TagLo1* register will be written to the WS RAM. Otherwise, the parity will be calculated for the write data. | Yes |
| 2#010 | S | Index Store WS | Index | Update the WS RAM at the specified index from the *TagLo2* Coprocessor 0 register.<br>If $ErrCtl_{PO}$ is set, the dirty parity values in the *TagLo2* register will be written to the WS RAM. Otherwise, the parity will be calculated for the write data. | Yes |
| 2#011 | I | Index Store Data | Index | Write the *DataHi0* and *DataLo0* Coprocessor 0 register contents at the way and dword index specified.<br>If $ErrCtl_{PO}$ is set, $ErrCtl_{PI}$ is used for the parity value. Otherwise, the parity value is calculated for the write data.<br>If $ErrCtl_{PCO}$ is set, $ErrCtl_{PCI}$ is used for the precode values. Otherwise, the precode values will be calculated based on the write data. | Yes |
| 2#011 | D | Index Store Data | Index | Write the *DataLo1* Coprocessor 0 register contents at the way and word index specified.<br>If $ErrCtl_{PO}$ is set, $ErrCtl_{PD}$ is used for the parity value. Otherwise, the parity value is calculated for the write data. | Yes |
| 2#011 | S | Index Store ECC | Index | Write the *DataLo2* Coprocessor 0 register contents to the ECC bits at the way and dword index specified. | Yes |
| All Others | All | | | Other codes should not be used while $ErrCtl_{WST}$ is set. | |

**Table 15.25 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[SPR] Set, ErrCtl[WST] Cleared**

| Code | Caches | Name | Effective Address Operand Type | Operation | Implemented? |
|---|---|---|---|---|---|
| 2#001 | I | Index Load Tag | Index | Read the SPRAM tag at the specified index into the *TagLo0* Coprocessor 0 register. Also read the instruction data and precode information corresponding to the byte index into the *DataHi*,*DataLo*, and *ErrCtl* registers | Yes |
| 2#001 | D | Index Load Tag | Index | Read the SPRAM tag at the specified index into the *TagLo1* Coprocessor 0 register. | Yes |
| 2#010 | I, D | Index Store Tag | Index | Update the SPRAM tag at the specified index from the *TagLoN* Coprocessor 0 register. | Yes |
| 2#011 | I | Index Store Data | Index | Write the *DataLo0* and *DataHi0* Coprocessor 0 register contents into the SPRAM at the dword index specified. | Yes |
| 2#011 | D | Index Store Data | Index | Write the *DataLo1* Coprocessor 0 register contents into the SPRAM at the word index specified. | Yes |
| All Others | I,D | | | Other codes should not be used while $ErrCtl_{SPR}$ is set. | |
| All | S,T | | | Secondary and Tertiary operations should not be performed while $ErrCtl_{SPR}$ is set. | |

**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 operation requires an address, and that address is uncacheable.

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

**Operation:**

```
vAddr ← GPR[base] + sign_extend(offset)
(pAddr, uncached) ← AddressTranslation(vAddr, DataReadReference)
CacheOp(op, vAddr, pAddr)
```

**Exceptions:**

TLB Refill Exception.

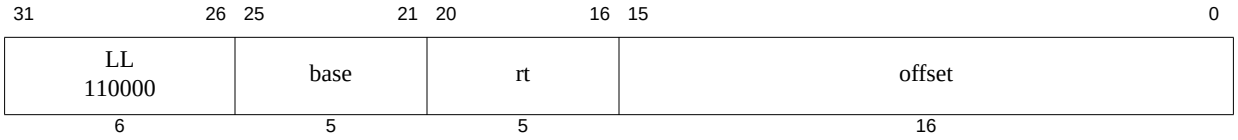
TLB Invalid Exception

Coprocessor Unusable Exception

Address Error Exception

Cache Error Exception

Bus Error Exception




**Format:** `LL rt, offset(base)` **MIPS32**

**Purpose:** Load Linked Word

To load a word from memory for an atomic read-modify-write

**Description:** `GPR[rt] ← memory[GPR[base] + offset]`

The LL and SC instructions provide the primitives to implement atomic read-modify-write (RMW) operations for synchronizable memory locations.

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and written into GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address.

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 synchronizable by all processors and I/O devices sharing the location; if it is not, the result in **UNPREDICTABLE**. Which storage is synchronizable is a function of both CPU and system implementations. See the documentation of the SC instruction for the formal definition.

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the effective address is nonzero, an Address Error exception occurs.

**Operation:**

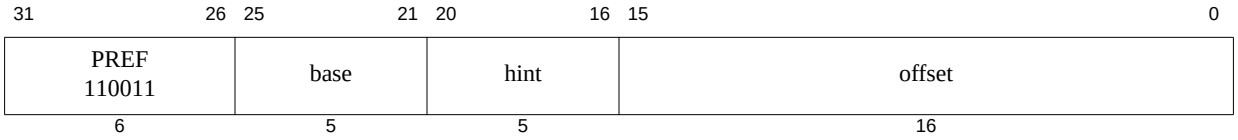
```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rt] ← memword
LLbit ← 1
```

**Exceptions:**

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

**Programming Notes:**

There is no Load Linked Word Unsigned operation corresponding to Load Word Unsigned.




**Format:** `PREF hint,offset(base)` **MIPS32**

**Purpose:** Prefetch

To move data between memory and cache.

**Description:** `prefetch_memory(GPR[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 except for PrepareForStore, 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, including TLB 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 moved, some action that is not architecturally visible, such as writeback of a dirty cache line, can take place.

It is implementation dependent whether a Bus Error or Cache Error exception is reported if such an error is detected as a by-product of the action taken by the PREF instruction.

PREF neither generates a memory operation nor modifies the state of a cache line for a location with an *uncached* memory access type, whether this type is specified by the address segment (e.g., kseg1), the programmed coherency attribute of a segment (e.g., the use of the K0, KU, or K23 fields in the *Config* register), or the per-page coherency attribute provided by the TLB.

If PREF results in a memory operation, the memory access type and coherency attribute used for the operation are determined by the memory access type and coherency attribute 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.

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

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.

### Table 15.26 Values of *hint* Field for PREF Instruction

| Value | Name | Data Use and Desired Prefetch Action |
|---|---|---|
| 0 | load | Use: Prefetched data is expected to be read (not modified).<br>Action: Fetch data as if for a load. |
| 1 | store | Use: Prefetched data is expected to be stored or modified.<br>Action: Fetch data as if for a store. |

**Table 15.26 Values of *hint* Field for PREF Instruction**

| | | |
|---|---|---|
| 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.<br>Action: Fetch data as if for a store. LRU replacement information is ignored and data is placed in way 0 of the cache, so it will be displaced by other streamed prefetches and not displace retained prefetches. If way 0 is locked, the prefetch will be dropped. |
| 5 | store_streamed | Use: Prefetched data is expected to be stored or modified but not reused extensively; it “streams” through cache.<br>Action: Fetch data as if for a store. LRU replacement information is ignored and data is placed in way 0 of the cache, so it will be displaced by other streamed prefetches and not displace retained prefetches. If way 0 is locked, the prefetch will be dropped. |
| 6 | load_retained | Use: Prefetched data is expected to be read (not modified) and reused extensively; it should be “retained” in the cache.<br>Action: Fetch data as if for a load. LRU replacement information is used, but way 0 of the cache is specifically excluded. This prevents streamed prefetches from displacing the line. |
| 7 | store_retained | Use: Prefetched data is expected to be stored or modified and reused extensively; it should be “retained” in the cache.<br>Action: Fetch data as if for a store. LRU replacement information is used, but way 0 of the cache is specifically excluded. This prevents streamed prefetches from displacing the line. |
| 8-24 | Reserved | Reserved - treated as a NOP. |
| 25 | writeback_invalidate (also known as “nudge”) | Use: Data is no longer expected to be used.<br>Action: Schedule a writeback of any dirty data. The cache line is marked as invalid upon completion of the writeback. If cache line is clean or locked, no action is taken. |
| 26-29 | Reserved | Reserved - treated as a NOP. |
| 30 | PrepareForStore | Use: Prepare the cache for writing an entire line, without the overhead involved in filling the line from memory.<br>Action: If the reference hits in the cache, no action is taken. If the reference misses in the cache, a line is selected for replacement, any valid and dirty victim is written back to memory, the entire line is filled with zero data, and the state of the line is marked as valid and dirty.<br>Programming Note: Because the cache line is filled with zero data only on a cache miss, software must not assume that this action, in and of itself, can be used as a fast bzero-type function. |
| 31 | Reserved | Reserved - treated as a NOP. |

**Restrictions:**

None

**Operation:**

```
vAddr ← GPR[base] + sign_extend(offset)
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
```

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

**Exceptions:**

Bus Error, Cache Error

Prefetch does not take any TLB-related or address-related exceptions under any circumstances.

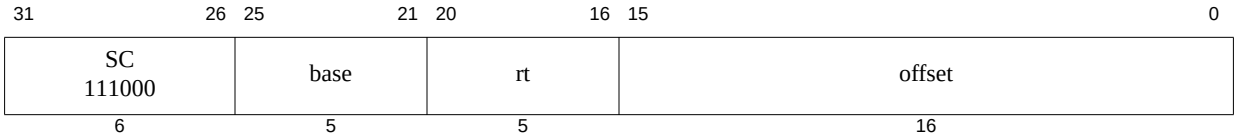
**Programming Notes:**

Prefetch cannot move data to or 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. A prefetch may be used using an address pointer before the validity of the pointer is determined without worrying about an addressing exception.

It is implementation dependent whether a Bus Error or Cache Error exception is reported if such an error is detected as a by-product of the action taken by the PREF instruction. Typically, this only occurs in systems which have high-reliability requirements.

Prefetch operations have no effect on cache lines that were previously locked with the CACHE instruction.




**Format:** `SC rt, offset(base)` **MIPS32**

**Purpose:** Store Conditional Word

To store a word to memory to complete an atomic read-modify-write

**Description:** `if atomic_update then memory[GPR[base] + offset] ← GPR[rt], GPR[rt] ← 1 else GPR[rt] ← 0`

The LL and SC instructions provide primitives to implement atomic read-modify-write (RMW) operations for synchronizable memory locations.

The 32-bit word in GPR *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 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 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 memory access instruction (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 **UNPREDICTABLE**:

- Execution of SC must have been preceded by execution of an LL instruction.
- An RMW sequence executed without intervening events that would cause the SC to fail 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.

**Restrictions:**

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

**Operation:**

```
vAddr ← sign_extend(offset) + GPR[base]
```

```
if vAddr1..0 ≠ 0^2 then
   SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
dataword ← GPR[rt]
if LLbit then
   StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
endif
GPR[rt] ← 0^31 || LLbit
```

**Exceptions:**

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

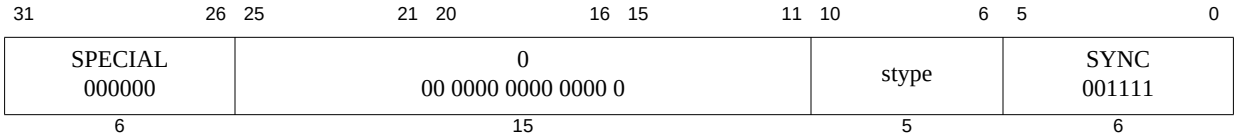
**Programming Notes:**

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.




**Format:** `SYNC (stype = 0 implied)` **MIPS32**

**Purpose:** Synchronize Shared Memory

To order loads and stores.

**Description:**

*Simple Description:*

- 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 (in Release 1 of the Architecture) or EHB (in Release 2 of the Architecture), 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 for future extensions to the architecture. A value of zero will always be defined such that it performs all defined synchronization operations. Non-zero values may be defined to remove some synchronization operations. As such, software should never use a non-zero value of the *stype* field, as this may inadvertently cause future failures if non-zero values remove synchronization operations.
- The SYNC instruction stalls until all loads, stores, refills are completed and all write buffers are empty.
- If the $Config7_{ES}$ bit is set, executing a SYNC instruction will cause a synchronizing transaction on the external bus. This will be a read with the OC_MReqInfo[3] bit set. Handling of this transaction is system dependent, but a typical system will flush any external write buffers and complete all pending transactions before completing the SYNC.

**Restrictions:**

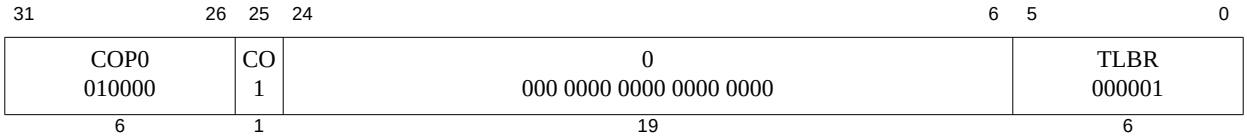
None

**Operation:**

```
SyncOperation(stype)
```

**Exceptions:**

None




**Format:** `TLBR` **MIPS32**

**Purpose:** Read Indexed TLB Entry

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. In Release 1 of the Architecture, it is implementation dependent whether multiple TLB matches are detected on a TLBR. However, implementations are strongly encouraged to report multiple TLB matches only on a TLB write. In Release 2 of the Architecture, multiple TLB matches may only be reported on a TLB write. 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.

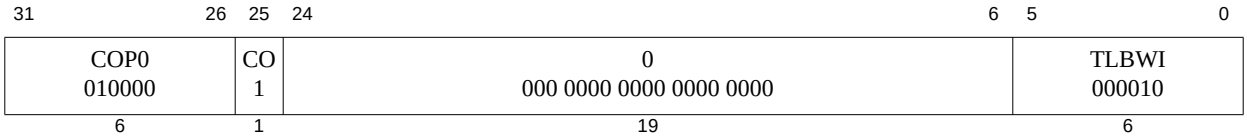
If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

**Operation:**

```
i ← Index
if i > (TLBEntries - 1) then
   UNDEFINED
endif
PageMask_Mask ← TLB[i]_Mask
EntryHi ←
          TLB[i]_VPN2 ||
          0^5 || TLB[i]_ASID
EntryLo1 ← 0^2 ||
          TLB[i]_PFN1 ||
          TLB[i]_C1 || TLB[i]_D1 || TLB[i]_V1 || TLB[i]_G
EntryLo0 ← 0^2 ||
          TLB[i]_PFN0 ||
          TLB[i]_C0 || TLB[i]_D0 || TLB[i]_V0 || TLB[i]_G
```

**Exceptions:**

Coprocessor Unusable




**Format:** `TLBWI` **MIPS32**

**Purpose:** Write Indexed TLB Entry

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. It is implementation dependent whether multiple TLB matches are detected on a TLBWI. In such an instance, a Machine Check Exception is signaled. See Section 5.4.1 "Hits, Misses, and Multiple Matches" for the cases in which a 34K core will signal a Machine Check. In Release 2 of the Architecture, multiple TLB matches may only be reported on a TLB write. 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.

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

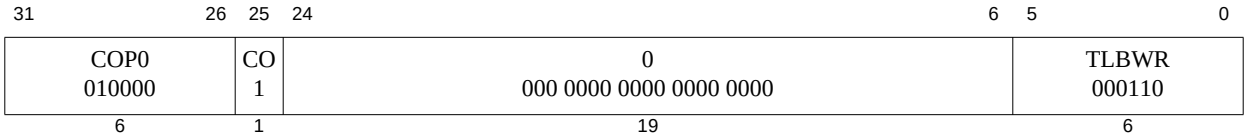
**Operation:**

```
i ← Index
TLB[i]Mask ← PageMaskMask
TLB[i]VPN2 ← EntryHiVPN2
TLB[i]ASID ← EntryHiASID
TLB[i]G ← EntryLo1G and EntryLo0G
TLB[i]PFN1 ← EntryLo1PFN
TLB[i]C1 ← EntryLo1C
TLB[i]D1 ← EntryLo1D
TLB[i]V1 ← EntryLo1V
TLB[i]PFN0 ← EntryLo0PFN
TLB[i]C0 ← EntryLo0C
TLB[i]D0 ← EntryLo0D
TLB[i]V0 ← EntryLo0V
```

**Exceptions:**

Coprocessor Unusable

Machine Check




**Format:** `TLBWR` **MIPS32**

**Purpose:** Write Random TLB Entry

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. It is implementation dependent whether multiple TLB matches are detected on a TLBWR. In such an instance, a Machine Check Exception is signaled. In Release 2 of the Architecture, multiple TLB matches may only be reported on a TLB write. See Section 5.4.1 "Hits, Misses, and Multiple Matches" for the cases in which a 34K core will signal a Machine Check. 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:**

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

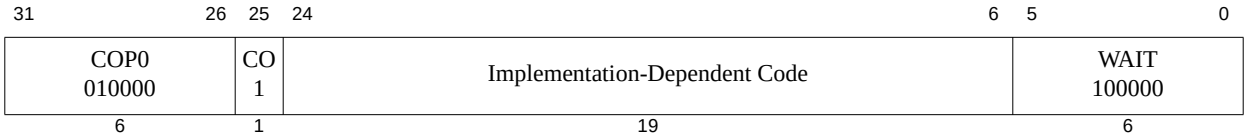
**Operation:**

```
i ← Random
TLB[i]Mask ← PageMaskMask
TLB[i]VPN2 ← EntryHiVPN2
TLB[i]ASID ← EntryHiASID
TLB[i]G ← EntryLo1G and EntryLo0G
TLB[i]PFN1 ← EntryLo1PFN
TLB[i]C1 ← EntryLo1C
TLB[i]D1 ← EntryLo1D
TLB[i]V1 ← EntryLo1V
TLB[i]PFN0 ← EntryLo0PFN
TLB[i]C0 ← EntryLo0C
TLB[i]D0 ← EntryLo0D
TLB[i]V0 ← EntryLo0V
```

**Exceptions:**

Coprocessor Unusable

Machine Check




**Format:** `WAIT` **MIPS32**

**Purpose:** Enter Standby Mode

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 core does not use the code field in this instruction.

If the pipeline restarts as the result of an enabled interrupt, that interrupt is taken between the WAIT instruction and the following instruction (EPC for the interrupt points at the instruction following the WAIT 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.

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

**Operation:**

```
I: Enter  lower power mode
I+1:/* Potential interrupt taken here */
```

**Exceptions:**

Coprocessor Unusable Exception

*Chapter 16*

# MIPS16e™ Application-Specific Extension to the MIPS32® Instruction Set

This chapter describes the MIPS16e ASE as implemented in the 34K core. Refer to Volume IV-a of the *MIPS32® Architecture Reference Manual* for a general description of the MIPS16e ASE as well as instruction descriptions.

This chapter covers the following topics:

- Section 16.1 "Instruction Bit Encoding"
- Section 16.2 "Instruction Listing"

## 16.1 Instruction Bit Encoding

Table 16.2 through Table 16.9 describe the encoding used for the MIPS16e ASE. Table 16.1 describes the meaning of the symbols used in the tables.

**Table 16.1 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 cause 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. |
| β | Operation or field codes marked with this symbol represent a valid encoding for a higher-order MIPS ISA level. Executing such an instruction cause a Reserved Instruction Exception. |
| θ | Operation or field codes marked with this symbol are available to licensed MIPS partners. To avoid multiple conflicting instruction definitions, the partner must notify MIPS Technologies, Inc. when one of these encodings is used. If no instruction is encoded with this value, executing such an instruction must cause a Reserved Instruction Exception (*SPECIAL2* encodings or coprocessor instruction encodings for a coprocessor to which access is allowed) or a Coprocessor Unusable Exception (coprocessor instruction encodings for a coprocessor to which access is not allowed). |
| σ | Field codes marked with this symbol represent an EJTAG support instruction and implementation of this encoding is optional for each implementation. If the encoding is not implemented, executing such an instruction must cause a Reserved Instruction Exception. If the encoding is implemented, it must match the instruction encoding as shown in the table. |
| ε | Operation or field codes marked with this symbol are reserved for MIPS Application Specific Extensions. If the ASE is not implemented, executing such an instruction must cause a Reserved Instruction Exception. |
| ϕ | Operation or field codes marked with this symbol are obsolete and will be removed from a future revision of the MIPS64 ISA. Software should avoid using these operation or field codes. |

### Table 16.2 MIPS16e Encoding of the Opcode Field

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

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

### Table 16.3 MIPS16e JAL(X) Encoding of the x Field

| **x** | bit 26 | |
|---|---|---|
| | 0 | 1 |
| | JAL | JALX |

### Table 16.4 MIPS16e SHIFT Encoding of the f Field

| **f** | bits 1..0 | | | |
|---|---|---|---|---|
| | 0 | 1 | 2 | 3 |
| | 00 | 01 | 10 | 11 |
| | SLL | β | SRL | SRA |

### Table 16.5 MIPS16e RRI-A Encoding of the f Field

| **f** | bit 4 | |
|---|---|---|
| | 0 | 1 |
| | ADDIU[1] | β |

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

### Table 16.6 MIPS16e I8 Encoding of the funct Field

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

1. The SWRASP function is used by the SW ra, offset(sp) instruction

2. The ADJSP function is used by the ADDIU sp, immediate instruction
3. The MOV32R function is used by the MOVE r32, rz instruction
4. The MOVR32 function is used by the MOVE ry, r32 instruction

**Table 16.7 MIPS16e RRR Encoding of the f Field**

| **f** | bits 1..0 | | | |
|---|---|---|---|---|
| | 0 | 1 | 2 | 3 |
| | 00 | 01 | 10 | 11 |
| | β | ADDU | β | SUBU |

**Table 16.8 MIPS16e RR Encoding of the Funct Field**

| **funct** | | bits 2..0 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| bits 4..3 | | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 0 | 00 | *J(AL)R(C)* δ | SDBBP | SLT | SLTU | SLLV | BREAK | SRLV | SRAV |
| 1 | 01 | β | * | CMP | NEG | AND | OR | XOR | NOT |
| 2 | 10 | MFHI | *CNVT* δ | MFLO | β | β | * | β | β |
| 3 | 11 | MULT | MULTU | DIV | DIVU | β | β | β | β |

**Table 16.9 MIPS16e I8 Encoding of the s Field when funct=SVRS**

| **s** | bit 7 | |
|---|---|---|
| | 0 | 1 |
| | RESTORE | SAVE |

**Table 16.10 MIPS16e RR Encoding of the ry Field when funct=J(AL)R(C)**

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

**Table 16.11 MIPS16e RR Encoding of the ry Field when funct=CNVT**

| **ry** | bits 7..5 | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| | ZEB | ZEH | β | * | SEB | SEH | β | * |

## 16.2 Instruction Listing

Table 16.12 through 16.19 list the MIPS16e instruction set.

**Table 16.12 MIPS16e Load and Store Instructions**

| Mnemonic | Instruction | Extensible Instruction |
|---|---|---|
| LB | Load Byte | Yes |
| LBU | Load Byte Unsigned | Yes |
| LH | Load Halfword | Yes |
| LHU | Load Halfword Unsigned | Yes |
| LW | Load Word | Yes |
| SB | Store Byte | Yes |
| SH | Store Halfword | Yes |
| SW | Store Word | Yes |

**Table 16.13 MIPS16e Save and Restore Instructions**

| Mnemonic | Instruction | Extensible Instruction |
|---|---|---|
| RESTORE | Restore Registers and Deallocate Stack Frame | Yes |
| SAVE | Save Registers and Setup Stack Frame | Yes |

**Table 16.14 MIPS16e ALU Immediate Instructions**

| Mnemonic | Instruction | Extensible Instruction |
|---|---|---|
| ADDIU | Add Immediate Unsigned | Yes |
| CMPI | Compare Immediate | Yes |
| LI | Load Immediate | Yes |
| SLTI | Set on Less Than Immediate | Yes |
| SLTIU | Set on Less Than Immediate Unsigned | Yes |

**Table 16.15 MIPS16e Arithmetic Two or Three Operand Register Instructions**

| Mnemonic | Instruction | Extensible Instruction |
|---|---|---|
| ADDU | Add Unsigned | No |
| AND | AND | No |
| CMP | Compare | No |

**Table 16.15 MIPS16e Arithmetic Two or Three Operand Register Instructions**

| Mnemonic | Instruction | Extensible Instruction |
|---|---|---|
| MOVE | Move | No |
| NEG | Negate | No |
| NOT | Not | No |
| OR | OR | No |
| SEB | Sign-Extend Byte | No |
| SEH | Sign-Extend Halfword | No |
| SLT | Set on Less Than | No |
| SLTU | Set on Less Than Unsigned | No |
| SUBU | Subtract Unsigned | No |
| XOR | Exclusive OR | No |
| ZEB | Zero-Extend Byte | No |
| ZEH | Zero-Extend Halfword | No |

**Table 16.16 MIPS16e Special Instructions**

| Mnemonic | Instruction | Extensible Instruction |
|---|---|---|
| BREAK | Breakpoint | No |
| SDBBP | Software Debug Breakpoint | No |
| EXTEND | Extend | No |

**Table 16.17 MIPS16e Multiply and Divide Instructions**

| Mnemonic | Instruction | Extensible Instruction |
|---|---|---|
| DIV | Divide | No |
| DIVU | Divide Unsigned | No |
| MFHI | Move From HI | No |
| MFLO | Move From LO | No |
| MULT | Multiply | No |
| MULTU | Multiply Unsigned | No |

**Table 16.18 MIPS16e Jump and Branch Instructions**

| Mnemonic | Instruction | Extensible Instruction |
|---|---|---|
| B | Branch Unconditional | Yes |

**Table 16.18 MIPS16e Jump and Branch Instructions**

| Mnemonic | Instruction | Extensible Instruction |
|---|---|---|
| BEQZ | Branch on Equal to Zero | Yes |
| BNEZ | Branch on Not Equal to Zero | Yes |
| BTEQZ | Branch on T Equal to Zero | Yes |
| BTNEZ | Branch on T Not Equal to Zero | Yes |
| JAL | Jump and Link | No |
| JALR | Jump and Link Register | No |
| JALRC | Jump and Link Register Compact | No |
| JALX | Jump and Link Exchange | No |
| JR | Jump Register | No |
| JRC | Jump Register Compact | No |

**Table 16.19 MIPS16e Shift Instructions**

| Mnemonic | Instruction | Extensible Instruction |
|---|---|---|
| SRA | Shift Right Arithmetic | Yes |
| SRAV | Shift Right Arithmetic Variable | No |
| SLL | Shift Left Logical | Yes |
| SLLV | Shift Left Logical Variable | No |
| SRL | Shift Right Logical | Yes |
| SRLV | Shift Right Logical Variable | No |

*Appendix A*

# Revision History

In the left hand page margins of this document you may find vertical change bars to note the location of significant changes to this document since its last release. Significant changes are defined as those which you should take note of as you use the MIPS IP. Changes to correct grammar, spelling errors or similar may or may not be noted with change bars. Change bars will be removed for changes which are more than one revision old.

Please note: Limitations on the authoring tools make it difficult to place change bars on changes to figures. Change bars on figure titles are used to denote a potential change in the figure itself. Certain parts of this document (Instruction set descriptions, EJTAG register definitions) are references to Architecture specifications, and the change bars within these sections indicate alterations since the previous version of the relevant Architecture document.

| Revision | Date | Description |
|---|---|---|
| 1.00 | August 25, 2006 | Initial version. |
| 1.01 | August 25, 2006 | Replacement for 1.00 - This is the correct version for external release. |
| 1.02 | December 19, 2006 | • Added ability to modify exception base when $Status_{BEV}$ is 1<br>• Fixed value of ProcessorID field in PrID register<br>• Cleaned up ITU description - use view names from MT docs, add addresses for the views<br>• Added missing MT perfcounter information<br>• Added descriptions for perfcounter events<br>• Updated implementation-dependent instruction descriptions<br>• Updated description for L2 bypass bit in Config2<br>• Enhanced description for External SYNC indication in Config7<br>• Updated ErrorEPC description for value held on reset exception |