MIPS32® 74K™ Processor Core Family Software User's Manual

Document Number: MD00519
Revision 01.03
November 14, 2008

MIPS Technologies, Inc.
955 East Arques Avenue
Sunnyvale, CA 94085-4521

Template: nB1.03, Built with tags: 2B

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction to the MIPS32® 74K™

The 74K™ 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 74K 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 74K 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 74K family has two members: the MIPS32® 74Kc™ core and the MIPS32® 74Kf™ core:

- The 74Kc 32-bit RISC core is optimized for high-performance applications.

- The 74Kf core adds an IEEE-754 compliant floating point unit.

The term *74K core*, as used in this document, generally refers to all cores in the 74K core family. When referring to characteristics unique to an individual family member, the specific core type is identified.

The core implements the MIPS32 Release 2 Instruction Set Architecture (ISA). It also implements the following Application-Specific Extensions (ASEs):

- The MIPS® DSP Application-Specific Extension (ASE) is optimized for signal processing applications.

- The MIPS16e™ Application-Specific Extension (ASE) is optimized for code compression.

The 74K core achieves its high performance through the implementation of advanced superscalar and out-of-order dispatch techniques in a deeply pipelined implementation of the MIPS32® architecture. The superscalar dispatch allows the core to dispatch two instructions per cycle to two pipelines: a 15-stage AGEN pipeline that executes all load/store and control transfer instructions, and a 14-stage ALU pipeline that executes all the rest of the instructions (arithmetic, logic, and general computations). The out-of-order approach allows each pipeline to operate independently and select from a pool of instructions for dispatch; and to ensure the availability of two instructions for dispatch, twice that number of instructions are fetched every cycle from an instruction cache. The 74K core also implements sophisticated branch prediction techniques that minimize the cost of a mispredicted branch in such a deeply pipelined core.

On the 74K core, instruction and data caches are configurable as 0, 16, 32, or 64 KB in size. Each cache is organized as 4-way set-associative data structure. Each cache is organized as 4-way set associative structure. The 74K core supports prefetching of sequential cache lines on instruction cache miss. The extent of prefetch can be configured via software to prefetch between 0 and 2 additional lines. The data cache features non-blocking load misses and can handle up to 9 outstanding load misses in up to 4 unique cache lines. On a cache miss, the processor can continue executing instructions while the load data is being fetched, until a dependent instruction is reached. Both caches are virtually indexed and physically tagged.

The MMU of the 74K core may contain a 4-entry instruction TLB (ITLB) and a dual-entry joint TLB (JTLB), with variable page sizes. The JTLB can be configured to have 16, 32, 48, or 64 dual entries. Optionally, the TLB can be replaced with a simplified fixed mapping (FM) translation mechanism for applications that do not require the full capabilities of a 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, as well as software breakpoints using 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 can be optionally included. In addition, optional PDtrace™ hardware 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).

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

1.1 74K™ Core Features

1.1.1 Pipeline

- Superscalar, dual-issue core supports 2 integer execution pipes
  - 15-stage AGEN Pipe: supports load/store, control transfer. and conditional move instructions
  - 14-stage ALU Pipe: supports all other arithmetic, logic, and computation instructions
- Out-of-order integer instruction dispatch
  - Selects one of eight instructions in each pipe
- Multiply Divide Unit
  - Offshoot of the ALU Pipe
  - Maximum issue rate of one 32x32 multiply per clock
  - Early-in divide control. Minimum 11, maximum 50-cycle clock latency on divide
- Dual- Issue Floating Point Unit supports two pipes (74Kf only)
  - Arithmetic pipe
  - To/From or Data transfer pipe
  - Floating point instruction dispatch is maintained in-order
- Dynamic branch/return prediction
  - Majority predictor featuring 3 tables of 256 entries, each with global history

    - 8-entry return prediction stack

1.1.2 Instruction Set

- 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: 1 to 4 sets are supported
    - Bit field manipulation instructions
- MIPS DSP ASE Rev 2
    - Fractional data types (Q15, Q31)
    - Saturating arithmetic
    - SIMD instructions operated on 2x16b or 4x8b simultaneously
    - 3 additional pairs of accumulator registers
- 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)
- Floating Point Instruction support (74Kf)

    - IEEE-754 compliant floating point unit
    - Compliant with MIPS 64-bit FPU standards
    - Supports single and double precision datatypes
- CorExtend™ User-Defined Instruction capability
    - Support for the CorExtend feature allows users to define and add instructions to the core
    - 3, 5, or multi-cycle latencies supported
    - Source operations from register, immediate field, or local state
    - Destination to a register or local state
    - Interface to multiply-divide unit, allowing sharing of accumulator registers

1.1.3 Memory Management, Caches, and Scratchpad Memory

- Standard Memory Management Unit
    - 16, 32, 48, or 64 dual-entry MIPS32-style JTLB with variable page sizes
    - 4-entry instruction TLB
- Optional Memory Management Unit
    - Simple Fixed Mapping Translation (FM)
    - Address spaces mapped using register bits
- Caches
    - Individually configurable instruction and data caches
    - Instruction cache sizes of 0, 16, 32, or 64 KB
    - Data cache sizes of 0, 16, 32, or 64 KB
    - Data cache access widths can be configured to be 64b or 128b
    - 4-way set associative
    - 256-bit (32-byte) cache line size
    - Configurable instruction cache line prefetch, to fetch between 0 to 2 additional cache lines on a miss
    - Non-blocking data cache
    - Up to 4 data cache line misses or 9 unique load misses
    - Supports writeback with write-allocation and write-through without write-allocation

    - Virtually indexed, physically tagged
    - Cache line locking support
    - Configurable support for parity
    - Support for front-side external L2 cache
    - Configurable support for elimination of aliases in 32 and 64KB data caches
- Independent Instruction and Data ScratchPad RAMs
    - Address range of 4K - 1MB supported
    - 64-bit OCP interfaces for external access

1.1.4 Interfaces

- OCP Interface
    - 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 easy bridging to other bus standards
    - Extensions for management of front side L2 cache
    - Critical data first and sub-block ordering support
- IEEE standard JTAG interface

1.1.5 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

1.1.6 Debug

- EJTAG Debug Support via JTAG interface
    - CPU control with start, stop, and single stepping

    - Software breakpoints via the SDBBP instruction
    - Optional hardware breakpoints on virtual addresses: 4 instruction and 2 data breakpoints
    - Test Access Port (TAP) facilitates high-speed download of application code
    - Optional MIPS PDtrace™ hardware to enable real-time tracing of executed code

1.1.7 Other

- R4000 Style Privileged Resource Architecture
    - Count/Compare registers for real-time timer interrupts
    - Instruction and data watch registers for software breakpoints
- Relocatable Bootstrap Exception Vector support
    - Support for hardware selectable exception base in a multi-core environment

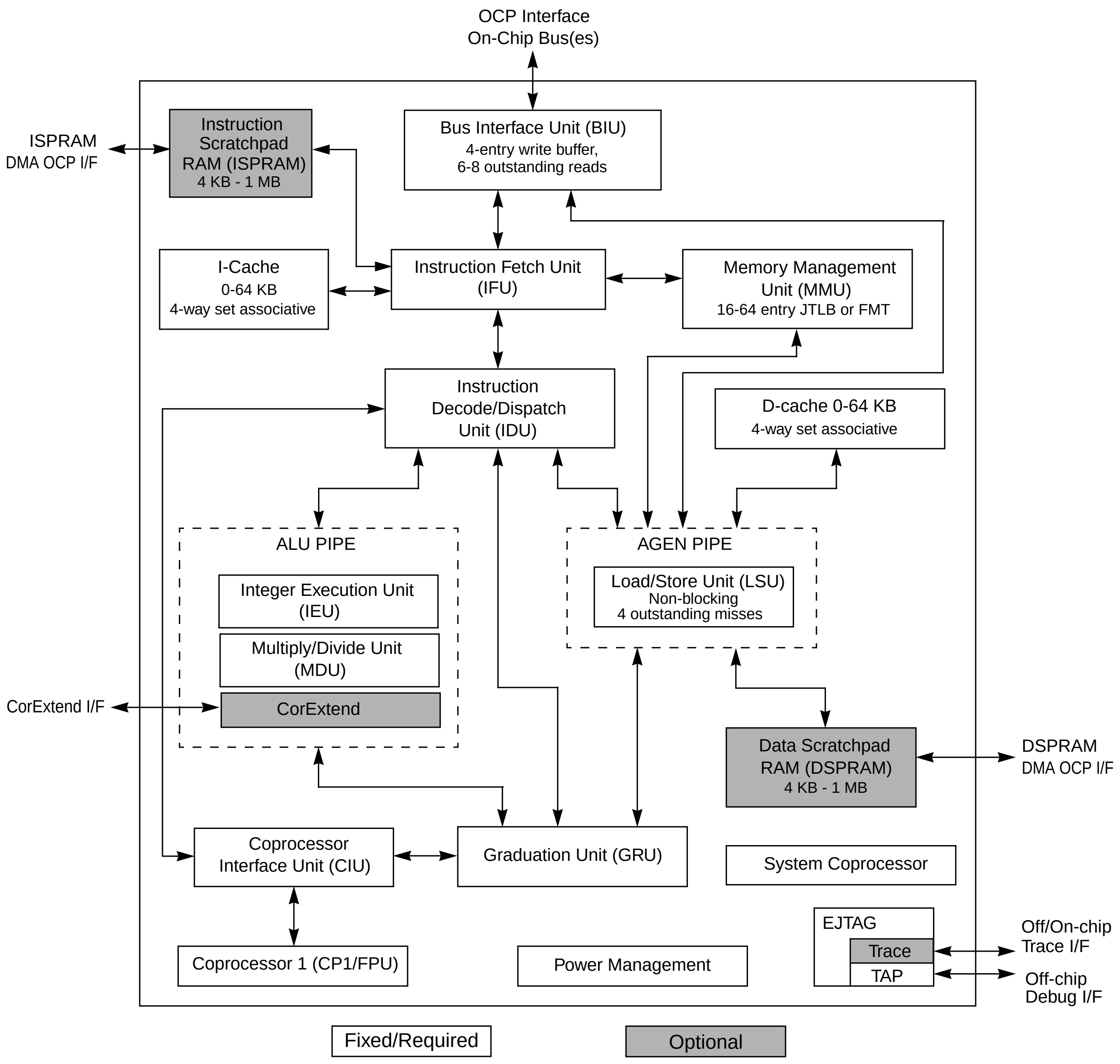
1.2 74K™ Core Block Diagram

The 74K core contains a number of blocks, shown in the block diagram in Figure 1.2. The major blocks are:

- Instruction Fetch Unit (IFU)
- Instruction Cache (I-cache)
- Instruction Decode and Dispatch Unit (IDU)
- Instruction Execution Unit (IEU)
- Multiply/Divide Unit (MDU)
- CorExtend® User Defined Instructions (UDI)
- System Control Coprocessor (CP0)
- Memory Management Unit (MMU)
- Load Store Unit (LSU)
- Data Cache (D-cache)
- Graduation unit (GRU)
- Bus Interface Unit (BIU)
- Coprocessor Interface unit (CIU) (only in 74Kf)
- Floating Point Unit (FPU) (only in 74Kf)

- Power Management
- Enhanced JTAG (EJTAG) Controller

Figure 1.1 74K™ Core Block Diagram



The functional blocks shown in Figure 1.2 are described in the following subsections.

1.2.1 Instruction Fetch Unit (IFU)

The Instruction Fetch Unit (IFU) is responsible for fetching instructions from the instruction cache/memory and providing them to the IDU. The IFU can fetch up to 4 instructions at a time from an aligned fetch address. The IFU has a 4-entry microTLB which is used to translate the virtual fetch address into the physical fetch address. This translated physical address is used to compare against tags in the instruction cache to determine a hit.

The IFU uses majority branch prediction based on gshare predictors. There are three, 256-entry Branch History Tables that are indexed by different combinations of instruction PC and Global History. The majority of these three predictions is used to determine the predicted direction of a conditional branch. The IFU also has a 8-entry Return Prediction Stack to predict subroutine return addresses.

There is a 12-entry Instruction Buffer to decouple the instruction fetch from execution. Up to 4 instructions at a time can be written into this buffer, but a maximum of 2 instructions at a time can be read from this buffer by the IDU.

The 74K core includes supports for the MIPS16e ASE. This ASE improves code density through the use of 16-bit encoding of many MIPS32 instructions plus some MIPS16e-specific instructions. The IFU contains the logic for the handling of MIPS16e instructions.

1.2.2 Instruction Cache

The instruction cache is an on-chip memory block of 0/16/32/64 KB, with 4-way associativity. The instruction 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.

A tag entry holds 21 bits of physical address, a valid bit, a lock bit, and an optional parity bit. There are 7 precode bits per instruction pair, making a total of 28 bits per tag entry. The data array line consists of 256 bits (8 MIPS32 instructions) of data. Each instruction doubleword (64 bits) has 8 bits of byte parity. The IFU interface consists of 128 bits (4 MIPS32 instructions) with 16 bits of parity. The LRU replacement bits (6 bits) are shared among the 4 ways of the data and tag array and are stored in a separate array.

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.

The instruction cache also supports cache line prefetching on miss. This feature can be configured by software to prefetch anywhere between 0 and 2 lines. The control for this prefetch resides in the Coprocessor Register field *Config7*$_{PREF}$. Refer to Appendix 7, “Config7 (CP0 Register 16, Select 7): CPU-specific Configuration” on page 173 for details on programming this feature. The default setting will fetch one additional cache line. This can be modified by the user of the core. Increasing the number of cache lines prefetched to 2, will typically provide increased performance for applications with a high instruction cache miss rate. Decreasing the number of cache lines prefetched to 0 may be appropriate for applications with very low instruction cache miss rates or if there is a desire to reduce memory bandwidth for power or other reasons.

1.2.3 Instruction Decode/Dispatch Unit (IDU)

This unit is responsible for receiving instructions from the IFU and dispatching them out-of-order to the execution units when their operands and required resources are available. Up to two instructions per cycle can be received in-order from the IFU. As the IDU dispatches instructions speculatively and out-of-order, results from these instructions are stored in temporary storage buffers referred to as *completion buffers*. The IDU assigns the completion buffer ID, as well as a separate instruction ID for each instruction. The instruction is also renamed by looking up in a Rename

Map, and the source registers are replaced (if necessary) by completion buffer IDs of producer instructions, so that operands may be bypassed as soon as possible.

Renamed instructions are assigned to one of two pipes (ALU or AGEN) and written into the Decode and Dispatch Queue (DDQ) for that pipe. The oldest instruction that has all the operands ready and meets all resource requirements is dispatched independently to the corresponding pipe. Instructions may be dispatched out-of-order relative to program order. Dispatched instructions do not stall in the pipe and write the results into the completion buffer.

The IDU also keeps track of the progress of the instruction through the pipe, updating the availability of operands in the Rename Map and in all dependent instructions in the DDQ.

The IDU also writes the instruction ID, completion buffer ID, and related information into structures in the Graduation Unit (GRU). The GRU reads instructions and corresponding results from the completion buffer, graduates the instructions, and updates the architectural state of the machine.

1.2.4 Instruction Execution Unit (IEU)

The Instruction Execution Unit implements the entire ALU pipe and parts of the AGEN pipe (parts of the AGEN pipe also reside in the LSU). The IEU provides data inputs to the multiply/divide unit (MDU) and CorExtend units and receives outputs from them. The LSU, MDU, and CorExtend Unit are described in subsequent sections.

The architecturally-defined General Purpose Registers (GPRs) reside in the IEU. In addition to these, the IEU also contains the completion buffers (CBs) used to store computed results. There is a dedicated completion buffer per pipeline. Each pipe of the IEU has input bypass muxes to select data from the GPRs, CBs, or from the pipeline when data forwarding is required. The IEU also contains the output muxes that generate final output data. In addition, the IEU has certain pipe-specific execution units described below.

ALU Pipe

The ALU pipe contains the ALU for performing arithmetic and logical operations, the Shifter, and the Leading Zero/One detector. The ALU pipe implements a subset of the DSP ASE instructions.

AGEN Pipe

The AGEN pipe contains the adder required for address computation in case of load/store and control transfer instructions. It also contains all the branch resolution logic.

The IEU also provides data inputs to the multiply/divide unit (MDU) and CorExtend units and receives outputs from them.

1.2.5 Multiply Divide Unit (MDU)

The multiply/divide unit implements the multiply and divide operations. This unit also executes multiply class instructions in the DSP ASE.

The MDU consists of a pipelined 32ξ32 multiplier, result/accumulation registers (HI and LO), a divide state machine, and the necessary multiplexors and control logic. The MDU supports execution of one multiply or multiply-accumulate operation every clock cycle. Divide operations are implemented with a simple 1 bit per clock radix 2 iterative SRT algorithm.

1.2.6 CorExtend® User Defined Instructions (UDIs)

This module contains support for CorExtend user-defined instructions. These instructions are defined at build-time for the 74K core. This feature makes 15 instructions in the opcode map available for user-defined customer use, and the latency of each instruction can be selected to be 3, 5, or >5 cycles. 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. Further details regarding CorExtend can be found in the *CorExtend® Instruction Integrator's Guide for MIPS32 74K™ Cores* (MD00523).

Refer to Table 13.5 for a specification of the opcode map available for user-defined instructions.

1.2.7 Load Store Unit (LSU)

The Load Store Unit, as the name implies, is primarily responsible for the implementation of Load/Store instructions. In addition to the Load/Store instructions, it also implements the Prefetch, CACHE, and some other special instructions. The LSU contains all the control logic for the data cache. In addition it contains several holding structures for address and data information. These are primarily a 4 cache line Fill Store Buffer (FSB) to service cache line misses, a 9-entry Load Data Queue (LDQ) to support 9 load misses, a 14-entry Load Store Queue (LSQ), and a 10-entry Load Store Graduation Buffer (LSGB) to hold information for Loads and Store instructions in flight.

1.2.8 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), power management, and the enabling/disabling of interrupts. Configuration information such as cache size, set associativity, and the presence of build-time options are available by accessing the CP0 registers. Refer to Chapter 7, "CP0 Registers of the 74K™ Core" for more information on the CP0 registers. Refer to Chapter 11, "EJTAG Debug Support in the 74K™ Core" for more information on EJTAG debug registers.

1.2.9 Memory Management Unit (MMU)

The 74K core contains an MMU that interfaces between the execution unit and the cache controllers, shown in Figure . Although the 74K 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 74K core implements its MMU based on a Translation Lookaside Buffer (TLB). The TLB consists of two translation buffers: a configurable 16/32/48/64 dual-ported, dual-entry fully associative Joint TLB (JTLB) and a 4-entry fully associative Instruction TLB (ITLB).

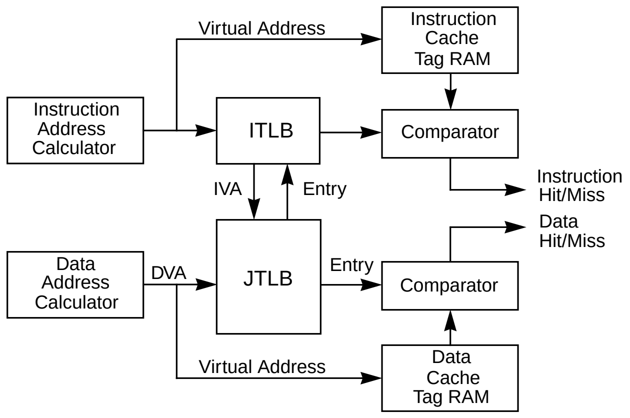
The ITLB, also referred to as the Instruction micro TLB, is managed by the hardware and is not visible to software. The ITLB contains a subset of the JTLB. When translating an instruction fetch address, the ITLB is accessed first. If there is no matching entry, the JTLB is used to translate the address and refill the ITLB. If the entry is not found in the JTLB, then an exception is taken.

In order to translate an address for a data access, the MMU looks in the JTLB directly, as there is no Data micro TLB present in the 74K core. The JTLB is dual-ported so as to avoid contention between instruction and data accesses.

The core optionally implements an FMT-based MMU instead of a TLB-based MMU. The FMT replaces the ITLB and JTLB blocks in Figure 1.2. The FMT performs a simple translation to obtain the physical address from the virtual address. Refer to Chapter 5, "Memory Management of the 74K™ 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



1.2.10 Data Cache

The data cache is an on-chip memory array of 0/16/32/64 KBytes. The cache is virtually indexed and physically tagged, allowing the virtual-to-physical address translation to occur in parallel with the cache access. The tag holds 20 bits of the physical address, a valid bit, a lock bit, and optionally a parity bit. For each entry there is also a corresponding 20- bit virtual tag. The virtual tag is used to determine the way selected on a cache access. As a result of this, data selection does not have to wait for the JTLB translation to complete.

The data entry is configurable to hold 64 or 128 bits of data per way, with optional parity per byte. The 128-bit option allows faster refill and evictions of the data cache and is recommended where data cache bandwidth is critical. In the 64-bit configuration, there are 4 data entries per tag entry, and in the 128-bit configuration there are 2 data entries for each tag entry. The tag and data entries exist for each way of the cache. A separate array holds the dirty and LRU bits (6b), dirty bits (4b), and optional dirty parity bits (4b) for all 4 ways.

A data cache locking mechanism is available that is similar to the mechanism in the instruction cache.

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

1.2.11 Scratchpad RAM

The 74K core allows blocks of scratchpad RAM to be attached to the load/store and instruction units. These allow low-latency access to a fixed block of memory.

1.2.12 Graduation Unit (GRU)

The Graduation Unit is responsible for graduating instructions in-order, even though they might have been dispatched and completed their result computation out-of-order. It does so by reading data and associated control information from the completion buffers in program order and committing them into architectural state in that same program order. It then releases any completion buffers and resources used by these instructions. The GRU is also responsible for evaluating the exception conditions reported by execution units and taking the appropriate exception.

1.2.13 Bus Interface Unit (BIU)

The Bus Interface Unit 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, 32-byte entries.

1.2.14 Coprocessor Interface Unit (CIU)

The coprocessor interface unit is responsible for maintaining an in-order interface between the integer core and the Floating Point Unit (FPU). The FPU is described in further detail in Chapter 3, “Floating-Point Unit of the 74Kf™ Core”.

1.2.15 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 74K™ Core” for more information on power management.

1.2.16 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 capability of debugging without debug code in the application, and for download of application code to the system.

An optional EJTAG feature is hardware breakpoints. A 74K core may have four instruction breakpoints and two data breakpoints, 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.

An optional MIPS Trace feature has been added to aid software debugging. The trace logic implements

PDtrace™ version 6, which allows tracing of the PC, load/store address, load/store data, and performance counter data, and also provides information about processor pipeline inefficiency. The trace information can be stored to either an on-chip trace memory or an off-chip trace probe. The optional on-chip trace memory can be configured in various sizes.These trace features provide a powerful software debugging mechanism. Refer to Chapter 11, “EJTAG Debug Support in the 74K™ Core” for more information on the EJTAG and tracing features.

Chapter 2

Pipeline of the 74K™ Core

The 74K processor core is a superscalar processor capable of issuing two integer instructions every clock cycle. The integer dual issue is achieved through two pipelines referred to as the ALU and AGEN pipelines. The 15-stage AGEN pipeline implements the memory transfer and control transfer class of instructions, and the 14-stage ALU pipeline implements all the rest of the instructions involving arithmetic, logic, and computation. The pipelines allows the processor to achieve high frequency while minimizing device complexity, reducing both cost and power consumption.

2.1 Integer Pipeline Description

The two primarily integer pipelines, the AGEN and ALU pipelines, are supported by a common front and back end. The common front end comprises of instruction fetch, decode, and dispatch, and represents the first eight stages of the pipeline, except in MIPS16e mode, where three more stages are present. The common back end comprises of instruction graduation and consumes the last two stages. The intermediate stages represent the instruction execution specific portions of the ALU and AGEN pipelines.

The two major pipelines are further made up of multiple mini-pipelines. The front end stages form the IFU and IDU pipelines. The backend forms the GRU pipeline.The functionality representing the IFU, IDU, and GRU pipelines reside in the units with the same name. The instruction execution specific functionality of the ALU and AGEN pipeline resides in the IEU and the LSU.

In addition to the above mentioned pipelines, there are a few other pipelines that exist in the 74K processor core. The Multiply Divide Unit (MDU) attaches to the ALU pipeline and is an offshoot of the ALU pipeline. The Floating Point Unit (FPU) attaches to the common front end and has a separate pipeline that is described later. There is also a separate post-graduation memory pipeline that comes into existence only for load/store instructions. This pipeline resides entirely in the Load Store Unit.

Figure 2.1 shows the stages of the 74K processor core pipeline. The pipeline stages shown in the figure are described in Table 2.1 and in the following subsections.

Figure 2.1 74K™ Core Pipeline

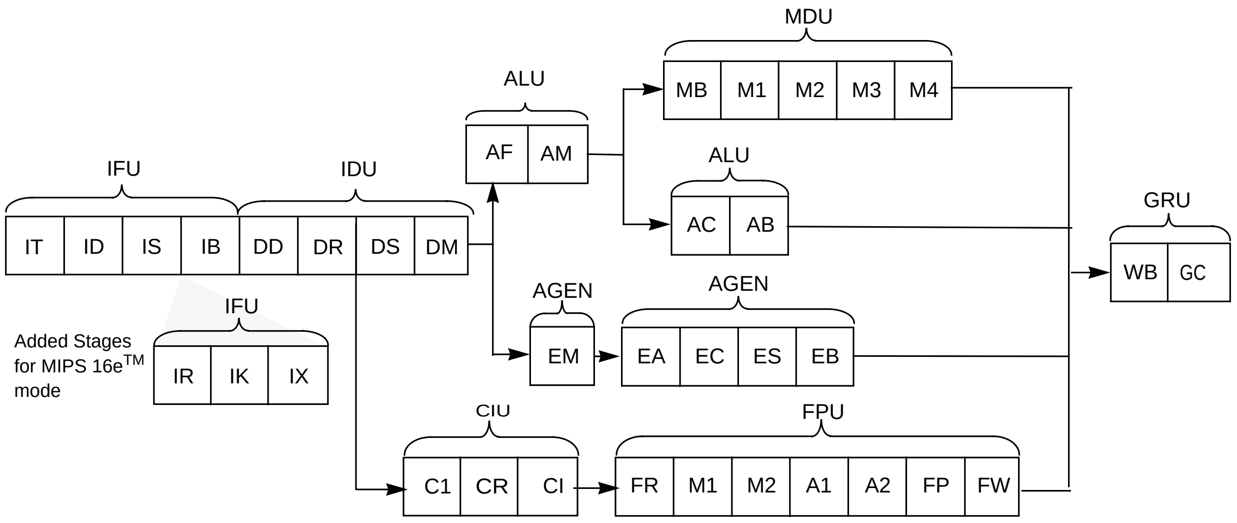


Table 2.1 74K™ Core Pipeline Stages Descriptions

Stage	Description	Stage	Description
IT	I-cache Tag read; ITLB Lookup; BHT Lookup	AF	ALU register File read
ID	I-cache Data read; Tag Compare	AM	ALU operand Mux select
IS	Way Select; Target Calculation Start	AC	ALU Compute
IB	Write Fetch Buffer; Target Calculation Done	AB	ALU Results Bypass
IR	MIPS16 Recode	(MB,M[1-4])	Booth recode, multiply stages 1-4
IK	MIPS16 Branch Decode and validate	EM	Execute operands Mux select
IX	MIPS16 macro expansion	EA	Address Generation; JTLB Access Start;Branch operand select
DD	Instruction Decode and Register Rename (read RMap)	EC	DCache Read; JTLB access continue; Branch Compare
DR	Write RMap; Write DDQs; Issue to CP1/CP2	ES	PTag Compare start; Vhint selects Data; Branch Redirect
DS	Select 1 instruction from DDQ0 and 1 from DDQ1	EB	Load Align; Load data bypass; Ptag validate data select Branch/Jump link data pipe forward.
DM	Finish instruction selection and update DDQ entries.	WB	Completion buffer write; Exceptions determined
C1	Adjust InOrder instruction queue write pointer	GC	Graduation Complete
CR	Read instruction from InOrder instruction queue		
CI	InOrder instruction dispatch to Coprocessor1		

2.1.1 IFU Pipeline

2.1.1.1 IT - Instruction Cache Tag Access

The Instruction Fetch Unit (IFU) determines the address to be accessed in the cycle prior to IT. The I-cache tags, the branch history table (BHT), and the Return Prediction Stack (RPS) are accessed in this stage. The tag data contains some precode bits for each instruction pair indicating instruction type, etc. The I-cache is virtually indexed and the initial lookup proceeds without any physical address bits. In parallel, the ITLB is accessed in this cycle in order to obtain the virtual to physical address translation.

In addition to this Instruction fetch, related Watch exception conditions and EJTAG breakpoint matches are also determined.

The functionality corresponding to this stage resides entirely in the IFU.

2.1.1.2 ID - Instruction Cache Data Access

In this stage, the I-cache arrays are accessed and 128 bits of instruction are read out for each cache way. While the I-cache data is being fetched, the tag outputs are compared against the physical address obtained from the ITLB lookup and the Refill Buffer tags, and one of the 4 associative ways or Refill Buffer entries is determined. The Refill Buffers represent the I-cache misses currently being serviced.

ITag parity error detection is also performed in this stage.

The functionality corresponding to this stage resides entirely in the IFU.

2.1.1.3 IS - Instruction Select

In this cycle, data from the I-cache or Refill Buffer is selected based on results of the tag compare in the previous cycle.

In case the instruction fetched from the I-cache is a Branch type instruction that is predicted taken, the computation of the target address is also started in this cycle.

The functionality corresponding to this stage resides entirely in the IFU.

2.1.1.4 IR - Instruction Recode

This stage comes into existence in the main pipeline only for MIPS16e instructions. MIPS16 instructions are recoded in this stage into MIPS32 equivalent instructions.

The functionality corresponding to this stage resides entirely in the IFU.

2.1.1.5 IK - Instruction

This stage also comes into existence in the main pipeline only for MIPS16e instructions. In this stage the instructions are decoded for control transfer instruction information and validated. The candidate branch/jump instruction is determined out of the 4 possible instructions in this stage.

The functionality corresponding to this stage resides entirely in the IFU.

2.1.1.6 IX - Instruction Macro Expansion

This stage also comes into existence in the main pipeline only for MIPS16e instructions. If a MIPS16e macro instruction is detected it is expanded into multiple MIPS32 instructions in this stage. This stage is bypassed if there are no macro instructions.

The functionality corresponding to this stage resides entirely in the IFU.

2.1.1.7 IB - Instruction Buffer

The MIPS16e and MIPS32 pipelines converge at this stage. Up to 4 instructions from either the IS or IX stages are written into the Fetch Buffers. A maximum of 2 instructions can be read from any Fetch Buffer. The write of the Fetch Buffer can be bypassed in the same cycle.

The functionality corresponding to this stage resides entirely in the IFU.

2.1.2 Instruction Decode Unit Pipeline

2.1.2.1 DD - Dispatch Decode

In this stage the IDU receives up to 2 instructions and decodes them. In parallel with the decode operation the RenameMap is looked up to determine whether the source operands are in the Register File, Completion Buffer/ PipeLine or Unavailable (pending long latency operation).

Lookups are done in this stage only if there are enough Completion Buffer resources available. Each instruction that crosses this stage receives a Completion Buffer ID (CBID) and InstructionID. The CBID determines the location in AGCB or ALCB where the execution unit can write the results. The InstructionID is a sequential ID that uniquely determines the age of the instruction in the pipe between the DD stage and Graduation.

A branch and its delay slot are always presented together by the IFU into the IDU.

This stage is also the synchronization point for several hazard preventing serializing operations such as EHB, MFC0, branch mispredict redirect resolution etc.

The functionality corresponding to this stage resides entirely in the IDU.

2.1.2.2 DR - Dispatch Rename

The instructions from DD stage will arrive into DR stage and any dependencies on older instructions and intra-dependencies among the two incoming instructions are resolved. The RenameMap is updated with the new destination CBID (if applicable) for both instructions.

Theinstructions are written into one of the two out-of-order dispatch queues, DDQ0 and DDQ1. The two queues are 6 entries deep. Each queue is associated with an execution pipe. The queues are divided as follows:

- DDQ1 (AGEN Pipe): Supports Memory Transfer (Load/Store), Control Transfer, and Conditional Move Instructions.
- DDQ0 (ALU Pipe): Supports all other instructions.

Coprocessor 1 instructions are handed off to the Coprocessor Interface Unit (CIU) in this stage. If necessary, some instructions (such as Coprocessor 1 loads and Coprocessor 1 branches, etc.) are written into DDQ1 as well the CIU.

The functionality corresponding to this stage resides entirely in the IDU.

2.1.2.3 DS - Dispatch Select

In this stage the IDU selects 1 instruction out of each DDQ to send down the corresponding execution pipe. It reads a counter associated with each source register to determine the availability for dispatch. Each register (CBID) has its own count down counters that is triggered by its producer that has been dispatched earlier. These counters are used only if the register value is to be bypassed from within the pipeline/Completion Buffer. Older available instructions are given higher priority for selection. An instruction issues out-of-order only when preceding instructions are held up by source register or resource dependencies.

The functionality corresponding to this stage resides entirely in the IDU.

2.1.2.4 DM - DDQ Mux

The result vector from the selection in the previous stage is used to completely read out the selected DDQ entry. The destination CBID of the selected instruction is broadcast to the DDQ and to the RenameMap so that they can start their count down timers. The initial value of the timers is determined in DD stage based on the instruction/execution pipe properties.

In addition for instructions destined for the AGEN Pipe, the access of the Register File and Completion Buffers is started in this stage.

The functionality corresponding to this stage resides largely in the IDU and to a small extent in the IEU.

2.1.3 ALU Pipeline

2.1.3.1 AF - ALU Pipe Register File Read

In this stage the Register File or Completion Buffer read is performed for the source operands of the instructions selected in the DM stage. This functionality requires 2 read ports on the Register File and Completion Buffer - corresponding to two sources for the instruction in ALU pipe. Only 1 instruction can be in the AF stage in the ALU pipe at any time.

The functionality corresponding to this stage resides entirely in the IEU.

2.1.3.2 AM - ALU Pipe Operand Bypass Select Mux

This stage has the final operand bypass muxes for the ALU pipe. The functionality corresponding to this stage resides entirely in the IEU.

2.1.3.3 AC - ALU Compute

The 74K core ALU is pipelined. Some ALU instructions complete the operation and bypass the results in this cycle. These instructions are referred to as single-cycle ops and they include all logical instructions {`AND`, `ANDI, OR, ORI, XOR, XORI, LUI`}, some shift instructions {`SLL sa<=8, SRL 31<=sa<=25`}, and some arithmetic instructions {`ADD rt=0, ADDU rt=0, SLT, SLTI, SLTU, SLTIU, SEH, SEB, ZEH, ZEB`}. In addition add instructions {`ADD, ADDU, ADDI, ADDIU`} complete the operation and bypass results to the ALU pipe in this cycle. Add instructions cannot bypass results to the AGEN pipe in this cycle, but will bypass to the AGEN pipe in the subsequent cycle. All other ALU instructions take 2 cycles to execute and bypass the results to both pipes.This stage corresponds to the first stage of execution for those instructions. Multiply and Divide class of instructions start their execution in a separate MDU pipe and the first stage of that pipe is aligned with this stage.

The functionality corresponding to this stage resides entirely in the IEU.

2.1.3.4 AB - ALU Bypass

This is the second stage of the ALU pipeline. Instructions whose latency exceeds a single cycle, perform their second cycle of computation in this cycle.

All ALU operations can bypass their results from this stage to the muxes in AM, EM and EA stages. All exception information is gathered in this stage to be written into the completion buffer.

The functionality corresponding to this stage resides entirely in the IEU.

2.1.4 MDU Pipeline

2.1.4.1 MB - Multiplier Booth Recode

The MDU performs Booth recoding on the input operand in this stage.

2.1.4.2 M1- M3 Multiplier Array

These stages represent the actual core of the multiplier array.

2.1.4.3 M4 - Multiply Add

This stage is used to perform the add or subtract operation in the case of complex multiply-add or multiply-subtract type of instructions.

2.1.5 AGEN Pipeline

2.1.5.1 EM - Execute Operand Bypass Select Mux

In the presence of shadow set registers, this stage accommodates the final selectors of register data from the different GPRs. In addition, this stage accommodates the bypass muxes for the Load/Store class instructions in the AGEN Pipe.

The functionality corresponding to this stage resides entirely in the IEU

2.1.5.2 EA - Execute and Address Generate

Load/Store Instructions

The effective address calculation for Load/Store class instructions is computed in this cycle. The JTLB access is also started in this cycle for Load/Store type instructions. The JTLB is accessed in parallel using source operands and a fast compare algorithm. This enables the JTLB access to start one cycle earlier than it would be otherwise possible. It makes it possible for the 74K core to access the JTLB inline and avoid a DTLB.

The functionality corresponding to this adder resides entirely in the IEU. The JTLB resides in the MMU.

Control Transfer Instructions

In the case of Control Transfer instructions, the effective redirect address computation is started in this cycle. The bypass muxes for source register operands for this class of instructions are also present in this stage. It is to be noted that there are two sets of bypass muxes in the AGEN Pipe. The first set is in the EM stage and is used for Load/Store instructions. The second set in the EA stage is used for Control Transfer instructions. While it would have been possible for Control transfer instructions to use the bypass muxes from EM and execute the branch in EA, performance simulation has shown that it would significantly reduces overall performance, as it increases the producer/consumer latency between most integer instructions and branches.

The functionality corresponding to Control Transfer Instructions for this stage resides entirely in the IEU.

2.1.5.3 EC - Execute and Cache Access

Load/Store Instructions

The Data Cache is accessed using the effective address calculated in the preceding EA stage. The local buffers in the Load Store Unit (LSU) are also compared against this effective address. The JTLB access is continued in this cycle. The Data cache tags are also accessed in this cycle. The Dtag entry will contain a physical tag and virtual tag.

The functionality corresponding to Dcache and Dtag resides entirely in the LSU. The JTLB resides in the MMU.

Control Transfer Instructions

Conditional Branch and jump instructions are resolved in this cycle. The branch comparison is done and compared against the predicted path. The branch instructions compute the alternate address and redirect the IFU if needed. Since branch instructions can be issued out-of-order the branch execution unit keeps track of the age of the last redirected branch. If the new resolution results in a mispredict the age of the new branch instruction is compared against the age of the last redirected branch that has not yet graduated. Only if the new branch is older than the previous branch the IFU will be redirected again. Register indirect jumps which have been predicted with the Return Stack are also compared against the real register value in this cycle. If there is a mispredict the correct target is sent to the IFU.

The functionality corresponding to Control Transfer Instructions for this stage resides entirely in the IEU.

2.1.5.4 ES - Execute and Cache Second

Load/Store Instructions

The JTLB access is completed in this cycle. The virtual tag is compared against the effective address in this cycle. The tag compare for the various internal LSU buffers is also completed in this cycle. Alignment of the dcache data from all four ways is also done in parallel in this cycle. If a store to load bypass situation is detected by virtue of a load effective address matching against one of the local buffers in the LSU (LSQ or FSB), those buffers are read in this stage.

The functionality corresponding to DCache, DTag, LSQ and other buffers resides entirely in the LSU. The JTLB resides in the MMU.

Control Transfer Instructions

The result of all the branch decision is prioritized and communicated to the IFU in this cycle. In addition to the branch unit redirects there can be redirects from the graduating instructions. These redirects are prioritized over the branch redirects. The final redirect is sent over to the IFU in this cycle. This redirect causes the IFU to kill its current fetch stream and all instructions in the IFU. A new fetch will be started with the newly received target address.

The functionality corresponding to Control Transfer Instructions for this stage resides entirely in the IEU.

2.1.5.5 EB - Execute and Cache Data Bypass

Load/Store Instructions

The Dcache data is selected based on the virtual tag comparison. The final Hit/Miss determination is done by the end of this cycle based on the physical tag comparison with the JTLB output. Alignment of data in case of data return from LSU buffers is done in this cycle. In case of a hit, the selection of data from the appropriate source is also completed in this cycle. The data return to the ALU bypass muxes in EM and EA is done in this cycle.

The functionality corresponding to DCache, DTag, LSQ and other buffers resides entirely in the LSU. The JTLB resides in the MMU.

All exception detection and prioritization within the LSU is completed in this cycle for the AGEN Pipe.

Control Transfer Instructions

In the case of branch and link or jump and link instructions the link register update information is carried forward in the pipe to be written into the completion buffer. This data is eventually written into the Register File when the branch/jump instruction graduates. This data is written into the AGEN completion buffer and it uses the same write port as Load/Store instructions.

The functionality corresponding to this stage for Control Transfer Instructions resides entirely in the IEU.

2.1.6 GRU Pipeline

2.1.6.1 WB - Writeback

The AGEN Pipe writes the AGEN completion buffer (AGCB) in this stage. The ALU pipe and the MDU pipe write into the ALU completion buffer (ALCB) in this stage. All units will also have written their exception information at this stage. The highest priority pipeline exception status will be available for each instruction at this point in the completion buffer structure.

In addition, the oldest 2 entries that have completed execution are identified as candidates for graduation in the subsequent stage.

The buffers and its controls reside across multiple units (LSU, IEU, GRU) depending on the functionality.

2.1.6.2 GC - Graduation Commit

This stage is the final stage of the graduation pipeline. In this stage, the two oldest ready instructions that have been identified in the previous stage are graduated. This implies that the result data corresponding to these two instructions is committed to the architecturally visible GPR, if no flush and redirect due to exceptions or other special cases is required. If a redirect is required, the graduation/exception logic in the GRU will send the appropriate redirect information to the IFU in this stage.

A select class of privileged instructions such as MTC0, TLB operations and CACHE instructions are actually executed at graduation. These instructions are sent to the different units for execution in this stage.

The functionality corresponding to this stage resides entirely in the GRU.

2.2 Programming the 74K Core

For guidelines on programming the 74K core and a better understanding of the impact of the pipeline to the software programmer, refer to the document titled *Programming the 74K Core Family (MD00520*).

2.3 Hazards

In general, the 74K 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 MIPS™ Architecture, hazards (primarily CP0 hazards) were relegated to implementation-dependent cycle-based solutions, primarily based on the SSNOP instruction. The 74K core implements out-of-order dispatch techniques, which is incompatible with this concept of allocating cycles through fixed instruction spacing.

Release 2 defines new instructions which act as explicit barriers that eliminate hazards The new instructions have been added in such a way that they are backward-compatible with existing MIPS processors.

The 74K core family requires that the programmer implement the hazard barrier instructions as defined in Release 2 of the architecture. This does not typically impact an application programmer and is relevant primarily to privileged software. The following sections describe the types of hazards that are addressed. The hazard descriptions in the subsequent sections are here to help the user in identifying hazards in code.

2.3.1 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.

2.3.1.1 Execution Hazards

Execution hazards are those created by the execution of one instruction, and seen by the execution of another instruction. Table 2.2 lists execution hazards.

Table 2.2 Execution Hazards

Producer	→	Consumer	Hazard On
TLBWR, TLBWI	→	TLBP, TLBR	TLB entry
		Load/store using new TLB entry	TLB entry
MTC0	→	Load/store affected by new state	*WatchHi* *WatchLo*
MTC0	→	MFC0	Any CP0 register
MTC0	→	EI/DI	*Status*
MTC0	→	RDHWR $3	*Count*
MTC0	→	Coprocessor instruction execution depends on the new value of $\text{Status}_{\text{CU}}$	$\textit{Status}_{\text{CU}}$
MTC0	→	ERET	*EPC* *DEPC* *ErrorEPC*
MTC0	→	ERET	*Status*
EI, DI	→	Interrupted instruction	$\textit{Status}_{\text{IE}}$
MTC0	→	Interrupted instruction	*Status*
MTC0	→	User-defined instruction	$\textit{Status}_{\text{ERL}}$ $\textit{Status}_{\text{EXL}}$
MTC0	→	Interrupted Instruction	$\textit{Cause}_{\text{IP}}$

Table 2.2 Execution Hazards (Continued)

Producer	→	Consumer	Hazard On
TLBR	→	MFC0	*EntryHi, EntryLo0, EntryLo1, PageMask*
TLBP	→	MFC0	*Index*
MTC0	→	TLBR TLBWI TLBWR	*EntryHi*
MTC0	→	TLBP Load/store affected by new state	*EntryHi*$_{\text{ASID}}$
MTC0	→	TLBWI TLBWR	*EntryLo0* *EntryLo1*
MTC0	→	TLBWI TLBWR	*Index*
MTC0	→	RDPGPR WRPGPR	*SRSCtl*$_{\text{PSS}}$
MTC0	→	Instruction not seeing a Timer Interrupt	Compare update that clears Timer Interrupt
MTC0	→	Instruction affected by change	Any other CP0 register

2.3.1.2 Instruction Hazards

Instruction hazards are those created by the execution of one instruction, and seen by the instruction fetch of another instruction. Table 2.3 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 required for reliable clearing of instruction hazards.

Table 2.3 Instruction Hazards

Producer	→	Consumer	Hazard On
TLBWR, TLBWI	→	Instruction fetch using new TLB entry	TLB entry
MTC0	→	Instruction fetch seeing the new value including: • change to ERL followed by an instruction fetch from the useg segment and • change to ERL or EXL followed by a Watch exception	*Status*
MTC0	→	Instruction fetch seeing the new value	*EntryHi*$_{\text{ASID}}$
MTC0	→	Instruction fetch seeing the new value	*WatchHi* *WatchLo*
MTC0 (write to config7)	→	JR,JALR seeing the new value of IHB of Config7	*IHB bit of Config7*
Instruction stream write via CACHE	→	Instruction fetch seeing the new instruction stream	Cache entries
Instruction stream write via store	→	Instruction fetch seeing the new instruction stream	Cache entries

2.3.2 Instruction Listing

Table 2.4 lists the instructions designed to eliminate hazards. See the document titled *MIPS32® Architecture Reference Manual Volume II: The MIPS32® Instruction Set* (MD00084) for a more detailed description of these instructions.

Table 2.4 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

2.3.2.1 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 MIPS 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.

2.3.3 Eliminating Hazards

In order to eliminate hazards, use one of the instructions listed in Table 2.4 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.

Chapter 3

Floating-Point Unit of the 74Kf™ Core

This chapter describes the MIPS64® Floating-Point Unit (FPU) included in the 74Kf core. This chapter contains the following sections:

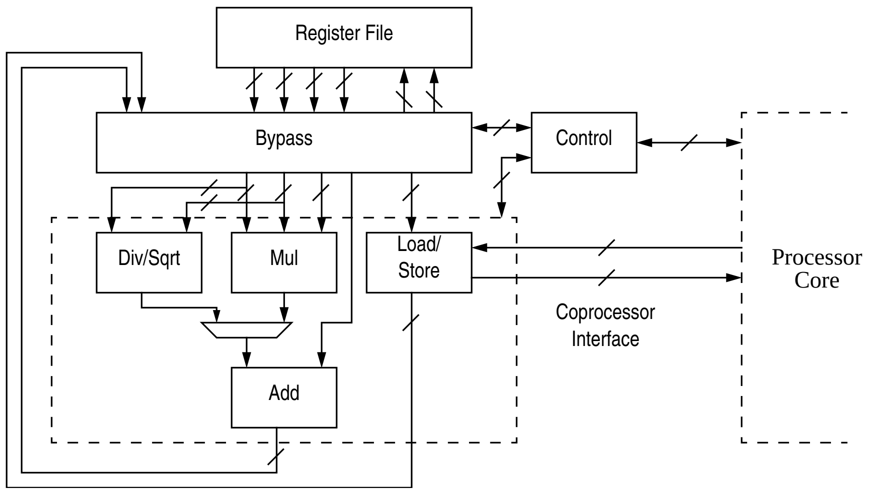
3.1 Features Overview

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 74K core FPU can operate on numbers within a specific range (in general, the IEEE normalized numbers), but it relies on a software handler to operate on numbers not handled by the FPU hardware (in general, the IEEE denormalized numbers). Supported number ranges for different instructions are described later in this chapter. A fast Flush To Zero mode is provided to optimize performance for cases where IEEE denormalized operands and results are not supported by hardware. The fast Flush to Zero mode is enabled through the CP1 *FCSR* register; use of this mode is recommended for best performance.

3.1.1 IEEE Standard 754

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

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

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

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

3.2 Enabling the Floating-Point Coprocessor

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

3.3 Data Formats

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

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

3.3.1 Floating-Point Formats

The FPU provides the following two floating-point formats:

- a 32-bit single-precision floating point (type S, shown in Figure 3.2)
- a 64-bit double-precision floating point (type D, shown in Figure 3.3)

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

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

The single and double floating-point data types are composed of three fields—sign, exponent, fraction—whose sizes are listed in Table 3.1.

Table 3.1 Parameters of Floating-Point Data Types

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

Table 3.1 Parameters of Floating-Point Data Types (Continued)

Parameter	Single	Double
Magnitude of smallest normalized representable number	1.1754943508e-38	2.2250738585e-308

Layouts of these three fields are shown in Figures 3.2 and 3.3 below. The fields are:

- 1-bit sign, *s*
- Biased exponent, *e* = *E* + *bias*
- Binary fraction, $f = .b_1\, b_2..b_{p\text{-}1}$ (the *b*0 bit is *hidden*; it is not recorded)

Figure 3.2 Single-Precision Floating-Point Format (S)

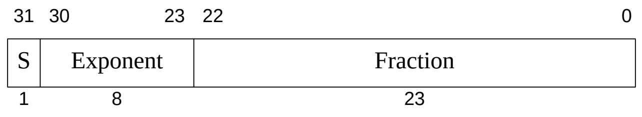
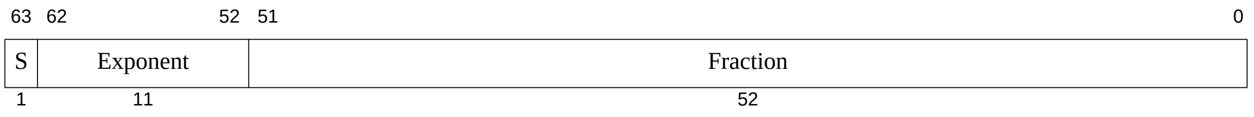


Figure 3.3 Double-Precision Floating-Point Format (D)



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[1]
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)

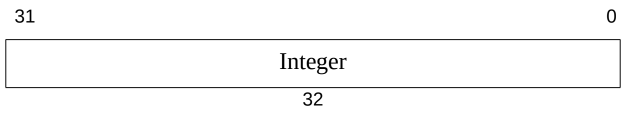
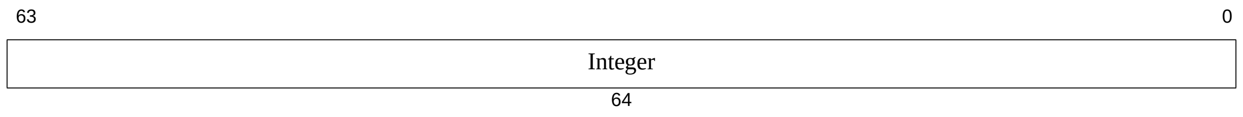


Figure 3.5 Longword Fixed-Point Format (L)



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.

Figures 3.6 and 3.7 show the FPR organization and the way that operand data is stored in them.

Figure 3.6 Single Floating-Point or Word Fixed-Point Operand in an FPR

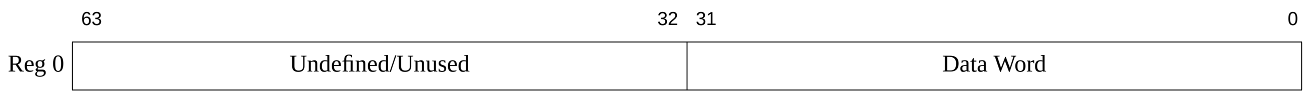
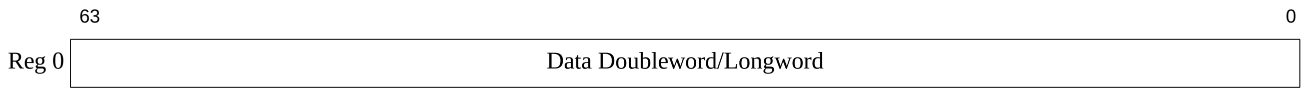


Figure 3.7 Double Floating-Point or Longword Fixed-Point Operand in an FPR



3.4.2 Formats of Values Used in FP Registers

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

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

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

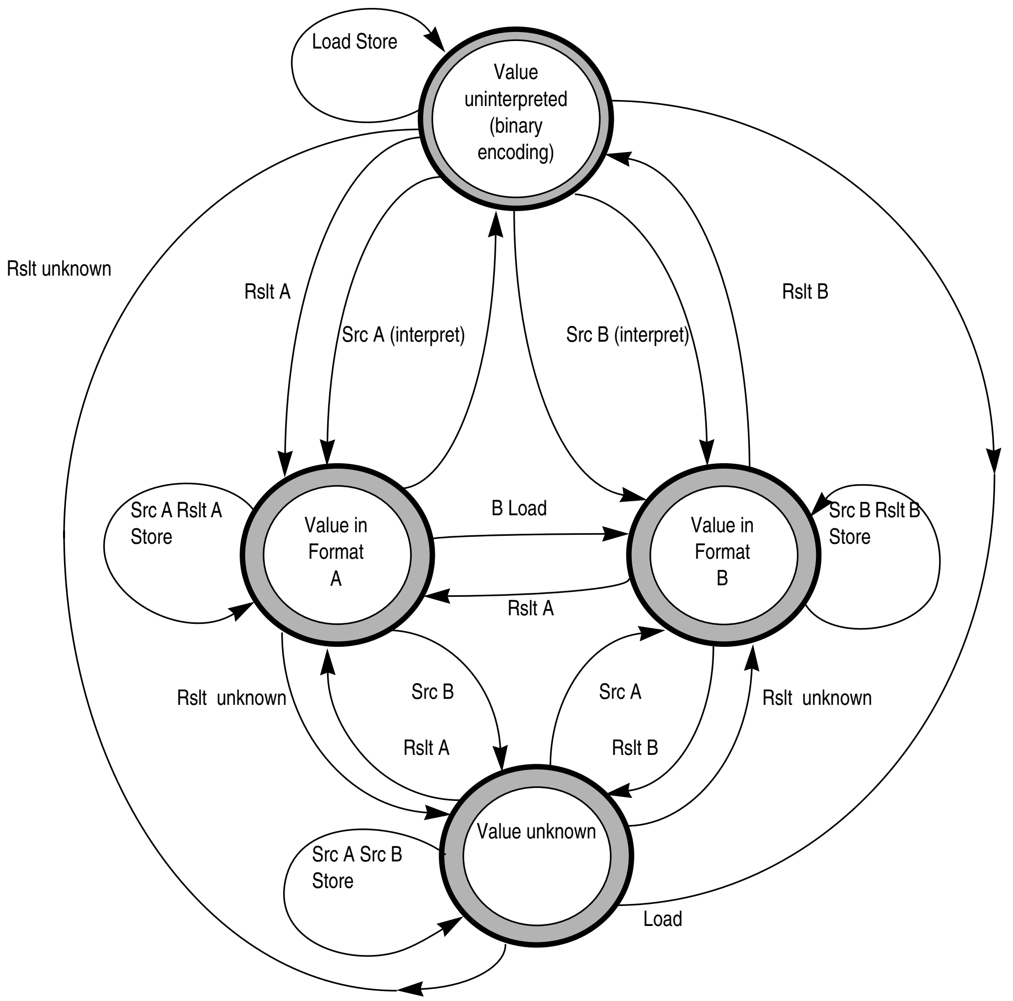
When an FPR with an *uninterpreted* value is used as a source operand by an instruction that requires a value of format *fmt*, the binary contents are interpreted as an encoded value in format *fmt*, and the value in the FPR changes to a value of format *fmt*. The binary contents cannot be reinterpreted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

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

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

The state diagram in Figure 3.8 illustrates the manner in which the formatted value in an FPR is set and changed.

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



A, B: Example formats
Load: Destination of LWC1, LDC1, MTC1 instructions.
Store: Source operand of SWC1, SDC1, 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

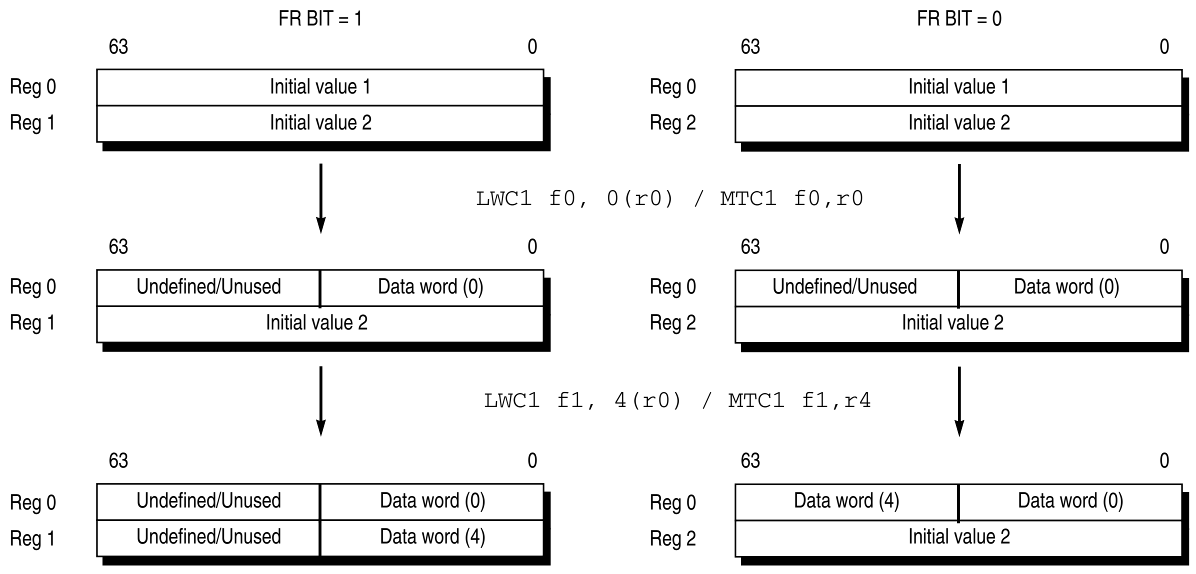
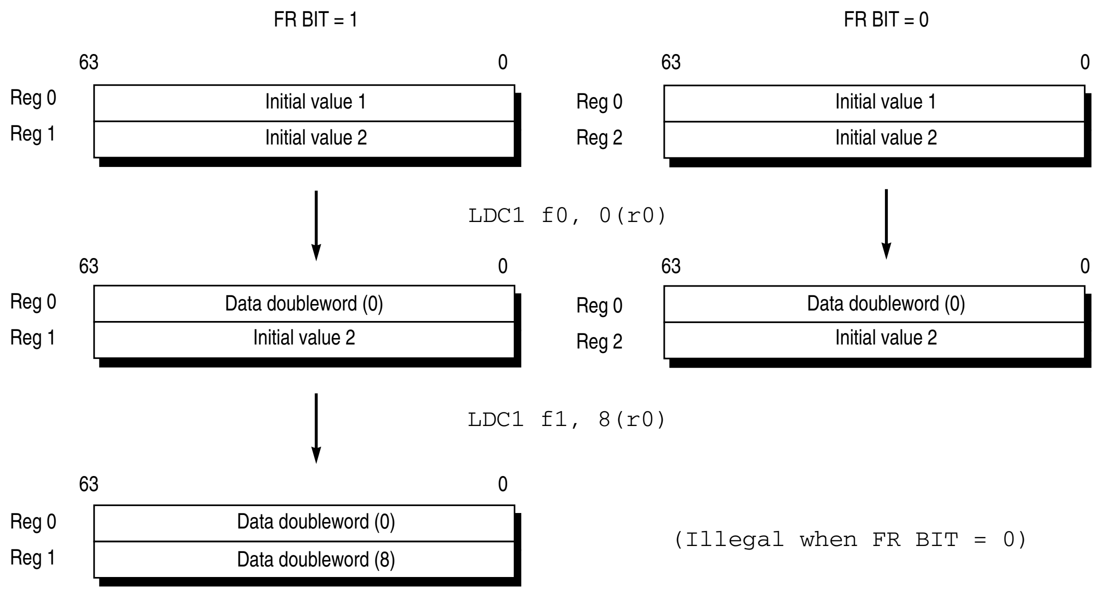


Figure 3.10 FPU Doubleword Load and Move-to Operations



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. Hardware updates of this field are visible by software reads. Software updates of this field are visible by hardware reads. If the reset state of this field is "Undefined," either software or hardware must initialize the value before the first read returns a predictable value. This definition should not be confused with the formal definition of UNDEFINED behavior.	
R	This field is either static or is updated only by hardware. If the Reset State of this field is either "0" or "Preset", hardware initializes this field to zero or to the appropriate state, respectively, on powerup. If the Reset State of this field is "Undefined", hardware updates this field only under those conditions specified in the description of the field.	A field to which the value written by software is ignored by hardware. Software may write any value to this field without affecting hardware behavior. Software reads of this field return the last value updated by hardware. If the Reset State of this field is "Undefined," software reads of this field result in an UNPREDICTABLE value except after a hardware update done under the conditions specified in the description of the field.
0	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. 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

31 25	24	23	22	21	20	19	18	17	16	15 8	7 0
0	FC	0	F64	L	W	3D	PS	D	S	ProcessorID	Revision

Table 3.6 FIR Bit Field Descriptions

Fields				
Name	**Bits**	**Description**	**Read / Write**	**Reset State**
FC	24	Indicates that full convert ranges are implemented: • 0: Full convert ranges not implemented • 1: Full convert ranges implemented 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: • 0: Not a 64-bit FPU • 1: A 64-bit FPU. 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: • 0: Long type not implemented • 1: Long implemented 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: • 0: Word type not implemented • 1: Word implemented 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: • 0: MIPS-3D not implemented • 1: MIPS-3D implemented This bit is always 0 to indicate that MIPS-3D is not implemented.	R	0
PS	18	Indicates that the paired-single (PS) floating-point data type and instructions are implemented: • 0: PS floating-point not implemented • 1: PS floating-point implemented This bit is always 0 to indicate that paired-single floating-point data types are not implemented.	R	0

Table 3.6 FIR Bit Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bits			
D	17	Indicates that the double-precision (D) floating-point data type and instructions are implemented: • 0: D floating-point not implemented • 1: D floating-point implemented This bit is always 1 to indicate that double-precision floating-point data types are implemented.	R	1
S	16	Indicates that the single-precision (S) floating-point data type and instructions are implemented: • 0: S floating-point not implemented • 1: S floating-point implemented This bit is always 1 to indicate that single-precision floating-point data types are implemented.	R	1
Processor ID	15:8	Identifies the floating-point processor. This value matches the corresponding field of the CP0 PRId register.	R	0x97
Revision	7:0	Specifies the revision number of the FPU. This field allows software to distinguish between different revisions 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

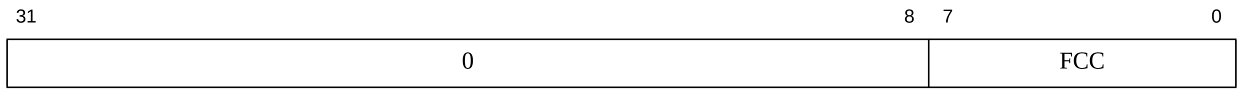


Table 3.7 FCCR Bit Field Descriptions

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

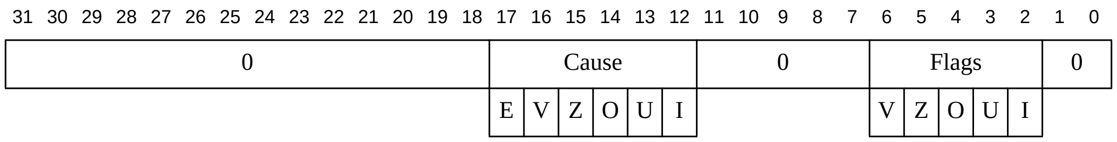


Table 3.8 FEXR Bit Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
Cause	17:12	Cause bits. Refer to the description of this field in 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 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

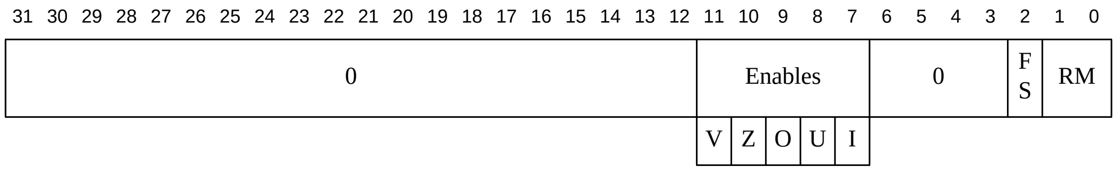


Table 3.9 FENR Bit Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
Enables	11:7	Enable bits. Refer to the description of this field in 3.5.5 "Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)".	R/W	Undefined
FS	2	Flush to Zero bit. Refer to the description of this field in 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 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

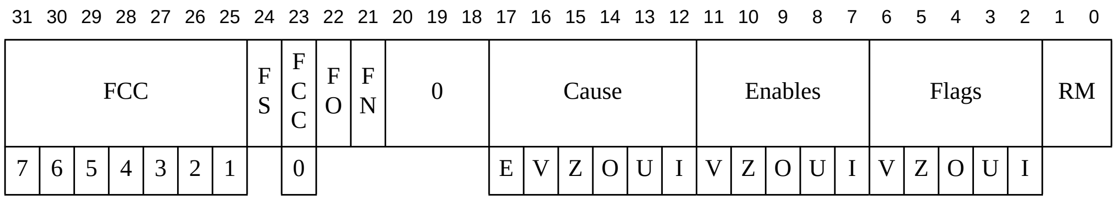


Table 3.10 FCSR Bit Field Descriptions

Fields: Name	Fields: Bit	Description	Read / Write	Reset State
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 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

Table 3.10 FCSR Bit Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bit			
Cause	17:12	Cause bits. These bits indicate the exception conditions that arise during execution of an FPU arithmetic instruction. A bit is set to 1 when the corresponding exception condition arises during the execution of an instruction; otherwise, it is cleared to 0. By reading the registers, the exception condition caused by the preceding FPU arithmetic instruction can be determined. Refer to Table 3.11 for the meaning of each cause bit.	R/W	Undefined
Enables	11:7	Enable bits. These bits control whether or not a trap is taken when an IEEE exception condition occurs for any of the five conditions. The trap occurs when both an enable bit and its corresponding cause bit are set either during an FPU arithmetic operation or by moving a value to the *FCSR* or one of its alternative representations. Note that Cause bit E (CauseE) has no corresponding enable bit; the MIPS architecture defines non-IEEE Unimplemented Operation exceptions as always enabled. Refer to Table 3.11 for the meaning of each enable bit.	R/W	Undefined
Flags	6:2	Flag bits. This field shows any exception conditions that have occurred for completed instructions since the flag was last reset by software. When an FPU arithmetic operation raises an IEEE exception condition that does not result in a Floating-Point Exception (the enable bit was off), the corresponding bit(s) in the Flags field are set, while the others remain unchanged. Arithmetic operations that result in a Floating-Point Exception (the enable bit was on) do not update the Flags field. Hardware never resets this field; software must explicitly reset this field. Refer to Table 3.11 for the meaning of each flag bit.	R/W	Undefined
RM	1:0	Rounding mode. This field indicates the rounding mode used for most floating-point operations (some operations use a specific rounding mode). Refer to Table 3.12 for the encoding of this field.	R/W	Undefined
0	20:18	These bits must be written as zeros; they return zeros on reads.	0	0

Table 3.11 Cause, Enables, and Flags Definitions

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

Table 3.11 Cause, Enables, and Flags Definitions (Continued)

Bit Name	Bit Meaning
I	Inexact

Table 3.12 Rounding Mode Definitions

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

3.5.6 Operation of the FS/FO/FN Bits

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

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

Table 3.13 Handling Denormalized Floating-point Numbers

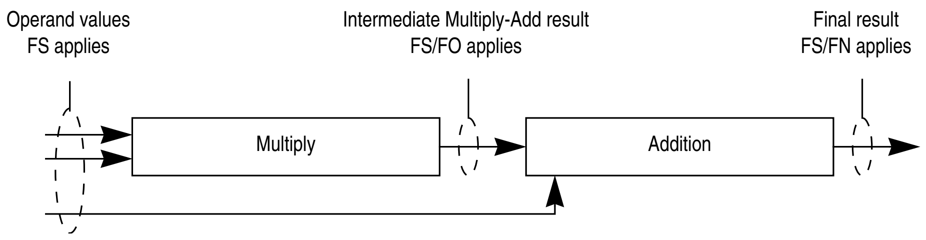
FS and FN bit:	ADD, CEIL, CVT, DIV, FLOOR, MADD, MSUB, MUL, NMADD, NMSUB, RECIP, ROUND, RSQRT, SQRT, TRUNC, SUB, ABS, C.cond, and NEG[1]
FO bit:	MADD, MSUB, NMADD, and NMSUB

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 3.14 lists the flushing behavior for tiny results..

Table 3.14 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 3.15.

Table 3.15 Handling of Denormalized Operand Values and Tiny Results Based on FS Bit Setting

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

3.5.6.2 Flush Override Bit

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

Handling of tiny intermediate results depends on the FO and FS bits as shown in Table 3.16.

Table 3.16 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.

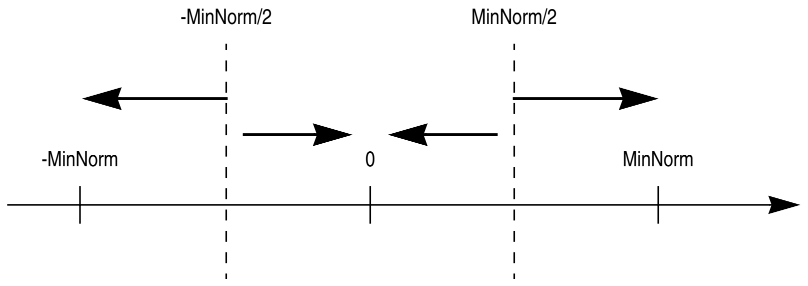
Table 3.16 Handling of Tiny Intermediate Result Based on the FO and FS Bit Settings (Continued)

FO Bit	FS Bit	Handling of Tiny Result Values
0	1	The intermediate result is forced to the value that would have been delivered for an untrapped underflow (see Table 3.34) instead of causing an Unimplemented Operation exception.
1	Don’t care	The intermediate result is kept in an internal format, which can be perceived as having the usual mantissa precision but with unlimited exponent precision and without forcing to a specific value or taking an exception.

3.5.6.3 Flush to Nearest

When the Flush to Nearest (FN) bit is set and the rounding mode is Round to Nearest (RN), a tiny final result is flushed to zero or MinNorm. If a tiny number is strictly below MinNorm/2, the result is flushed to zero; otherwise, it is flushed to MinNorm (see Figure 3.17). The flushed result has the same sign as the result prior to flushing. Note that the FN bit takes precedence over the FS bit.

Figure 3.17 Flushing to Nearest when Rounding Mode is Round to Nearest



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

Handling of tiny final results depends on the FN and FS bits as shown in Table 3.17.

Table 3.17 Handling of Tiny Final Result Based on FN and FS Bit Settings

FN Bit	FS Bit	Handling of Tiny Result Values
0	0	An Unimplemented Operation exception is taken.
0	1	Final result is forced to the value that would have been delivered for an untrapped underflow (see Table 3.34) rather than causing an Unimplemented Operation exception.
1	Don’t care	Final result is rounded to either zero or 2^{E_min} (MinNorm), whichever is closest when in Round to Nearest (RN) rounding mode. For other rounding modes, a final result is given as if FS was set to 1.

3.5.6.4 Recommended FS/FO/FN Settings

Table 3.18 summarizes the recommended FS/FO/FN settings.

Table 3.18 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 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 13, “74K™ Processor Core Instructions” on page 303, 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 3.19 lists the supported transfer operations.

Table 3.19 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 3.20 through 3.22 list the FPU data transfer instructions.

Table 3.20 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 3.21 FPU Loads and Stores Using Register+Register Address Mode

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

Table 3.22 FPU Move To and From Instructions

Mnemonic	Instruction
CFC1	Move Control Word From Floating Point
CTC1	Move Control Word To Floating Point

Table 3.22 FPU Move To and From Instructions (Continued)

Mnemonic	Instruction
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 3.23 lists the FPU IEEE compliant arithmetic operations.

Table 3.23 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 3.24 lists the FPU-approximate arithmetic operations.

Table 3.24 FPU-Approximate Arithmetic Operations

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

Four compound-operation instructions perform variations of multiply-accumulate operations; that is, multiply two operands, accumulate the result to a third operand, and produce a result. These instructions are listed in Table 3.25. 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 3.25 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 3.26 and Table 3.27 list the FPU conversion instructions according to their rounding mode.

Table 3.26 FPU Conversion Operations Using the FCSR Rounding Mode

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

Table 3.27 FPU Conversion Operations Using a Directed Rounding Mode

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

3.6.4 Formatted Operand-Value Move Instructions

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

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

Conditional move instructions operate in a way that might be unexpected. They always force the value in the destination register to become a value of the format specified in the instruction. If the destination register does not contain an operand of the specified format before the conditional move is executed, the contents become undefined. (For more information, see the individual descriptions of the conditional move instructions in the *MIPS32 Architecture Reference Manual, Volume II*.)

Table 3.28 through Table 3.30 list the formatted operand-value move instructions.

Table 3.28 FPU Formatted Operand Move Instruction

Mnemonic	Instruction
MOV.fmt	Floating-Point Move

Table 3.29 FPU Conditional Move on True/False Instructions

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

Table 3.30 FPU Conditional Move on Zero/Non-Zero Instructions

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

3.6.5 Conditional Branch Instructions

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

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

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

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

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

Table 3.31 lists the conditional branch (branch and branch likely) FPU instructions; Table 3.32 lists the deprecated conditional branch likely instructions.

Table 3.31 FPU Conditional Branch Instructions

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

Table 3.32 Deprecated FPU Conditional Branch Likely Instructions

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

3.6.6 Miscellaneous Instructions

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

Table 3.33 lists these conditional move instructions.

Table 3.33 CPU Conditional Move on FPU True/False Instructions

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

3.7 Exceptions

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

3.7.1 Precise Exception Mode

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

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

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

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

If a floating-point operation sets only non-enabled cause bits, no trap occurs and the default result defined by IEEE Standard 754 is stored (see Table 3.34). When a floating-point operation does not trap, the program can monitor the exception conditions by reading the Cause field.

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

3.7.2 Exception Conditions

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

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

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

At the program's direction, an IEEE exception condition can either cause a trap or not cause a trap. IEEE Standard 754 specifies the result to be delivered in case no trap is taken. The FPU supplies these results whenever the exception condition does not result in a trap. The default action taken depends on the type of exception condition and, in the case of the Overflow and Underflow, the current rounding mode. Table 3.34 summarizes the default results.

Table 3.34 Result for Exceptions Not Trapped

Bit	Description	Default Action
V	Invalid Operation	Supplies a quiet NaN.
Z	Divide by zero	Supplies a properly signed infinity.

Table 3.34 Result for Exceptions Not Trapped (Continued)

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

3.7.2.1 Invalid Operation Exception

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

The following operations are invalid:

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

3.7.2.2 Division By Zero Exception

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

3.7.2.3 Underflow Exception

Two related events contribute to underflow:

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

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

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

3.7.2.4 Overflow Exception

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

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

3.7.2.5 Inexact Exception

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

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

3.7.2.6 Unimplemented Operation Exception

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

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

There is no enable bit for this condition; it always causes a trap (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.

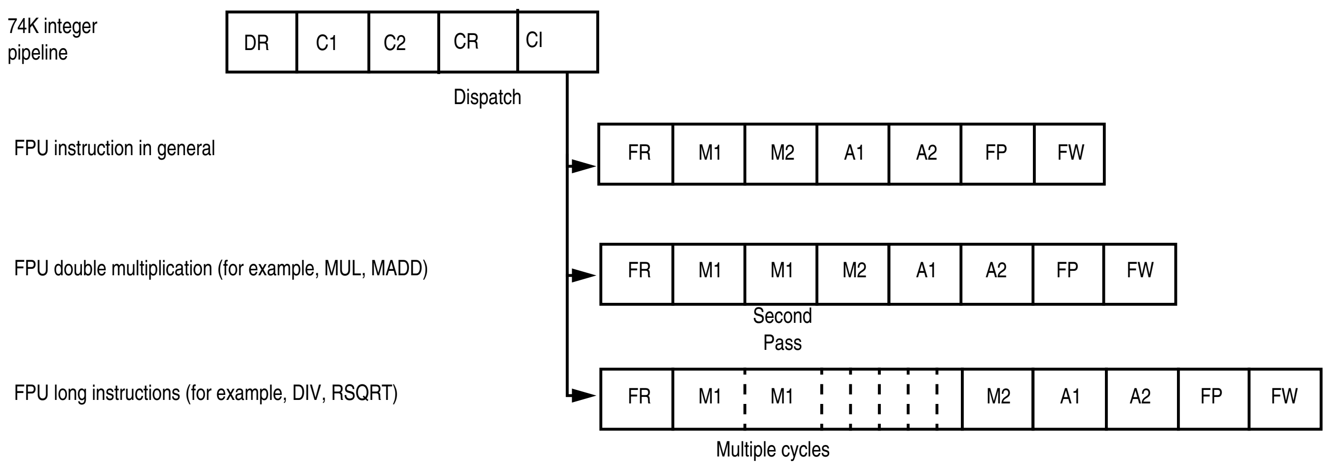
3.8.1 Pipeline Overview

The FPU has a seven stage pipeline to which the integer pipeline dispatches instructions. The FPU supports limited dual-issue. It can accept one “to/from” instruction and one arithmetic instruction every cycle. A “to/from” instruction is a data transfer instruction and covers the following instructions: MFC1, MFHC1, MTC1, MTHC1, CFC1, CTC1, LWC1, LWXC1, LUXC1, LDC1, LDXC1, SWC1, SWXC1, SDC1 and SDXC1. The arithmetic group refers to all other floating point instructions.

The FPU pipeline runs in parallel with the 74K integer pipeline. The FPU can be built to run at different frequencies compared to the integer core. The supported ratios of integer core clock to FPU clk are: 2:1, 3:2 and 1:1.

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


74K integer pipeline
DR
C1
C2
CR
CI
Dispatch
FPU instruction in general
FR
M1
M2
A1
A2
FP
FW
FPU double multiplication (for example, MUL, MADD)
FR
M1
M1
M2
A1
A2
FP
FW
Second Pass
FPU long instructions (for example, DIV, RSQRT)
FR
M1
M1
M2
A1
A2
FP
FW
Multiple cycles

3.8.1.1 DR Stage - Dispatch Rename

The DR stage is described in detail in Chapter 2, “Pipeline of the 74K™ Core” on page 37. This stage is common to both the integer and floating point pipeline. The two pipelines fork off separately after this stage.

3.8.1.2 C1 - Coprocessor Interface Unit Stage 1

In this stage, the Coprocessor Interface Unit (CIU) receives two instructions from the IDU. It does some preliminary decoding and determines if there is space available in its internal queues for the received instructions.

3.8.1.3 CR Stage - Coprocessor Interface Unit Queue Read

The CIU internal instruction queues are read in this stage and up to two instructions are selected for dispatch to the FPU.

3.8.1.4 CI Stage - Coprocessor 1 Interface

This pipeline stage represents the interface stage, where instructions are sent to the FPU

3.8.1.5 FR Stage - Decode, Register Read, and Unpack

This pipeline is the second interface stage, where the instruction start is sent to the FPU.

The FR stage also has the following additional functionality:

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

3.8.1.6 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.
- If an exception can be predicted, then it is sent out in this pipeline stage.

3.8.1.7 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.

3.8.1.8 A1 Stage - Addition First Step

This stage performs the first step of the addition.

3.8.1.9 A2 Stage - Addition Second and Final Step

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

3.8.1.10 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.

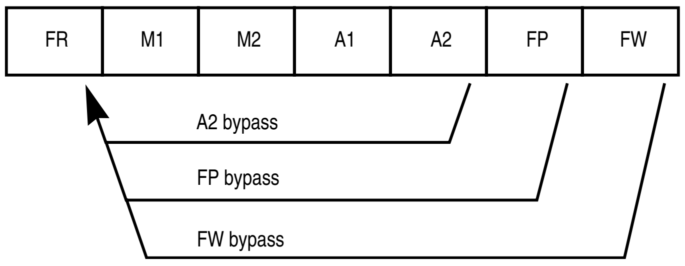
3.8.1.11 FW Stage - Register Write

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 3.35 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 3.35 74Kf 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

Table 3.35 74Kf Core FPU Latency and Repeat Rate (Continued)

Opcode[1]	Latency (cycles)	Repeat Rate (cycles)
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 74K core includes support for the MIPS DSP Application-Specific Extension (ASE) Revision 2 that provides enhanced performance capabilities for a wide range of signal-processing applications, with computational support for fractional data types, SIMD, saturation, and other operations that are commonly used in these applications.

Refer to *MIPS32® Architecture For Programmers, Volume IV-e* for a general description of the DSP ASE and detailed descriptions of the DSP instructions. Additional programming information is contained in *Programming the MIPS 74K Family Cores for DSP* and in the DSP chapter of *Programming the MIPS32® 74K™ Core Family.*

4.1 Additional Register State for the DSP ASE

The DSP ASE defines three additional accumulator registers and one additional control/status register, as described below. 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 operations.

4.1.1 HI-LO Registers

The DSP ASE includes three HI/LO accumulator register pairs (ac1, ac2, and ac3) in addition to the HI/LO register pair (ac0) in the standard MIPS32 architecture. These registers improve the parallelization of independent accumulation routines—for example, filter operations, convolutions, etc. DSP instructions that target the accumulators use two instruction bits to specify the destination accumulator, with the zero value referring to the original accumulator.

4.1.2 DSP Control Register

The *DSPControl* register contains control and status information used by DSP instructions. Figure 4.1 illustrates the bits in this register, and Table 4.1 describes their usage.

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

31 28	27 24	23 16	15	14	13	12 7	6	5 0
0	ccond	ouflag	0	EFI	c	scount	0	pos

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

Fields					
Name	**Bits**	**Description**	**Read / Write**	**Reset State**	**Compliance**
0	31:28	Reserved. Used in the MIPS64 architecture but not used in the MIPS32 architecture. Must be written as zero; returns zero on read.	0	0	Required
ccond	27:24	Condition code bits set by compare instructions. The compare instruction sets the right-most bits as required by the number of elements in the vector compare. 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. 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 instruction fails. These instructions fail when there are insufficient bits to extract, that is, when the value of pos in *DSPControl* is less than the value of size specified in the instruction. This bit is not sticky, so each invocation of one of the four instructions will reset the bit depending on whether or not the instruction failed.	R/W	0	Required
c	13	Carry bit. This bit is set and used by special add instructions that implement a 64-bit add across two GPRs. The ADDSC instruction sets the bit and the ADDWC instruction 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. 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). 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, as described in Table 4.2. These bits are sticky and can be reset only by an explicit write to these bits in the register (using the WRDSP instruction).

Table 4.2 DSPControl ouflag Bits

Bit Number	Description
16	This bit is set when the destination is accumulator (*HI-LO* pair) zero, and an operation overflow or underflow occurs. These instructions are: DPAQ_S, DPAQ_SA, DPSQ_S, DPSQ_SA, DPAQX_S, DPAQX_SA, DPSQX_S, DPSQX_SA, MAQ_S, MAQ_SA and MULSAQ_S.
17	Same instructions as above, when the destination is accumulator (*HI-LO* pair) one.
18	Same instructions as above, when the destination is accumulator (*HI-LO* pair) two.
19	Same instructions as above, when the destination is accumulator (*HI-LO* pair) three.
20	Instructions that set this bit on an overflow/underflow: ABSQ_S, ADDQ, ADDQ_S, ADDU, ADDU_S, ADDWC, SUBQ, SUBQ_S, SUBU and SUBU_S.
21	Instructions that set this bit on an overflow/underflow: MUL, MUL_S, MULEQ_S, MULEU_S, MULQ_RS, and MULQ_S.
22	Instructions that set this bit on an overflow/underflow: PRECRQ_RS, SHLL, SHLL_S, SHLLV, and SHLLV_S.
23	Instructions that set this bit on an overflow/underflow: EXTR, EXTR_S, EXTR_RS, EXTRV, and EXTRV_RS.

4.2 Software Detection of the DSP ASE Revision 2

The presence of the MIPS DSP ASE in the 74K core is indicated by two static bits in the *Config3* register: the *DSPP* (*DSP Present*) bit indicates the presence of the DSP ASE, and the DSP2P (DSP Rev2 Present) bit indicates the presence of the MIPS DSP ASE Rev2. Because all members of the 74K core family support both ASEs, these bits are always set to 1.

The *MX* (*DSP ASE Enable*) read/write bit in the CP0 *Status* register must be set to enable access to the extra instructions defined by the DSP ASE, as well as to the MTLO/HI, MFLO/HI instructions that access accumulators ac1, ac2, and ac3. Executing a DSP ASE instruction or the MTLO/HI, MFLO/HI instructions with this bit set to zero causes a DSP State Disabled Exception (exception code 26 in the CP0 *Cause* register). This exception can be used by system software to do lazy context-switching.

Chapter 5

Memory Management of the 74K™ Core

The 74K 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:

- Section 5.2 "Modes of Operation"
- Section 5.3 "Translation Lookaside Buffer"
- Section 5.4 "Virtual-to-Physical Address Translation"
- Section 5.5 "Fixed Mapping MMU"

5.1 Introduction

The MMU in a 74K processor core translates a virtual address to a physical address before the request is sent to the cache controllers for tag comparison or to the bus interface unit for an external memory reference. Virtual-to-physical address translation is especially useful for operating systems that must manage physical memory to accommodate multiple tasks active in the same memory, and possibly in the same virtual address space (though, of course, in different locations in physical memory). The MMU also enforces the protection of memory areas and defines the cache protocols.

By default, the MMU is TLB-based. The TLB consists of two address-translation buffers: a dual-ported 16/32/48/64 dual-entry fully associative Joint TLB (JTLB) and a 4-entry instruction micro TLB (ITLB). When an instruction address is translated, the ITLB is accessed first, and if the translation is not found, the JTLB is accessed. If there is a miss in the JTLB, an exception is taken. When a data reference is translated, the JTLB is accessed directly. If the address is not present in the JTLB, an exception is taken. The JTLB is dual-ported to prevent contention between instruction and data accesses.

Optionally, the MMU can implement a Fixed Mapping (FM) mechanism, based on a simple algorithm that translates virtual addresses into physical addresses. These translations are different for various regions of the virtual address space.

Figure 5.1 shows how the MMU with TLB interacts with cache accesses, and Figure 5.2 shows the equivalent for the FM MMU.

Figure 5.1 Address Translation For Cache Access with TLB MMU

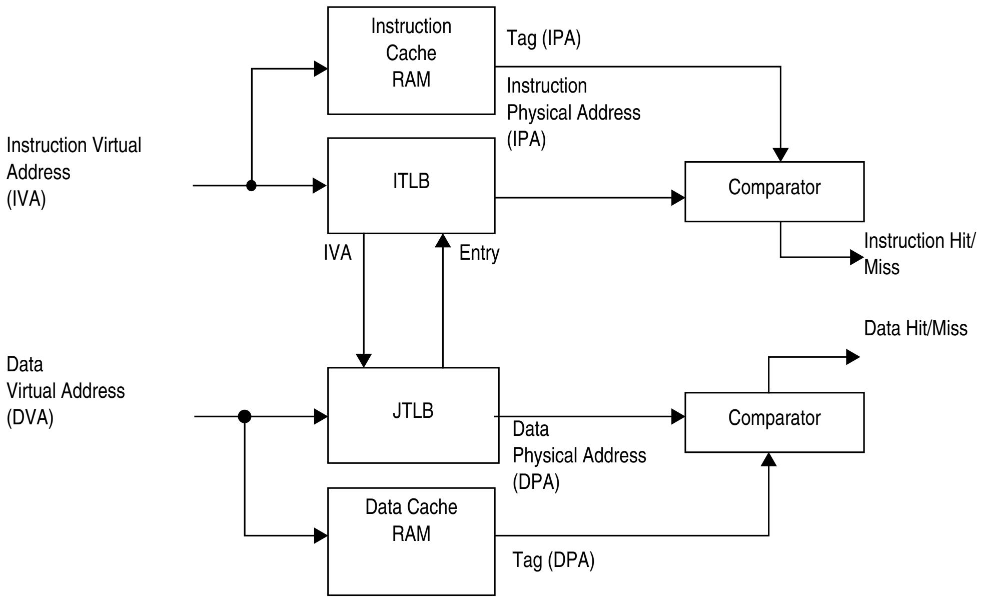
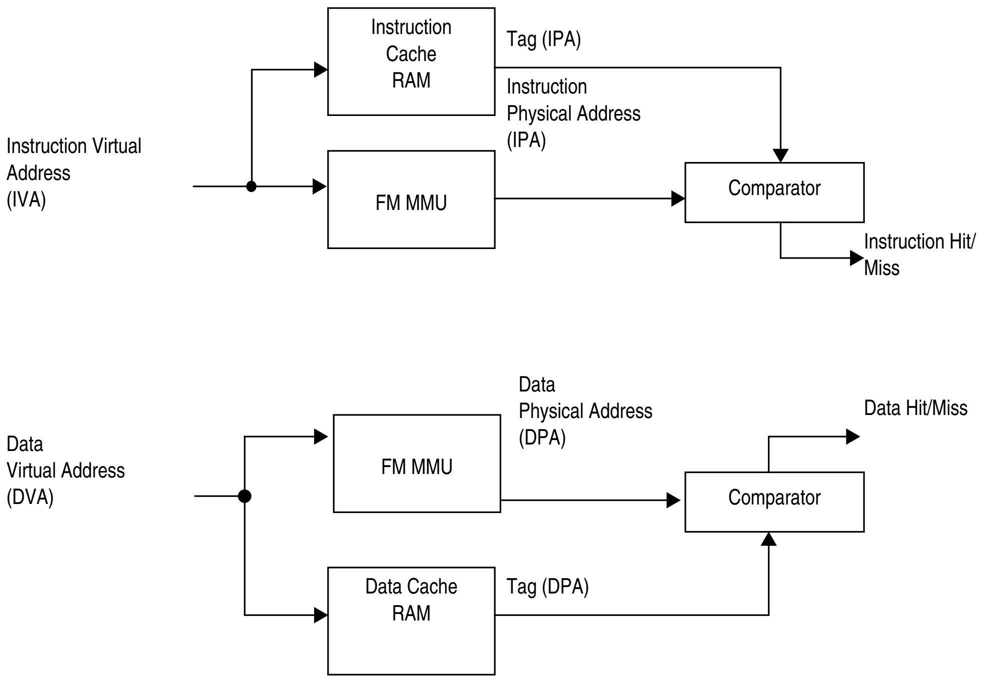


Figure 5.2 Address Translation For Cache Access with FM MMU


Instruction Cache RAM
Tag (IPA)
Instruction Physical Address (IPA)
Instruction Virtual Address (IVA)
FM MMU
Comparator
Instruction Hit/ Miss
Data Physical Address (DPA)
Data Hit/Miss
Data Virtual Address (DVA)
FM MMU
Comparator
Data Cache RAM
Tag (DPA)

5.2 Modes of Operation

The MMU’s virtual-to-physical address translation is determined by the mode in which the processor is operating. A 74K processor core operates in one of four modes:

- User mode
- Supervisor mode
- Kernel mode
- Debug mode

User mode is most often used for application programs. Supervisor mode is 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 usually occurs within a software development tool.

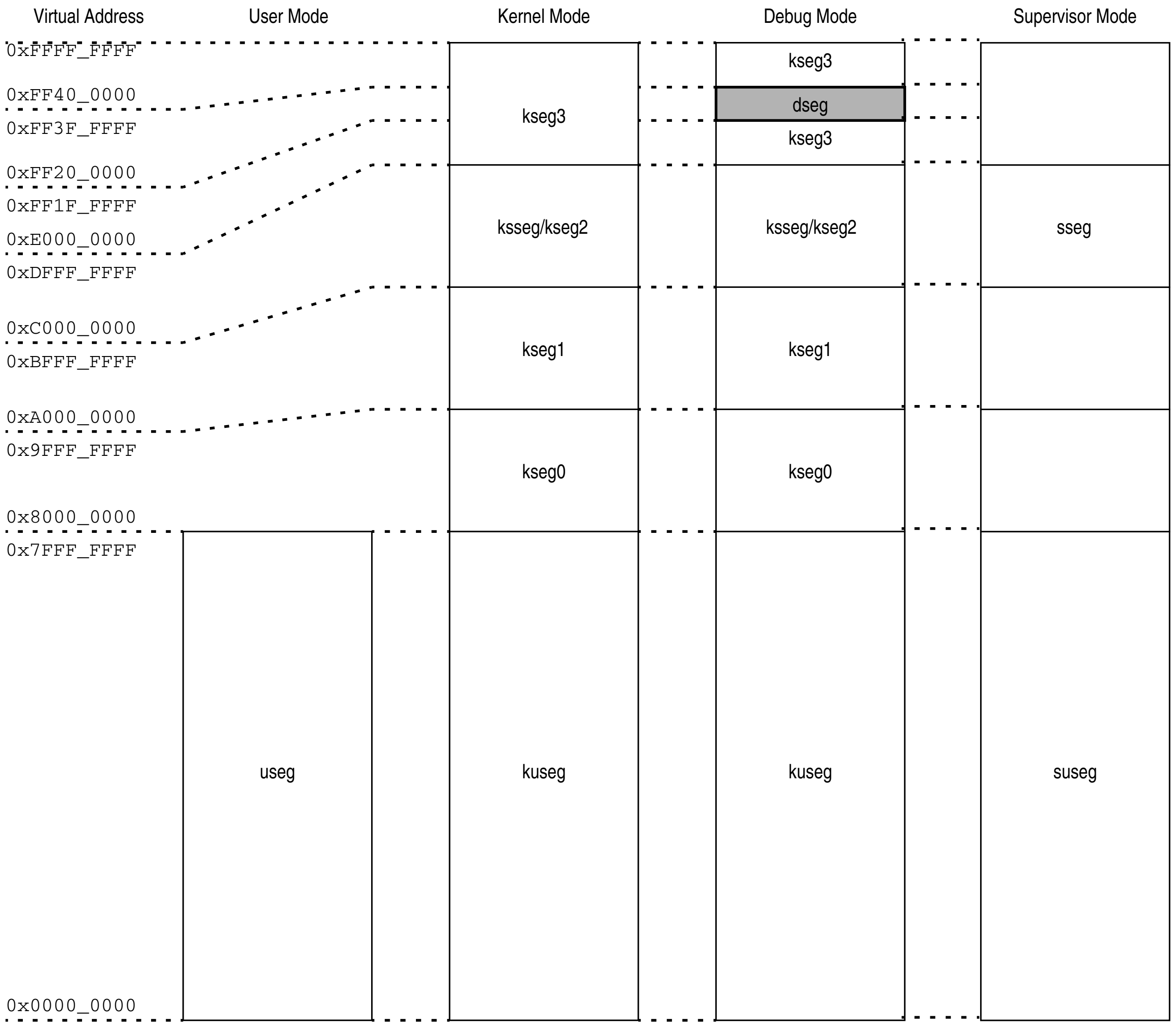
5.2.1 Virtual Memory Segments

The MIPS32 architecture supports a 4 GByte virtual address space that is partitioned into a number of segments, each characterized by a set of attributes defined by hardware and software. 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 each of the four modes.

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. In Kernel mode, software has access to the entire address space, as well as all CP0 registers.

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 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. Access to dseg 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 74K™ Processor Core Virtual Memory Map



Segments can be mapped or unmapped, as described in the following subsections.

5.2.1.1 Unmapped Segments

An unmapped segment does not use the TLB or the FM to translate 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 *Config* register (see Section 7.2.21 "Config (CP0 Register 16, Select 0): Legacy Configuration Register").

5.2.1.2 Mapped Segments

A mapped segment uses the TLB or the FM to translate from virtual to physical addresses.

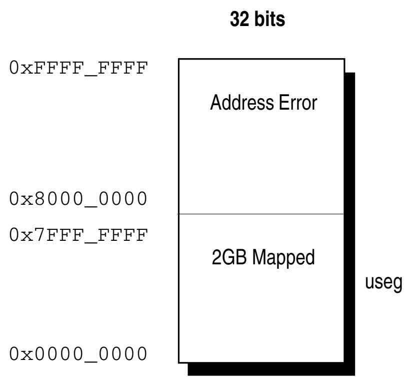
For the core with TLB, the translation of mapped segments is handled on a per-page basis. Included in this translation is information defining whether the page is cacheable or not, and the protection attributes that apply to the page.

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 K23 and KU fields of the CP0 register *Config* (see Section 7.2.21 “Config (CP0 Register 16, Select 0): Legacy Configuration Register”). 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* = 0b10
- *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 A(31) = 0	0	0	0b10	useg	0x0000_0000 --> 0x7FFF_FFFF	2 GByte (2^{31} bytes)

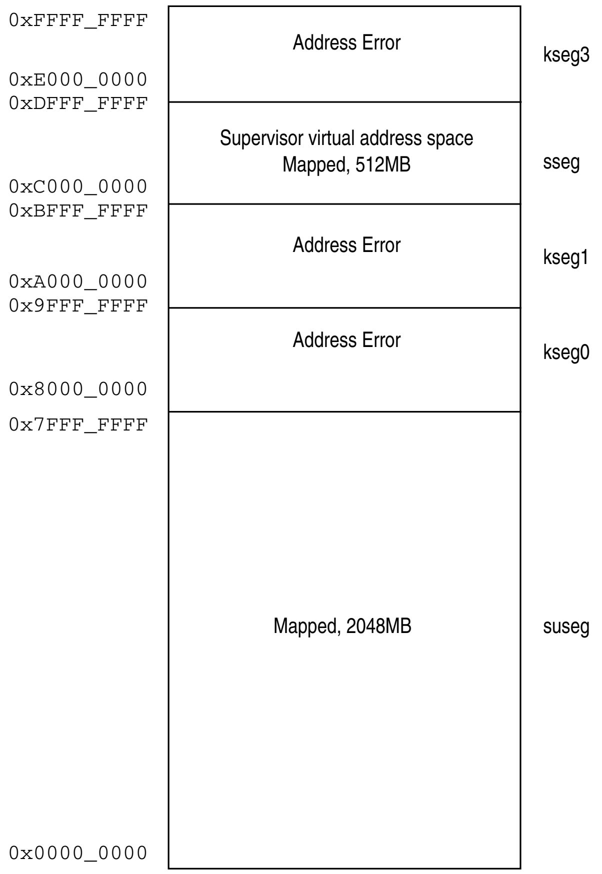
All valid user mode virtual addresses have their most-significant bit cleared to 0, indicating that user mode can only access the lower half of the virtual memory map. All attempts to reference an address with the most-significant bit set while in user mode causes an address error exception.

All references to useg are mapped 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 uniform virtual address spaces are available: a 2 GByte (2^{31} bytes) virtual address space called the supervisor user segment (suseg), and a 512 MByte virtual address space called the supervisor segment (sseg). The supervisor-mode virtual address space is shown in Figure 5.5.

Figure 5.5 Supervisor Mode Virtual Address Space



The supervisor user segment begins 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 in Supervisor mode cause an address error exception.

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

- *UM* = 0 and *SM*= 1
- *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 EXL	ERL	UM	SM	Segment Name	Address Range	Segment Size
32-bit A(31) = 0	0	0	0	1	suseg	0x0000_0000 --> 0x7FFF_FFFF	2 GByte (2^{31} bytes)
32-bit A(31:29) = 110_2	0	0	0	1	sseg	0xC000_0000 -> 0xDFFF_FFFF	512MB (2^{29} bytes)

The system maps all references to s*useg and sseg* 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 s*useg* and *sseg* 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* = 0b00
- *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.

In Kernel mode, a program has access to the entire virtual address space. 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. The characteristics of kernel-mode segments are listed in Table 5.3.

The core enters Kernel mode both at reset and when an exception is recognized.

Figure 5.6 Kernel Mode Virtual Address Space

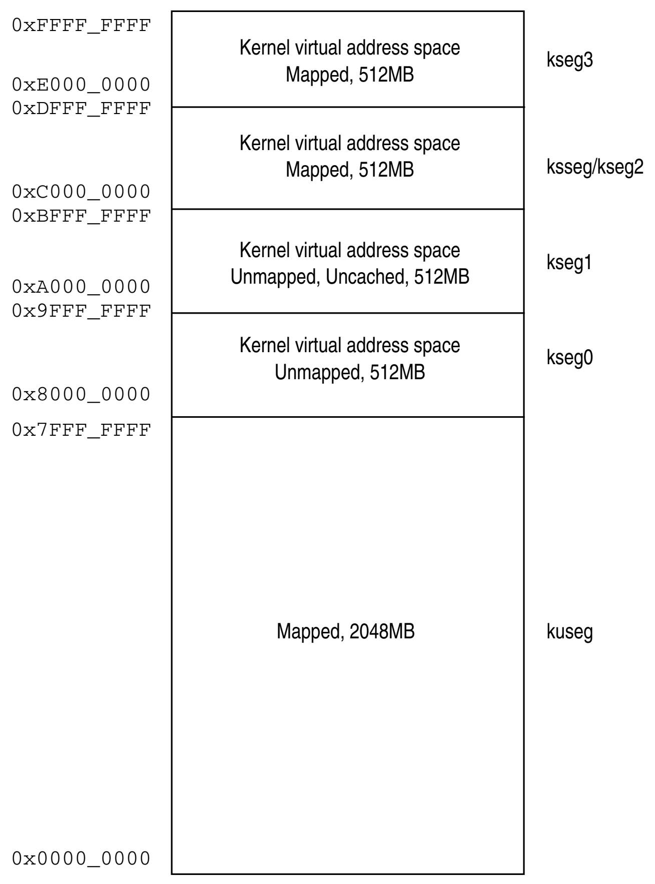


Table 5.3 Kernel Mode Segments

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, 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.

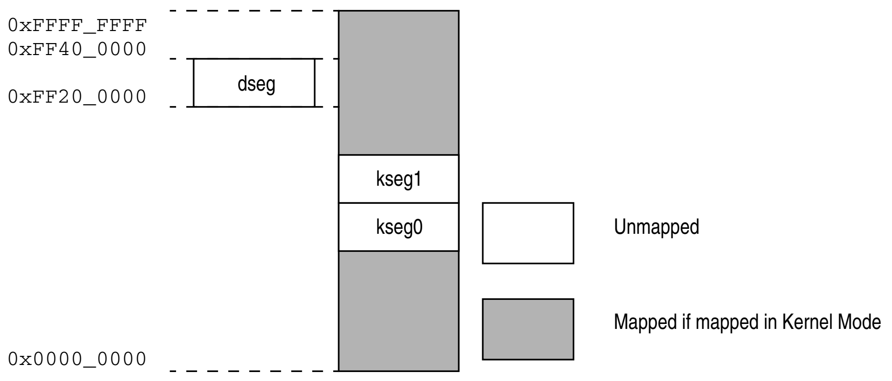
5.2.4.5 Kernel Mode, Kernel Space 3 (kseg3)

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

Except for kseg3, debug-mode address space is identical to kernel-mode address space with respect to mapped and unmapped areas. In kseg3, a debug segment (dseg) coexists 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



dseg is subdivided into the dmseg segment at 0xFF20_0000 to 0xFF2F_FFFF, which is used when the debug 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 of the segments are shown in Table 5.4.

Accesses to memory that would normally cause an exception in 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 in debug mode, which allows the debug handler to be executed from uncached, unmapped memory.

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

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 Debug Mode, Register (drseg)

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

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 drseg. For more information about the *DCR*, refer to Chapter 11, “EJTAG Debug Support in the 74K™ Core” on page 223.

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 Debug Mode, Memory (dmseg)

The conditions for CPU accesses to the dmseg address range (0xFF20_0000 to 0xFF2F_FFFF) are shown in Table 5.6.

Table 5.6 CPU Access to dmseg

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	dmseg
Fetch	0	Don’t care	See comments below
Load / Store	0	0	See comments below

An attempt to access dmseg when the ProbEn bit in the DCR register is 0 should not happen, because 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 occur, the reference hangs until it is satisfied by the probe. The probe must not assume that there will never be a reference to dmseg when 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 TLB memory-management scheme used in the 74Kc processor core includes two address-translation units:

- 16, 32, 48, or 64 dual-entry fully associative Joint TLB (JTLB)
- 4-entry fully associative Instruction micro TLB (ITLB)

5.3.1 Joint TLB

The 74K core implements a 16-64 dual-entry, fully associative Joint TLB that 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 ranges shown in the figure serve to clarify which address bits are (or may be) affected during the translation process.

Figure 5.8 JTLB Entry (Tag and Data)

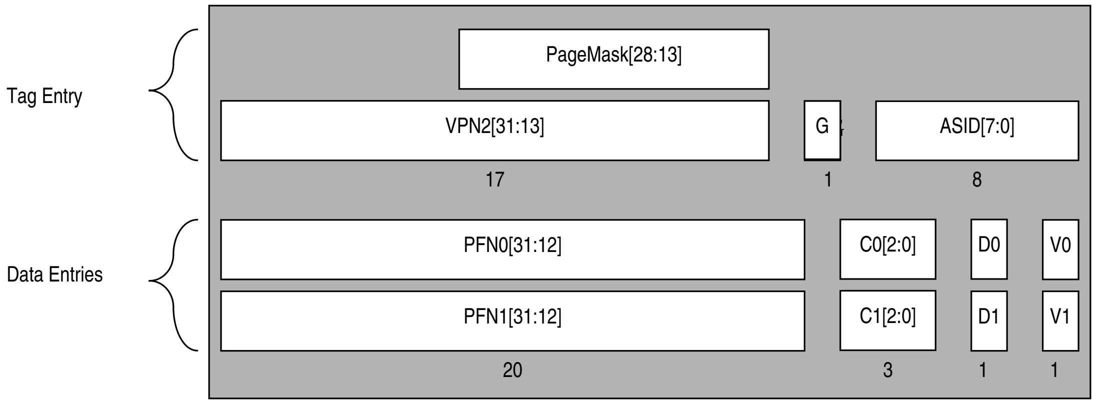


Table 5.7 and Table 5.8 explain each of the fields in a JTLB entry.

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.

PageMask	Page Size	Even/Odd Bank Select Bit
00_0000_0000_0000_00	4KB	VAddr[12]
00_0000_0000_0000_11	16KB	VAddr[14]
00_0000_0000_0011_11	64KB	VAddr[16]
00_0000_0000_1111_11	256KB	VAddr[18]
00_0000_0011_1111_11	1MB	VAddr[20]
00_0000_1111_1111_11	4MB	VAddr[22]
00_0011_1111_1111_11	16MB	VAddr[24]
00_1111_1111_1111_11	64MB	VAddr[26]
11_1111_1111_1111_11	256MB	VAddr[28]

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

Field Name	Description
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.
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:

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

Table 5.8 TLB Data Entry Fields (Continued)

Field Name	Description
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.

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 74K™ Core” on page 143 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 4-entry, fully-associative TLB dedicated to performing 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.

5.4 Virtual-to-Physical Address Translation

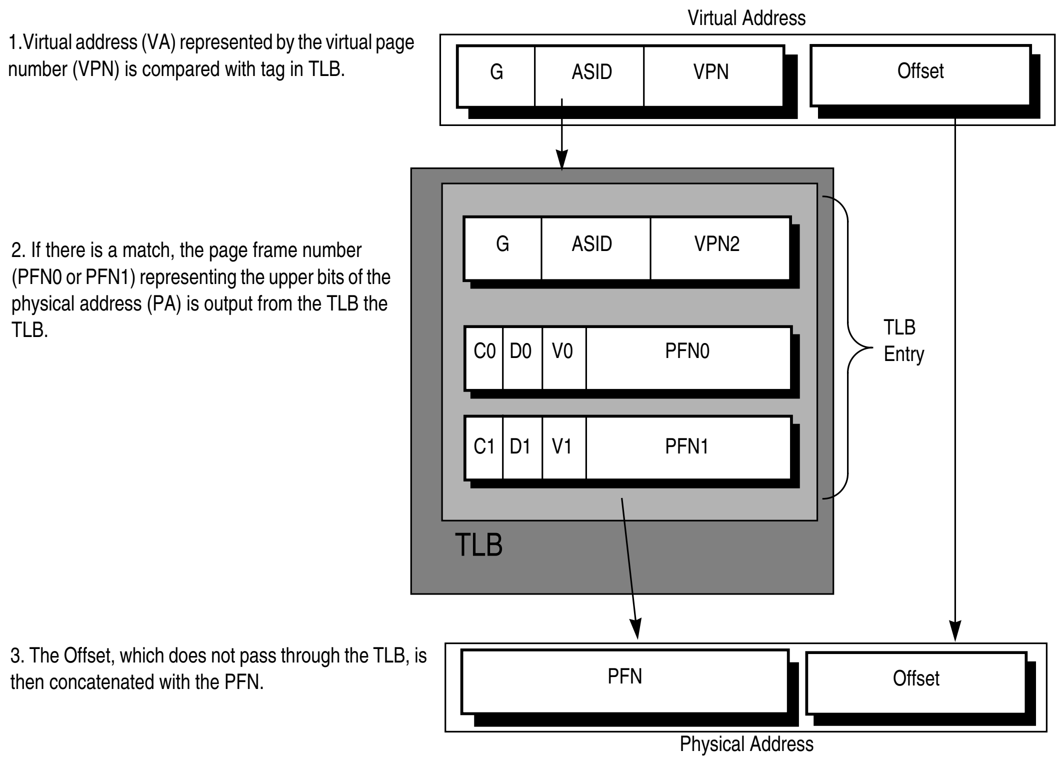
Converting a virtual address to a physical address begins by comparing the virtual address from the processor with the virtual addresses in the TLB. There is a match when the VPN of the address is the same as the VPN field of the entry, and either:

- 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 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 flushes during a context switch. This 8-bit ASID contains the number assigned to that process and is stored in the CP0 *EntryHi* register.

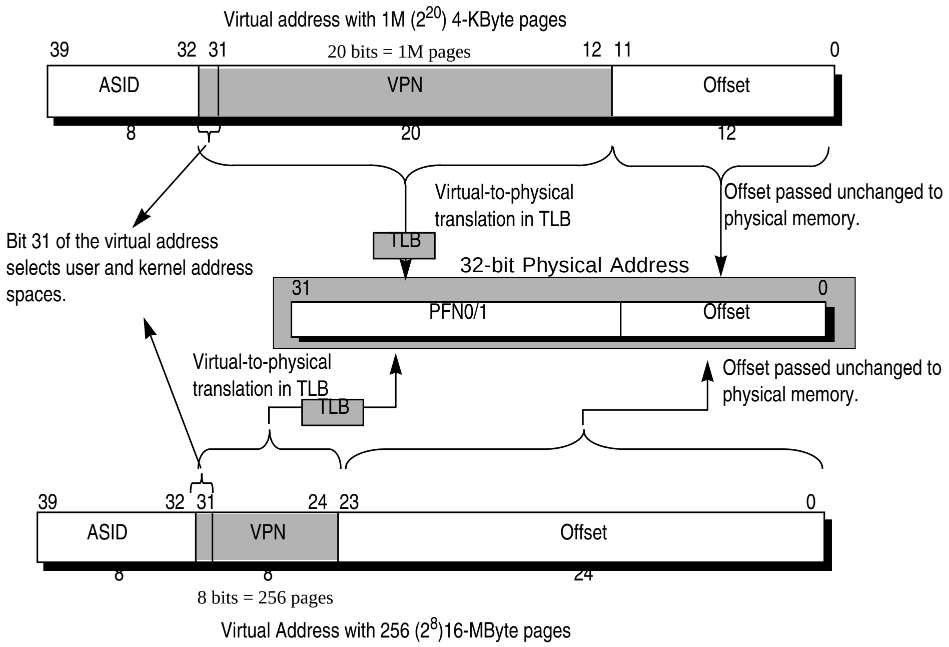
Figure 5.9 Overview of Virtual-to-Physical Address Translation


Virtual Address
1.Virtual address (VA) represented by the virtual page number (VPN) is compared with tag in TLB.
G
ASID
VPN
Offset
2. If there is a match, the page frame number (PFN0 or PFN1) representing the upper bits of the physical address (PA) is output from the TLB the TLB.
G
ASID
VPN2
C0
D0
V0
PFN0
C1
D1
V1
PFN1
TLB Entry
TLB
3. The Offset, which does not pass through the TLB, is then concatenated with the PFN.
PFN
Offset
Physical Address

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


Virtual address with 1M (2^{20}) 4-KByte pages
39
32
31
20 bits = 1M pages
12
11
0
ASID
VPN
Offset
8
20
12
Virtual-to-physical translation in TLB
Offset passed unchanged to physical memory.
Bit 31 of the virtual address selects user and kernel address spaces.
TLB
32-bit Physical Address
31
0
PFN0/1
Offset
Virtual-to-physical translation in TLB
TLB
Offset passed unchanged to physical memory.
39
32
31
24
23
0
ASID
VPN
Offset
8
8
24
8 bits = 256 pages
Virtual Address with 256 (2^8)16-MByte pages

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 4 KB to 256 MB in powers of 4. If a match is found, but the entry is invalid (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 livelock 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 entry in the TLB is valid, and the entry being written is valid, the core takes a machine-check exception, sets the TS bit in the CP0 *Status* register, and aborts the write operation. For further details on exceptions, please refer to Chapter 6, “Exceptions and Interrupts in the 74K™ Core” on page 107. There is a hidden bit in each TLB entry that is cleared on a Reset. This bit is set once the TLB entry is written and is included in the match detection. Therefore, uninitialized TLB entries will not cause a TLB shutdown.

Compared with previous cores from MIPS Technologies, the 74K core uses a more relaxed check for multiple matches in order to avoid machine check exceptions while flushing or initializing the TLB. On a write, all matching entries are disabled to prevent them from matching on future compares. A machine check is only signaled if the entry being written has its valid bit set, the matching entry in the TLB has its valid bit set, and the matching entry is not the entry being written. The cases for the signalling of the machine check exception are enumerated in Table 5.9.

Table 5.9 Machine Check Exception

Existing Match	Matching Entry equals Written Entry	Existing Page Valid Bit	Written Page Valid Bit	Machine Check
No	X	X	X	No
Yes	Yes	X	X	No
Yes	No	0	0	No
Yes	No	0	1	No
Yes	No	1	0	No
Yes	No	1	1	Yes

5.4.2 Memory Space

To assist in controlling both the amount of mapped space and the replacement characteristics of various memory regions, the 74K 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 KBytes to 256 MBytes, in multiples of 4. The *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 74K core implements the following page sizes:

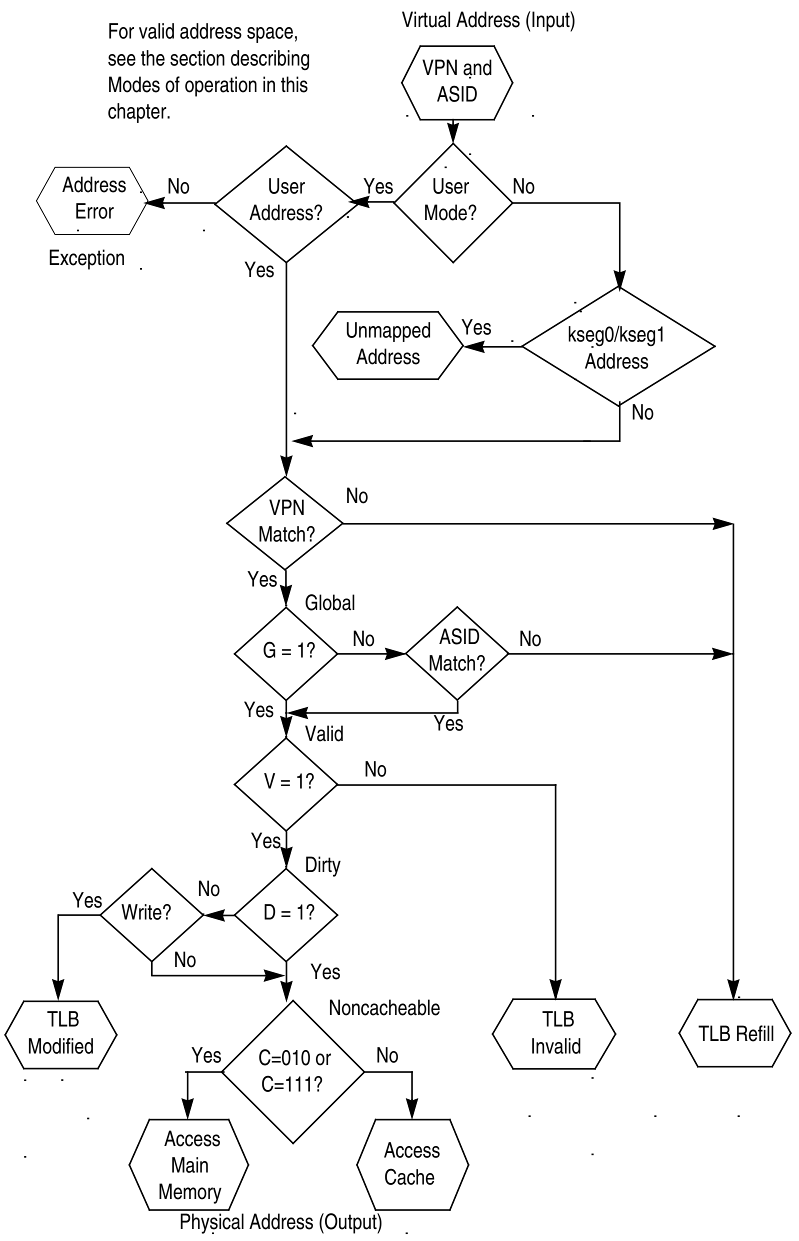
4K, 16K, 64K, 256K, 1M, 4M, 16M, 64M, and 256M.

Software can determine which page sizes are supported by writing all ones to the *PageMask* register, then reading the value back. For additional information, see Section 7.2.6 “PageMask (CP0 Register 5, Select 0): Control for Variable Page Size in TLB Entries”.

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 74K 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. Refer to Section 7.2.7 “Wired (CP0 Register 6, Select 0): Controls Number of Fixed ("wired") TLB Entries” for further details.

Figure 5.11 TLB Address Translation Flow in the 74K™ Processor Core



5.4.3 TLB Instructions

Table 5.10 lists the TLB-related instructions. Refer to Chapter 13, “74K™ Processor Core Instructions” on page 303 for more information on these instructions.

Table 5.10 TLB Instructions

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

Table 5.10 TLB Instructions (Continued)

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

5.5 Fixed Mapping MMU

The 74K 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 5.11 shows the encoding for the K23 (bits 30:28), KU (bits 27:25) and K0 (bits 2:0) fields of the *Config* register.

Table 5.11 Cache Coherency Attributes

Config Register Fields K23, KU, and K0	Cache Coherency Attribute
0	Cacheable, noncoherent, write-through, no write-allocate
1	Reserved
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 5.12 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 5.11 for the encoding.
kseg0	0x8000_0000-0x9FFF_FFFF	Controlled by the K0 field (bits 2:0) of the *Config* register. See Table 5.11 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 5.11 for the encoding.
kseg3	0xE000_0000-0xFFFF_FFFF	Controlled by K23 field (bits 30:28) of the *Config* register. Refer to Table 5.11 for the encoding.

The FM performs a simple translation to map virtual addresses to physical addresses. This mapping is shown in Figure 5.12.

When the ERL bit in the *Status* register is set, useg and kuseg are unmapped and uncached, just as they are when there is a TLB. The mapping when ERL = 1 is shown in Figure 5.13. The ERL bit is usually not asserted by software, but is asserted by hardware after a Reset, NMI, or Cache Error. See Section 6.8 "Exception Descriptions" for further information on exceptions.

Figure 5.12 FM Memory Map (ERL=0) in the 74K™ Processor Core

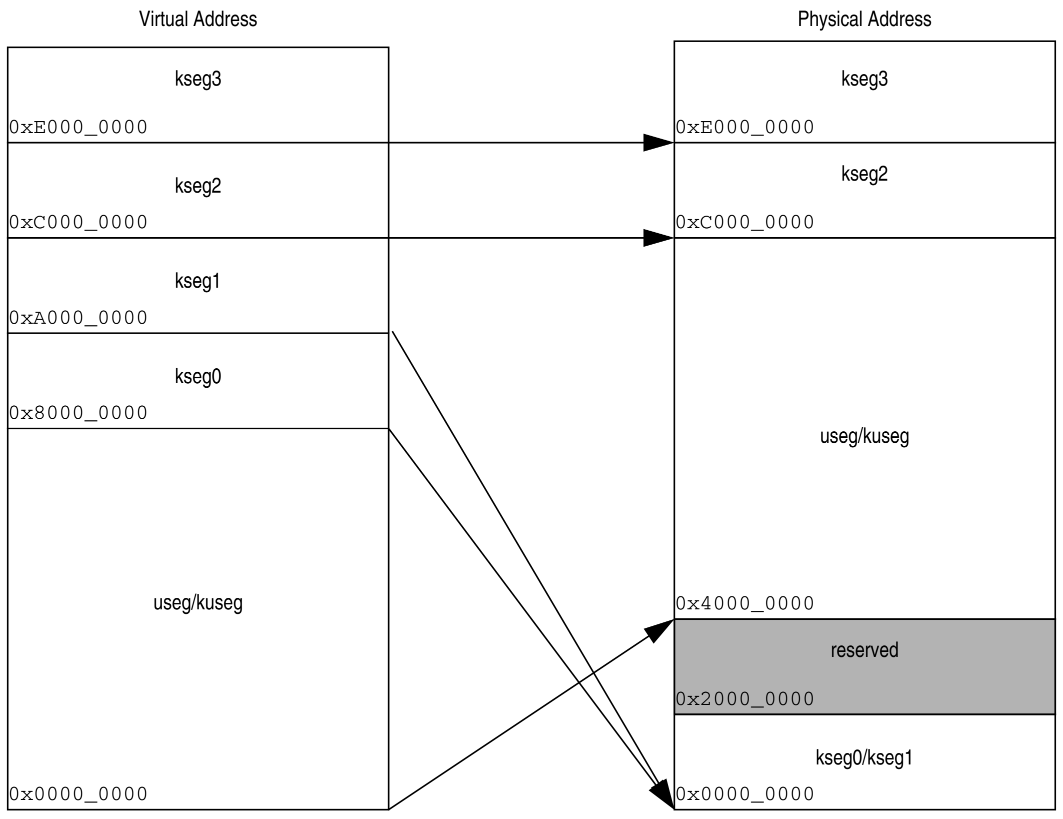
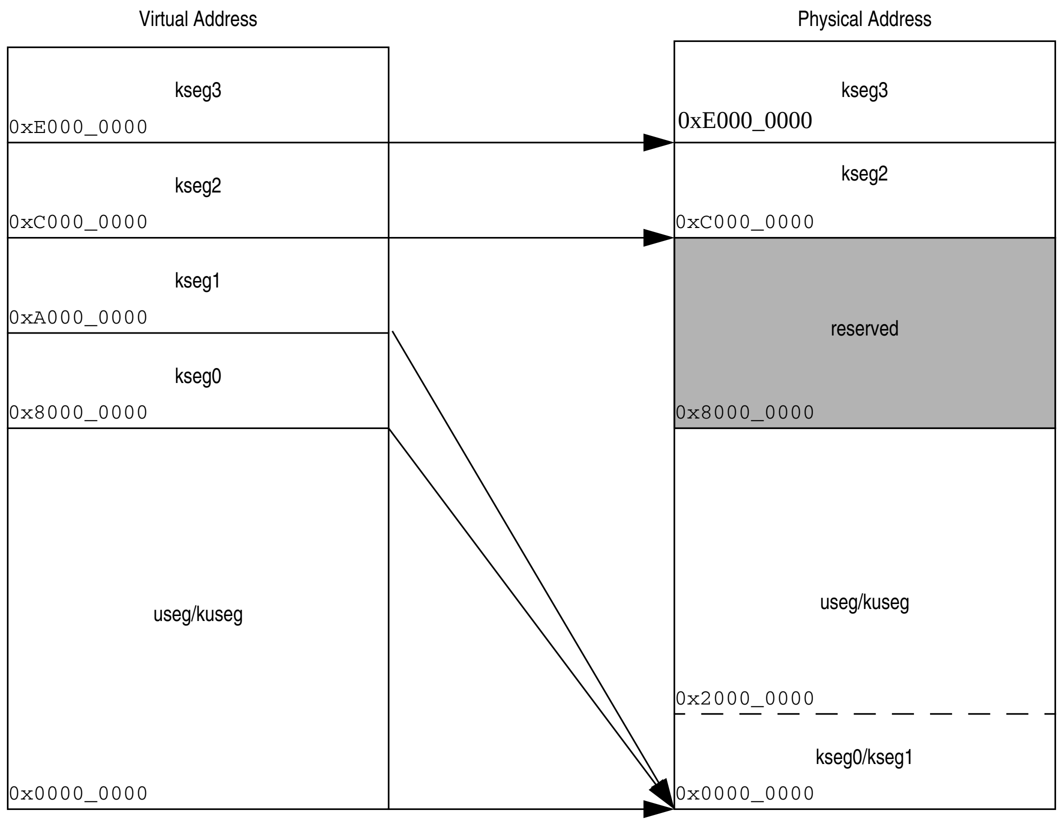


Figure 5.13 FM Memory Map (ERL=1) in the 74K™ Processor Core



Chapter 6

Exceptions and Interrupts in the 74K™ Core

The 74K processor core receives exceptions from a number of sources, including arithmetic overflows, misses in the translation lookaside buffer (TLB), I/O interrupts, and system calls. When the CPU detects an exception, the normal sequence of instruction execution is suspended and the processor enters kernel mode, disables interrupts, loads the *Exception Program Counter* (*EPC*) register with the location where execution can restart after the exception has been serviced, and forces execution of a software exception handler located at a specific address.

The software exception 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.

Exceptions may be precise or imprecise. Precise exceptions are those for which the *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 (as indicated by the *BD* bit in the *Cause* register), the address of the branch instruction immediately preceding the delay slot. Imprecise exceptions, on the other hand, are those for which no return address can be identified. Bus error exceptions and CP2 exceptions are examples of imprecise exceptions.

This chapter contains the following sections:

6.1 Exception Conditions

When an exception condition occurs, the instruction causing the exception 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.

When the 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 types of exceptions, this implementation allows all preceding instructions to complete execution and prevents all subsequent instructions from completing. Thus, the value in the *EPC* (or *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.

Imprecise exceptions are taken after the instruction that caused them has completed and potentially after following instructions have completed.

6.2 Exception Priority

Table 6.1 contains a list and a brief description of all exception conditions, The exceptions are listed in the order of their relative priority, from highest priority (Reset) to lowest priority (Load/store bus error). When several exceptions occur simultaneously, the exception with the highest priority is 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.
Machine Check	TLB write that conflicts with an existing entry.
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. Fetch reference to protected address.
TLBL	Fetch TLB miss. Fetch TLB hit to page with V=0.
ICache Error	Parity error on I-cache 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.

Table 6.1 Priority of Exceptions (Continued)

Exception	Description
DSPDis	DSP ASE State Disabled.
RI	Execution of a Reserved Instruction.
FPE	Floating Point exception.
Ov	Execution of an arithmetic instruction that overflowed.
Tr	Execution of a trap (when trap condition is true).
DDBL / DDBS	EJTAG Data Address Break (address only).
WATCH	A reference to an address in one of the watch registers (data).
AdEL	Load address alignment error. Load reference to protected address.
AdES	Store address alignment error. Store to protected address.
TLBL	Load TLB miss. Load TLB hit to page with V=0
TLBS	Store TLB miss. Store TLB hit to page with V=0.
TLB Mod	Store to TLB page with D=0.
DCache Error	Cache parity error. Imprecise.
DBE	Load or store bus error. Imprecise.

6.3 Interrupts

In the MIPS32® Release 1 architecture, support for exceptions included two software interrupts, six hardware interrupts, and a special-purpose timer interrupt. The timer interrupt was provided external to the core and was typically combined with hardware interrupt 5 in a system-dependent manner. Interrupts were handled either through the general exception vector (offset 0x180) or the special interrupt vector (0x200), based on the value of *CauseIV*. Software was required to prioritize interrupts as a function of the *CauseIV* bits in the interrupt handler prologue.

Release 2 of the Architecture, implemented by the 74K core, adds a number of upward-compatible extensions to the Release 1 interrupt architecture, including support for vectored interrupts and the implementation of a new interrupt mode that permits the use of an external interrupt controller.

Additionally, internal performance counters have been added to the 74K core. These counters can be configured to count various events within the core. When the MSB of the counter is set, it can trigger a performance counter interrupt. This interrupt, like the timer interrupt, is an output from the core that can be brought back into the core's interrupt pins in a system-dependent manner.

6.3.1 Interrupt Modes

The 74K core includes support for three interrupt modes, as defined by Release 2 of the Architecture:

- Interrupt Compatibility mode, in which the behavior of the 74K is identical to the behavior of 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. Although this mode is architecturally optional, it is always present on the 74K core, so the *VInt* bit will always read as a 1.

- External Interrupt Controller (EIC) mode, which redefines the way interrupts are handled to provide full support for an external interrupt controller that handles prioritization and vectoring of interrupts. As with VI mode, this mode is architecturally optional. The presence of this mode is denoted by the *VEIC* bit in the *Config3* register. On the 74K 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.

Following reset, the 74K processor defaults to Compatibility mode, which is fully compatible with all 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

StatusBEV	*CauseIV*	*IntCtlVS*	*Config3VINT*	*Config3VEIC*	**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	Cannot occur because *IntCtl*$_{VS}$ cannot 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 0x180 (if *Cause*$_{IV}$ = 0) or vector offset 0x200 (if *Cause* $_{IV}$ = 1). This mode is in effect when any of the following conditions are true:

- *Cause*$_{IV}$ = 0
- *Status*$_{BEV}$ = 1
- *IntCtl*$_{VS}$ = 0, which is the case if vectored interrupts are not implemented or have been disabled.

Here is a typical software handler for compatibility mode:

```
/*
 * Assumptions:
 *  - CauseIV = 1 (if it were zero, the interrupt exception would have to
 *                 be isolated from the general exception vector before arriving
 *                 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 IntCtlVS */
   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 simple 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 StatusIM 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 simply return to the interrupted code.
 */
   eret                       /* Return to interrupted code */

NestedException:
/*
 * Nested exceptions typically require saving the EPC and Status registers,
 * saving 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 cannot 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 */
        s      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) */
        l      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

In Vectored Interrupt (VI) mode, a priority encoder prioritizes pending interrupts and generates a vector which can be used to direct each interrupt to a dedicated handler routine. This mode also allows each interrupt to be mapped to a GPR shadow register set for use by the interrupt handler. VI mode is in effect when all the following conditions are true:

- *Config3*$_{VInt}$ = 1
- *Config3*$_{VEIC}$ = 0
- *IntCtl*$_{VS}$ ≠ 0
- *Cause*$_{IV}$ = 1
- *Status*$_{BEV}$ = 0

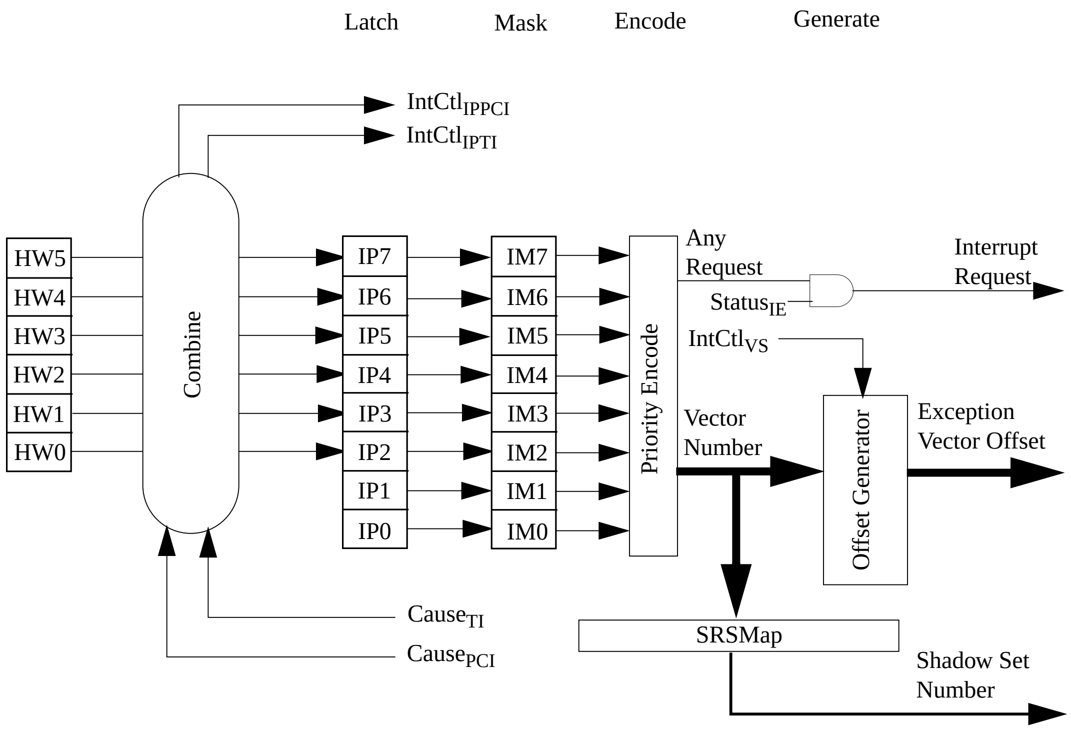
In VI mode, the six hardware interrupts are interpreted as individual hardware interrupt requests. The timer and performance counter interrupts are combined in a system-dependent way (external to the core) with the hardware interrupts (the interrupt with which they are combined is indicated by the *IntCtl*$_{IPTI/IPCI}$ 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 cannot 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 */
   s      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 SRSCtlPSS 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) */
   l      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 (EIC) 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 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, which prioritizes these interrupts in a system-dependent way with other hardware interrupts. The interrupt controller can be a hardwired logic block, or it can be configurable by 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.

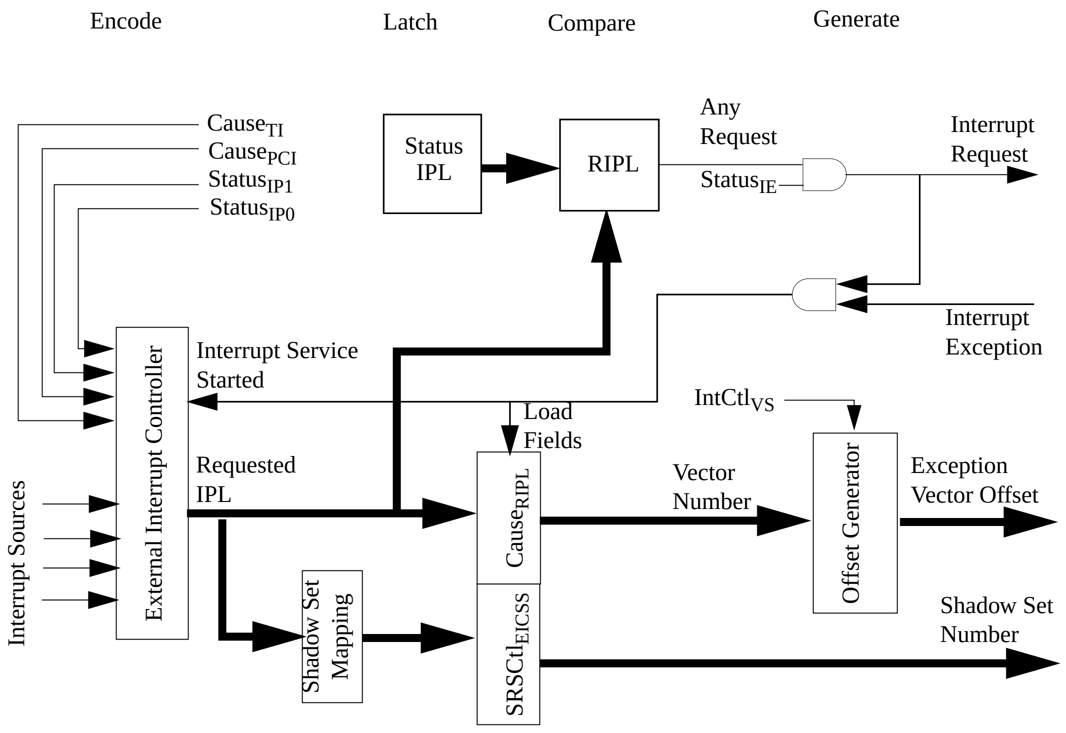
The external interrupt controller prioritizes its interrupt requests and produces the vector number of the highest priority interrupt to be serviced. The vector number, called the Requested Interrupt Priority Level (RIPL), is a 6-bit encoded value in the range 0..63, inclusive. The values 1..63 represent the lowest (1) to highest (63) RIPL for the interrupt to be serviced. A value of 0 indicates that no interrupt requests are pending. The interrupt controller inputs this value on the 6 hardware interrupt lines, which are treated as an encoded value in EIC mode.

$Status_{IPL}$ (which overlays $Status_{IM7..IM2}$) is interpreted as the Interrupt Priority Level (IPL) at which the processor is currently operating (a value of zero indicates 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 a 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 mode, the external interrupt controller is also responsible for supplying the GPR shadow register 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 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. Here is an example of such a routine:

```
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 SRSCtlPSS 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 0x200 to create the exception vector offset. For VI 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. 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				
	0b00001	**0b00010**	**0b00100**	**0b01000**	**0b10000**
0	0x0200	0x0200	0x0200	0x0200	0x0200
1	0x0220	0x0240	0x0280	0x0300	0x0400
2	0x0240	0x0280	0x0300	0x0400	0x0600
3	0x0260	0x02C0	0x0380	0x0500	0x0800
4	0x0280	0x0300	0x0400	0x0600	0x0A00
5	0x02A0	0x0340	0x0480	0x0700	0x0C00
6	0x02C0	0x0380	0x0500	0x0800	0x0E00
7	0x02E0	0x03C0	0x0580	0x0900	0x1000
• • •					
61	0x09A0	0x1140	0x2080	0x3F00	0x7C00
62	0x09C0	0x1180	0x2100	0x4000	0x7E00
63	0x09E0	0x11C0	0x2180	0x4100	0x8000

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

```
vectorOffset ← 0x200 + (vectorNumber × (IntCtl_VS || 0b00000))
```

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 74K 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 operate 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 (the set that was current before the last exception or interrupt occurred).

If the processor is operating in VI 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 are 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 are 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 shows 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 shows the vector base addresses when *SI_UseExceptionBase* = 1. As can be seen in Table 6.6, when *SI_UseExceptionBase* equals 1, the exception vectors for cases where $Status_{BEV}$ = 0 are not affected.

Table 6.7 shows 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. Table 6.4 (on page 117) shows the offset from the base address in the case where $Status_{BEV}$ = 0 and $Cause_{IV}$ = 1. 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, it is assumed that $IntCtl_{VS}$ = 0.

Table 6.5 Exception Vector Base Addresses, SI_UseExceptionBase = 0

Exception	$Status_{BEV}$	
	0	1
Reset, NMI	`0xBFC0.0000`	
EJTAG Debug (with *ProbEn* = 0 in the EJTAG_Control_register)	`0xBFC0.0480`	
EJTAG Debug (with *ProbEn* = 1 in the EJTAG_Control_register)	`0xFF20.0200`	

Table 6.5 Exception Vector Base Addresses, SI_UseExceptionBase = 0

Exception	$Status_{BEV}$	
	0	1
Cache Error	*EBase*$_{31..30}$ ‖ 1 ‖ *EBase*$_{28..12}$ ‖ `0x000` Note that *EBase*$_{31..30}$ have the fixed value `0b10`	`0xBFC0.0300`
Other	*EBase*$_{31..12}$ ‖ `0x000` Note that *EBase*$_{31..30}$ have the fixed value `0b10`	`0xBFC0.0200`
'‖' denotes bit string concatenation		

Table 6.6 Exception Vector Base Addresses, SI_UseExceptionBase = 1

Exception	$Status_{BEV}$	
	0	1
Reset, NMI	`0b10 ||` *SI_ExceptionBase* `[29:12] || 0x000`	
EJTAG Debug (with *ProbEn* = 0 in the EJTAG_Control_register)	`0b10 ||`*SI_ExceptionBase*`[29:12] || 0x480`	
EJTAG Debug (with *ProbEn* = 1 in the EJTAG_Control_register)	`0xFF20.0200`	
Cache Error	*EBase*$_{31..30}$ ‖ 1 ‖ *EBase*$_{28..12}$ ‖ `0x000` Note that *EBase*$_{31..30}$ have the fixed value `0b10`	`0b101 ||` *SI_ExceptionBase*`[28:12] ||` `0x300`
Other	*EBase*$_{31..12}$ ‖ `0x000` Note that *EBase*$_{31..30}$ have the fixed value `0b10`	`0b10 ||` *SI_ExceptionBase*`[29:12] ||` `0x200`
'‖' denotes bit string concatenation		

Table 6.7 Exception Vector Offsets

Exception	Vector Offset
TLB Refill, *EXL* = 0	`0x000`
General Exception	`0x180`
Interrupt, $Cause_{IV}$ = 1	`0x200` (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 (IntCtl$_{VS}$ = 0)
Reset, NMI	0	x	x	x	x	`0xBFC0.0000`
Reset, NMI	1	x	x	x	x	`0b10 ||` *SI_ExceptionBase*`[29:12] || 0x000`
EJTAG Debug	0	x	x	x	0	`0xBFC0.0480`
EJTAG Debug	1	x	x	x	0	`0b10 ||` *SI_ExceptionBase*`[29:12] || 0x480`
EJTAG Debug	x	x	x	x	1	`0xFF20.0200`
TLB Refill	0	0	1	x	x	`0xEBase[31:12] || 0x180`
TLB Refill	0	1	0	x	x	`0xBFC0.0200`
TLB Refill	1	1	0	x	x	`0b10 ||` *SI_ExceptionBase*`[29:12] || 0x200`
TLB Refill	0	1	1	x	x	`0xBFC0.0380`
TLB Refill	1	1	1	x	x	`0b10 || SI_ExceptionBase[29:12] || 0x380`
Cache Error	0	0	x	x	x	`0xEBase[31:30] || 0b1 || EBase[28:12] || 0x100`
Cache Error	0	1	x	x	x	`0xBFC0.0300`
Cache Error	1	1	x	x	x	`0b101 ||` *SI_ExceptionBase*`[28:12] || 0x300`
Interrupt	x	0	0	0	x	`0xEBase[31:12] || 0x180`
Interrupt	x	0	0	1	x	`0xEBase[31:12] || 0x200`
Interrupt	0	1	0	0	x	`0xBFC0.0380`
Interrupt	1	1	0	0	x	`0b10 ||` *SI_ExceptionBase*`[29:12] || 0x380`
Interrupt	0	1	0	1	x	`0xBFC0.0400`
Interrupt	1	1	0	1	x	`0b10 ||` *SI_ExceptionBase*`[29:12] || 0x400`
All others	0	0	x	x	x	`0xEBase[31:12] || 0x180`
All others	0	1	x	x	x	`0xBFC0.0380`
All others	1	1	x	x	x	`0b10 ||` *SI_ExceptionBase*`[29:12] || 0x380`

‘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 7.21). 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 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 begins executing 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 ← 0x180
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..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 ← 0x000
    elseif (ExceptionType = Interrupt) then
        if ($Cause_{IV}$ = 0) then
            vectorOffset ← 0x180
        else
            if ($Status_{BEV}$ = 1) or ($IntCtl_{VS}$ = 0) then
                vectorOffset ← 0x200
            else
                if $Config3_{VEIC}$ = 1 then
                    VecNum ← $Cause_{RIPL}$
                    NewShadowSet ← $SRSCtl_{EICSS}$
                else
                    VecNum ← VIntPriorityEncoder()
                    NewShadowSet ← $SRSMap_{IPL\times4+3..IPL\times4}$
                endif
                vectorOffset ← 0x200 + (VecNum × ($IntCtl_{VS}$ ‖ 0b00000))
            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 ← 0xBFC0.0200
else
    if ArchitectureRevision ≥ 2 then
        /* The fixed value of $EBase_{31..30}$ forces the base to be in kseg0 or kseg1 */
        vectorBase ← $EBase_{31..12}$ ‖ 0x000
    else
```

```
        vectorBase ← 0x8000.0000
    endif
endif

/* Exception PC is the sum of vectorBase and vectorOffset */
PC ← vectorBase$_{..30}$ ‖ (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 [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.

- The *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 [5:0]) in the *Debug* register.

No other CP0 registers or fields are changed due to the debug exception, and 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
1	0xFF20_0200 in dmseg

6.8 Exception Descriptions

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
$\text{Status}_{RP}$ ← 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* register 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 *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 *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:

```
$Status_{BEV} \leftarrow 1$
$Status_{TS} \leftarrow 0$
$Status_{SR} \leftarrow 0$
$Status_{NMI} \leftarrow 1$
$Status_{ERL} \leftarrow 1$
if InstructionInBranchDelaySlot then
    ErrorEPC ← PC - 4
else
    ErrorEPC ← PC
endif
PC ← 0xBFC0_0000
```

6.8.5 Machine Check Exception

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

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

***Cause* Register ExcCode Value:**

MCheck

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

6.8.6 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 6.3 "Interrupts" on page 109 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
CauseIP	Indicates the interrupts that are pending.

Entry Vector Used:

See 6.3.2 "Generation of Exception Vector Offsets for Vectored Interrupts" on page 117 for the entry vector used, depending on the interrupt mode the processor is operating in.

6.8.7 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

6.8.8 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* register 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 Watch Exception

Register State	Value
Cause$_{\text{WP}}$	Indicates that the watch exception was deferred until after *Status*$_{\text{EXL}}$, *Status*$_{\text{ERL}}$, and *Debug*$_{\text{DM}}$ 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$_{\text{I,R,W}}$	Set for the watch channel that matched, and indicates which type of match there was.

Entry Vector Used:

General exception vector (offset 0x180)

6.8.9 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 Address Exception Error

Register State	Value
BadVAddr	Failing address
Context$_{\text{VPN2}}$	UNPREDICTABLE
EntryHi$_{\text{VPN2}}$	UNPREDICTABLE
EntryLo0	UNPREDICTABLE
EntryLo1	UNPREDICTABLE

Entry Vector Used:

General exception vector (offset 0x180)

6.8.10 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 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*$_{\text{EXL}}$ = 0 at the time of exception;

General exception vector (offset 0x180) if *Status*$_{\text{EXL}}$ = 1 at the time of exception

6.8.11 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 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)

6.8.12 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 error

***Cause* Register ExcCode Value**

N/A

Additional State Saved

Table 6.16 CP0 Register States on Cache Error Exception

Register State	Value
CacheErr	Error state
ErrorEPC	Restart PC

Entry Vector Used

Cache error vector (offset 0x100)

6.8.13 Bus Error Exception — Instruction Fetch or Data Access

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

Instruction errors are precise, will Data bus errors can be imprecise. These errors are taken when the ERR code is returned on the *OC_SResp* input.

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)

6.8.14 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

6.8.15 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)

6.8.16 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)

6.8.17 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)

6.8.18 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 Coprocessor Unusable Exception

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

Entry Vector Used:

General exception vector (offset 0x180)

6.8.19 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)

6.8.20 Execution Exception — DSP ASE State Disabled

The DSP ASE State Disabled exception an execution exception. It occurs when an attempt is made to execute a DSP ASE instruction when the MX bit in the Status register is not set. This allows an OS to do “lazy” context switching.

Cause **Register ExcCode Value:**

DSPDis

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

6.8.21 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 Floating Point Exception

Register State	Value
FCSR	Indicates the cause of the floating point exception

Entry Vector Used:

General exception vector (offset 0x180)

6.8.22 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)

6.8.23 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)

6.8.24 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

6.8.25 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 6.19 Register States on 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
- TLB miss exceptions
- Reset and NMI exceptions
- Debug exceptions

Generally speaking, exceptions are handled by hardware and 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)

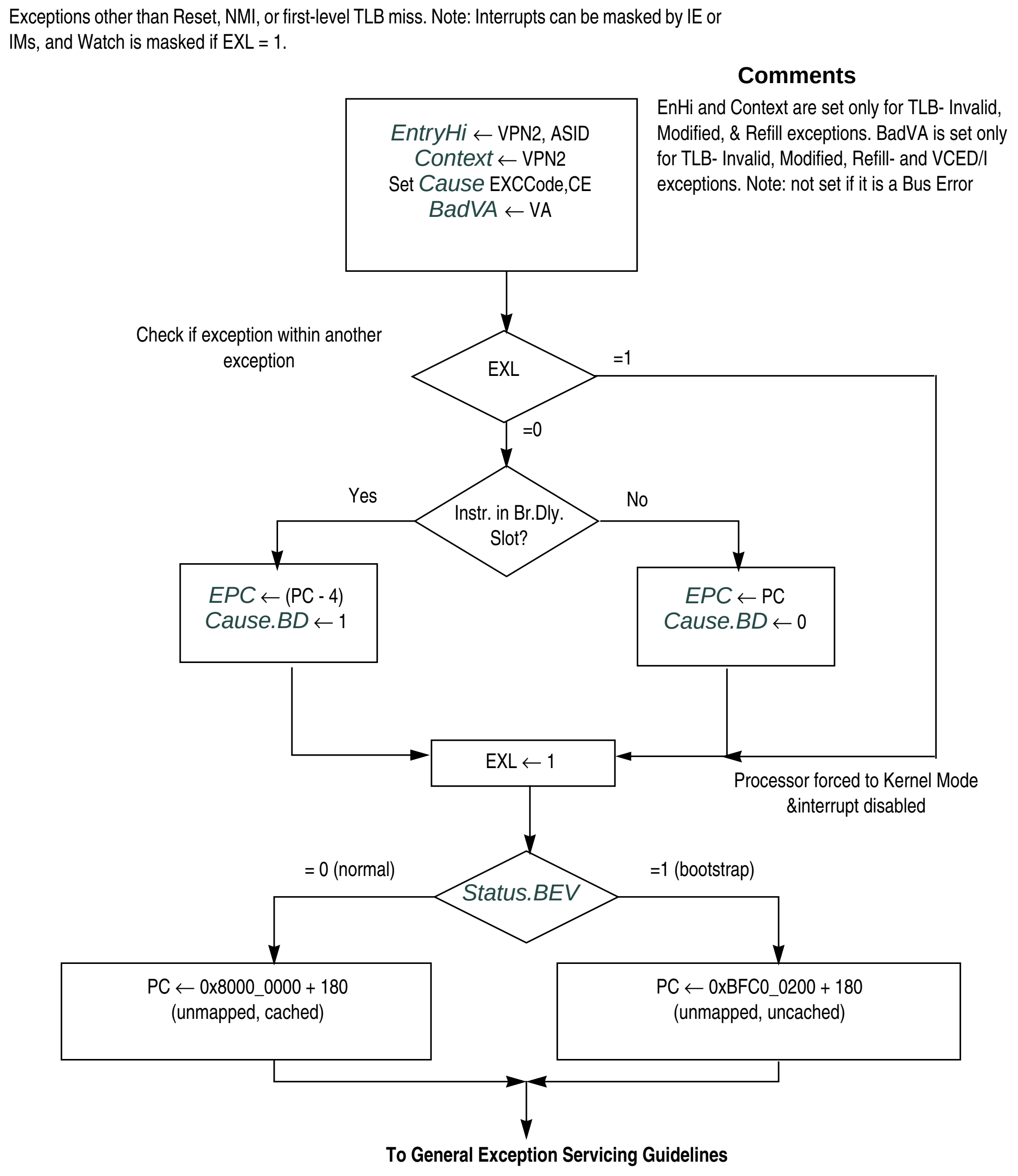

Exceptions other than Reset, NMI, or first-level TLB miss. Note: Interrupts can be masked by IE or IMs, and Watch is masked if EXL = 1.
Comments
EntryHi ← VPN2, ASID
Context ← VPN2
Set *Cause* EXCCode,CE
BadVA ← VA
EnHi and Context are set only for TLB- Invalid, Modified, & Refill exceptions. BadVA is set only for TLB- Invalid, Modified, Refill- and VCED/I exceptions. Note: not set if it is a Bus Error
Check if exception within another exception
EXL
=1
=0
Yes
Instr. in Br.Dly. Slot?
No
EPC ← (PC - 4)
Cause.BD ← 1
EPC ← PC
Cause.BD ← 0
EXL ← 1
Processor forced to Kernel Mode &interrupt disabled
= 0 (normal)
Status.BEV
=1 (bootstrap)
PC ← 0x8000_0000 + 180
(unmapped, cached)
PC ← 0xBFC0_0200 + 180
(unmapped, uncached)
To General Exception Servicing Guidelines

Figure 6.4 General Exception Servicing Guidelines (SW)

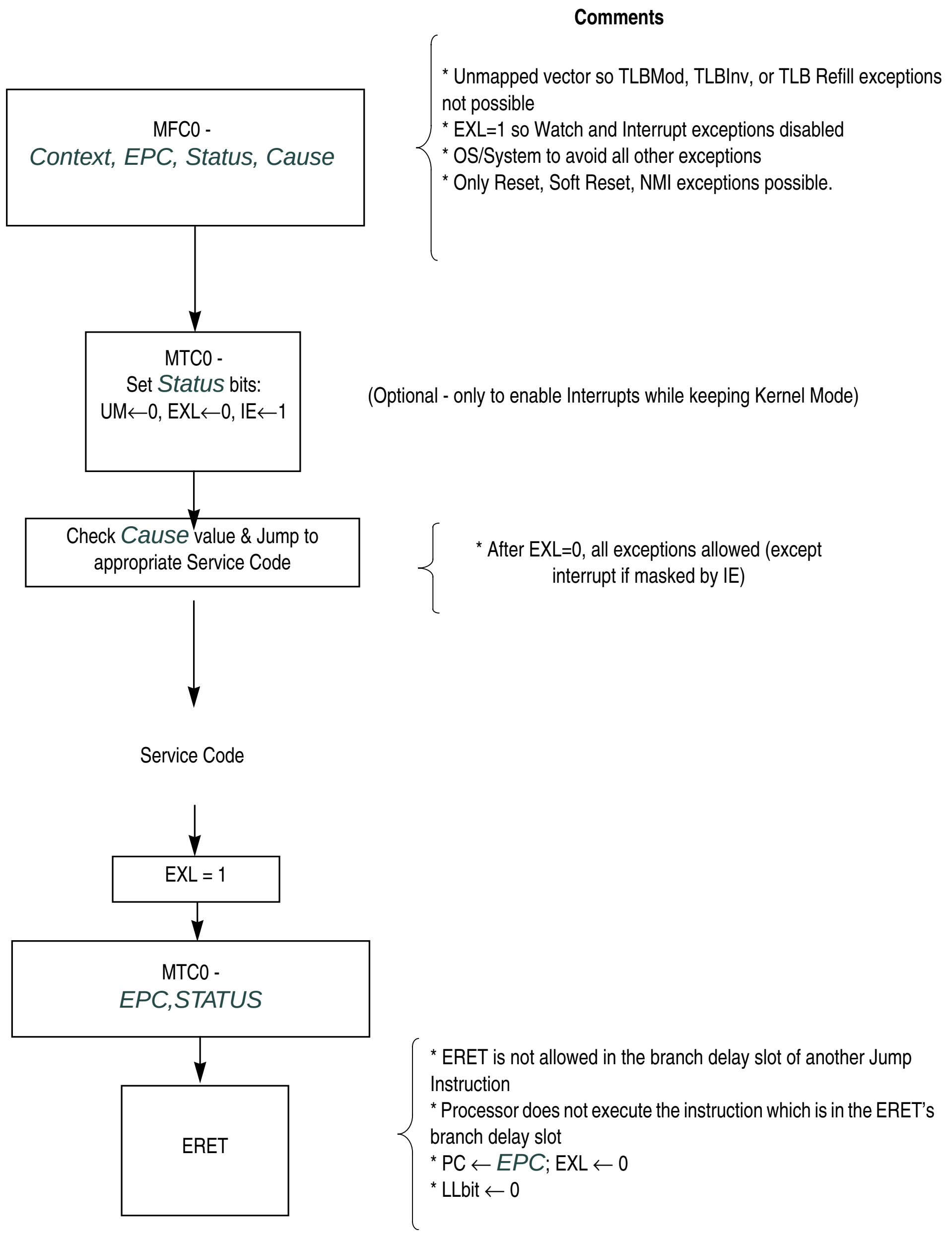

Comments
MFC0 - Context, EPC, Status, Cause
* Unmapped vector so TLBMod, TLBInv, or TLB Refill exceptions not possible
* EXL=1 so Watch and Interrupt exceptions disabled
* OS/System to avoid all other exceptions
* Only Reset, Soft Reset, NMI exceptions possible.
MTC0 - Set Status bits: UM←0, EXL←0, IE←1
(Optional - only to enable Interrupts while keeping Kernel Mode)
Check Cause value & Jump to appropriate Service Code
* After EXL=0, all exceptions allowed (except interrupt if masked by IE)
Service Code
EXL = 1
MTC0 - EPC,STATUS
ERET
* ERET is not allowed in the branch delay slot of another Jump Instruction
* Processor does not execute the instruction which is in the ERET's branch delay slot
* PC ← EPC; EXL ← 0
* LLbit ← 0

Figure 6.5 TLB Miss Exception Handler (HW)

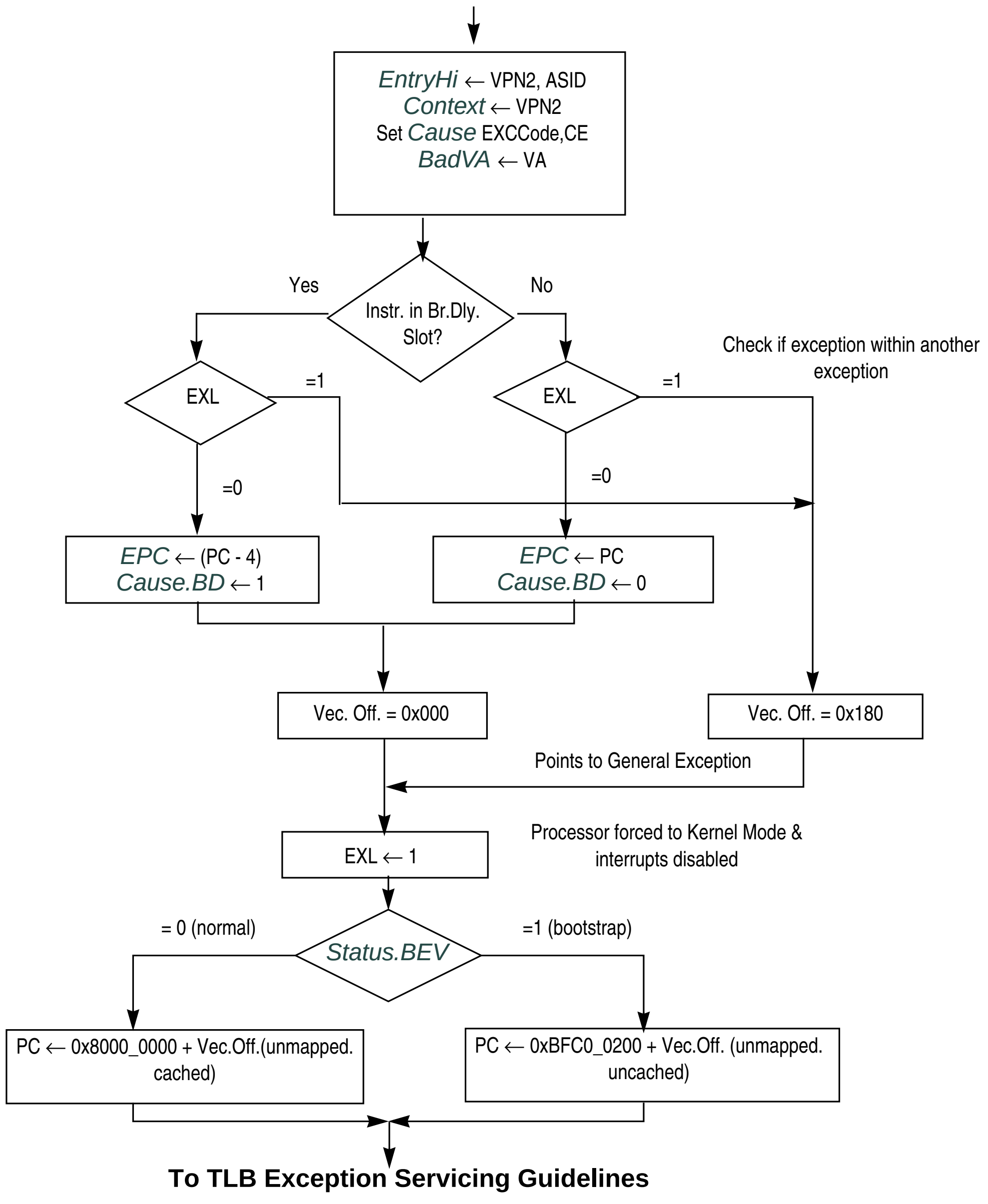

EntryHi ← VPN2, ASID
Context ← VPN2
Set *Cause* EXCCode,CE
BadVA ← VA
Yes
Instr. in Br.Dly. Slot?
No
Check if exception within another exception
EXL
=1
EXL
=1
=0
=0
EPC ← (PC - 4)
Cause.BD ← 1
EPC ← PC
Cause.BD ← 0
Vec. Off. = 0x000
Vec. Off. = 0x180
Points to General Exception
EXL ← 1
Processor forced to Kernel Mode & interrupts disabled
= 0 (normal)
Status.BEV
=1 (bootstrap)
PC ← 0x8000_0000 + Vec.Off.(unmapped. cached)
PC ← 0xBFC0_0200 + Vec.Off. (unmapped. uncached)
To TLB Exception Servicing Guidelines

Figure 6.6 TLB Exception Servicing Guidelines (SW)

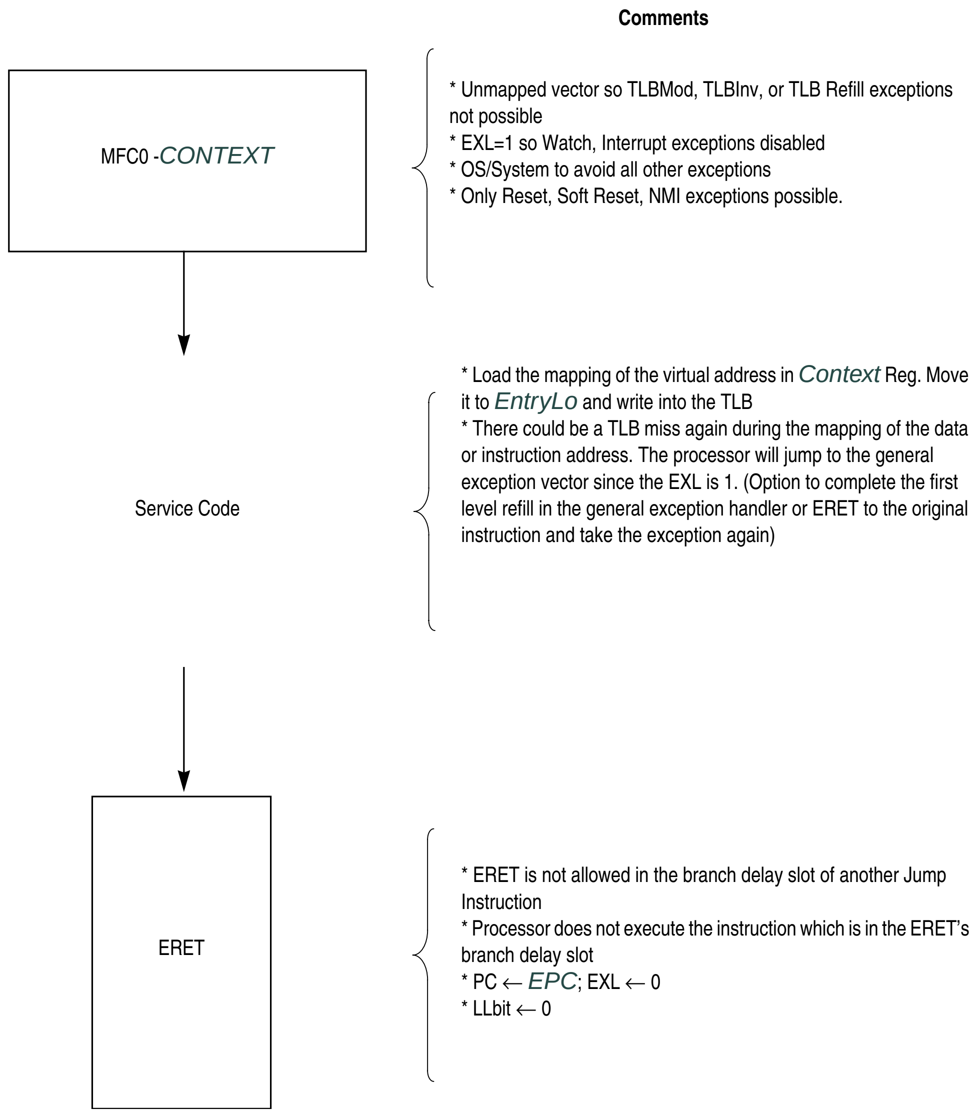
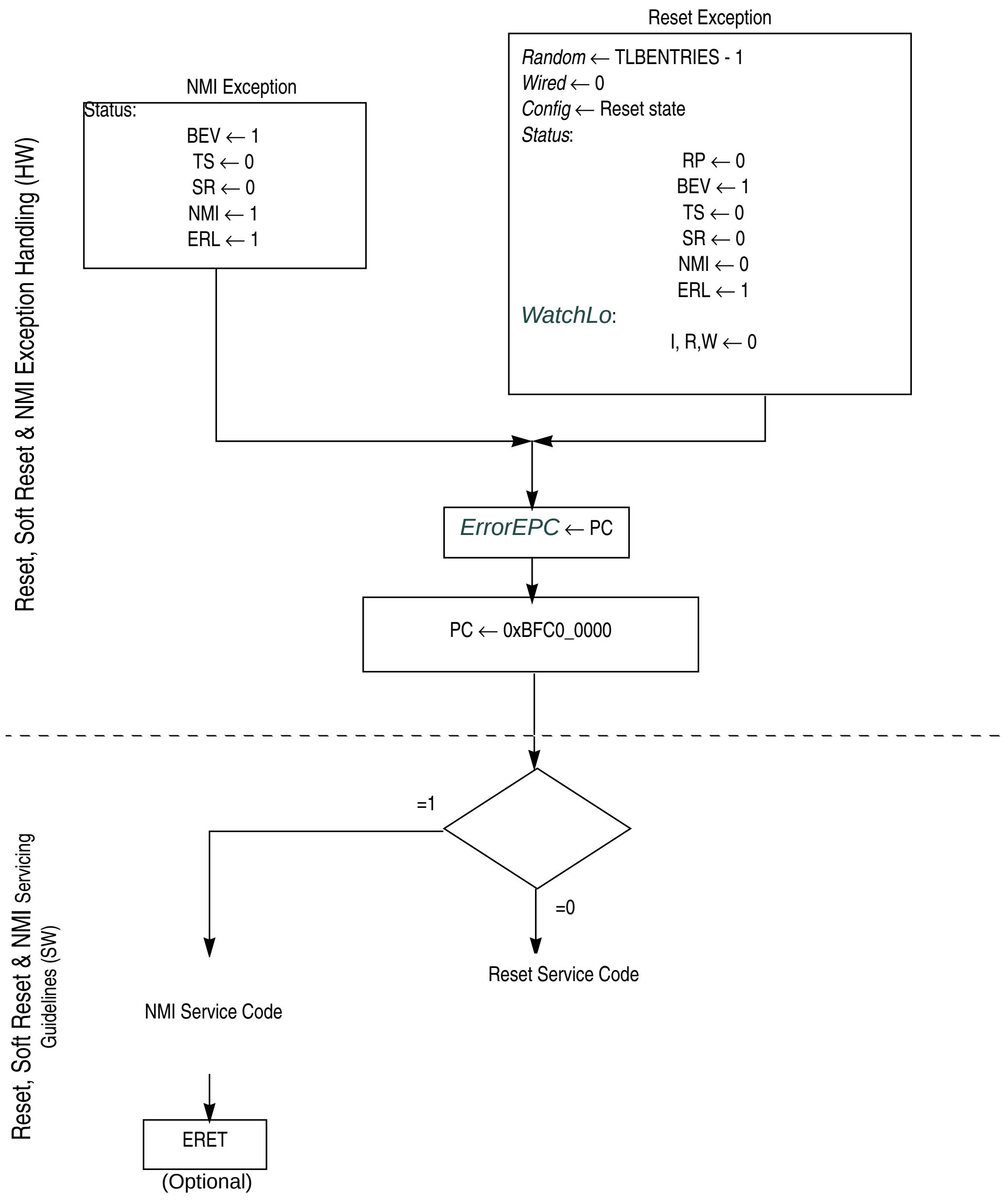

Comments
MFC0 -CONTEXT
* Unmapped vector so TLBMod, TLBInv, or TLB Refill exceptions not possible
* EXL=1 so Watch, Interrupt exceptions disabled
* OS/System to avoid all other exceptions
* Only Reset, Soft Reset, NMI exceptions possible.
Service Code
* Load the mapping of the virtual address in Context Reg. Move it to EntryLo and write into the TLB
* There could be a TLB miss again during the mapping of the data or instruction address. The processor will jump to the general exception vector since the EXL is 1. (Option to complete the first level refill in the general exception handler or ERET to the original instruction and take the exception again)
ERET
* ERET is not allowed in the branch delay slot of another Jump Instruction
* Processor does not execute the instruction which is in the ERET's branch delay slot
* PC ← EPC; EXL ← 0
* LLbit ← 0

Figure 6.7 Reset and NMI Exception Handling and Servicing Guidelines



Chapter 7

CP0 Registers of the 74K™ Core

The System Control Coprocessor (CP0) provides the register interface to the 74K processor core and supports memory management, address translation, exception handling, and other privileged operations. Each CP0 register has a unique number that identifies it, referred to as its *register number*. CP0 register numbers are denoted by `n.s`, where "`n`" is the register number (between 0-31) and "`s`" is the "select" field (0-7). If the select field is omitted, it is zero. A select field of "`x`" denotes all eight potential select numbers.

The CP0 EJTAG registers are described in Chapter 11, “EJTAG Debug Support in the 74K™ Core."

7.1 CP0 Register Summary

The CP0 registers are described in three tables. Table 7.1 lists the registers in alphabetic order, Table 7.2 lists the registers in numerical order, and Table 7.3 groups the registers according to their function. The CP0 registers are described individually in Section 7.2, "CP0 Register Descriptions."

Table 7.1 CP0 Registers in Alphabetical Order

Name	Number	Name	Number	Name	Number	Name	Number
BadVAddr	8.0	*DESAVE*	31.0	*IntCtl*	12.1	*Status*	12.0
CacheErr	27.0	*DTagHi*	29.2	*ITagHi*	29.0	*TraceControl*	23.1
Cause	13.0	*DTagLo*	28.2	*ITagLo*	28.0	*TraceControl2*	23.2
Compare	11.0	*EBase*	15.1	*L23DataHi*	29.5	*TraceControl3*	24.2
Config	16.0	*EntryHi*	10.0	*L23DataLo*	28.5	*TraceIPBC*	23.4
Config1-2	16.1-2	*EntryLo0-1*	2.0 3.0	*L23TagLo*	28.4	*TraceDPBC*	23.5
Config3	16.3	*EPC*	14.0	*PageMask*	5.0	*UserLocal*	4.2
Config7	16.7	*ErrCtl*	26.0	*PerfCnt0-3*	25.1 25.3 25.5 25.7	*UserTraceData1*	23.3
Context	4.0	*ErrorEPC*	30.0	*PerfCtl0-3*	25.0 25.2 25.4 25.6	*UserTraceData2*	24.3
Count	9.0	*HWREna*	7.0	*PRId*	15.0	*WatchHi0-3*	19.0-3
DDataLo	28.3	*IDataHi*	29.1	*Random*	1.0	*WatchLo0-3*	18.0-3
Debug	23.0	*IDataLo*	28.1	*SRSCtl*	12.2	*Wired*	6.0
DEPC	24.0	*Index*	0.0	*SRSMap*	12.3		

Table 7.2 CP0 Registers in Numerical Order

Number	Register	Description	Page
0.0	*Index*	Index into the TLB array	7.2.1, p.148
1.0	*Random*	Randomly generated index into the TLB array	7.2.2, p.149
2.0 3.0	*EntryLo0-1*	Output (physical) side of TLB entry (even-/odd-numbered virtual pages)	7.2.3, p.149
4.0	*Context*	Mixture of pre-programmed and *BadVAddr* bits which can act as an OS page table pointer.	7.2.4, p.151
4.2	*UserLocal*	Kernel-writable but user-readable software-defined thread ID	7.2.5, p.151
5.0	*PageMask*	Control for variable page size in TLB entries	7.2.6, p.153
6.0	*Wired*	Controls the number of fixed ("wired") TLB entries	7.2.7, p.154
7.0	*HWREna*	Bitmask limiting user-mode access to **`rdhwr`** registers	7.2.8, p.154
8.0	*BadVAddr*	Address causing the last TLB-related exception	7.2.9, p.155
9.0	*Count*	Free-running counter at pipeline or sub-multiple speed	7.10, p.155
10.0	*EntryHi*	High-order portion of the TLB entry	7.2.11, p.156
11.0	*Compare*	Timer interrupt control	7.2.12, p.157
12.0	*Status*	Processor status and control	7.2.13, p.157
12.1	*IntCtl*	Setup for interrupt vector and interrupt priority features.	7.2.14, p.160
12.2	*SRSCtl*	Shadow register set selectors	7.2.15, p.162
12.3	*SRSMap*	Shadow set choice for each interrupt level in VI mode	7.2.16, p.163
13.0	*Cause*	Cause of last general exception	7.2.17, p.164
14.0	*EPC*	Restart address from exception	7.2.18, p.166
15.0	*PRId*	Processor identification and revision	7.2.19, p.166
15.1	*EBase*	Exception entry point base address and CPU/VPE ID	7.2.20, p.168
16.0	*Config*	Legacy configuration register	7.2.21, p.169
16.1-2	*Config1-2*	MIPS32/64 configuration registers (caches etc)	7.2.22, p.170
16.3	*Config3*	Configuration register showing ASEs etc	7.2.23, p.172
16.6	*Config7*	CPU-specific configuration	7.2.24, p.173
18.0-3	*WatchLo0-3*	Watchpoint address and qualifiers	7.2.25, p.176
19.0-3	*WatchHi0-3*	Watchpoint control/status	7.2.26, p.176
23.0	*Debug*	EJTAG Debug status/control register	7.2.27, p.177
23.1	*TraceControl*	EJTAG Trace Control register	7.2.28, p.179
23.2	*TraceControl2*	EJTAG Trace Control2 register	7.2.29, p.182
23.3	*UserTraceData1*	EJTAG User Trace Data1 register	7.2.30, p.184
23.4	*TraceIBPC*	EJTAG Trace Instruction breakpoint control register	7.2.31, p.184
23.5	*TraceDBPC*	EJTAG Trace Data breakpoint control register	7.2.32, p.185
24.0	*DEPC*	Restart address from last EJTAG debug exception	7.2.33, p.187
24.2	*TraceControl3*	EJTAG Trace Control3 register	7.2.34, p.187
24.3	*UserTraceData2*	EJTAG User Trace Data2 register	7.2.30, p.184

Table 7.2 CP0 Registers in Numerical Order (Continued)

Number	Register	Description	Page
25.0 25.2 25.4 25.6	*PerfCtl0-3*	Performance counter control	7.2.35, p.189
25.1 25.3 25.5 25.7	*PerfCnt0-3*	Performance counters	7.2.36, p.193
26.0	*ErrCtl*	Software parity control and test modes for cache RAM arrays	7.2.37, p.193
27.0	*CacheErr*	Cache parity exception status	7.2.38, p.195
28.0	*ITagLo*	Read/write interface for I-cache tag cacheops	7.2.39, p.196
28.1	*IDataLo*	Low-order data read/write interface for I-cache special cacheops	7.2.40, p.198
28.2	*DTagLo*	Read/write interface for load/store tag cacheops	7.2.39, p.196
28.3	*DDataLo*	Low-order data read/write interface for D-cache	7.2.42, p.200
28.4	*L23TagLo*	Level 2/3 cache Tag information	7.2.43, p.201
28.5	*L23DataLo*	Low-order data read/write interface for Level 2/3 cache	7.2.44, p.201
29.0	*ITagHi*	I-cache predecode bits	7.2.45, p.201
29.1	*IDataHi*	High-order data read/write interface for I-cache special cacheops	7.2.46, p.202
29.2	*DTagHi*	D-cache virtual index (including ASID)	7.2.47, p.202
29.5	*L23DataHi*	High-order data read/write interface for Level 2/3 cache	7.2.48, p.203
30.0	*ErrorEPC*	Restart location from a reset or a cache error exception	7.2.49, p.203
31.0	*DESAVE*	Scratch read/write register for EJTAG debug exception handler	7.2.50, p.204

Table 7.3 CP0 Registers Grouped by Function

Basic modes	*Status*	12.0
OS/userland thread ID	*UserLocal*	4.2
Exception Control	*Cause*	13.0
	EPC	14.0
Timer	*Compare*	11.0
	Count	9.0
CPU Configuration	*Config*	16.0
	Config1-2	16.1-2
	Config3	16.3
	Config7	16.6-7
	EBase	15.1
	IntCtl	12.1
	PRId	15.0
	SRSCtl	12.2
	SRSMap	12.3
TLB Management	*BadVAddr*	8.0
	Context	4.0
	EntryHi	10.0
	EntryLo0-1	2.0 3.0
	Index	0.0
	PageMask	5.0
	Random	1.0
	Wired	6.0
Cache Management	*DDataLo*	28.3
	DTagHi	29.2
	DTagLo	28.2
	ErrCtl	28.2
	ErrorEPC	26.0
	IDataHi	29.1
	IDataLo	28.1
	ITagHi	29.0
	ITagLo	28.0
	L23DataHi	29.5
	L23TagLo	28.4
EJTAG Debug	*DEPC*	24.0
	DESAVE	31.0
	Debug	23.0
PDtrace	*TraceControl*	23.1
	TraceControl2	23.2
	TraceControl3	24.2
	TraceIPBC	23.4
	TraceIDBC	23.5
	UserTraceData1	23.3
	UserTraceData2	24.3
Profiling	*PerfCnt0-3*	25.1 25.3 25.5 25.7
	PerfCtl0-3	25.0 25.2 25.4 25.6
	PerfCnt0-3	25.1 25.3 25.5 25.7
Debug/Analysis	*WatchHi0-3*	19.0-3
	WatchLo0-3	18.0-3
Control rdhwr Access	*HWREna*	7.0
Parity/ECC control	*CacheErr*	27.0

Note that after a CP0 register has been updated, there is a hazard period of zero or more instructions from the update instruction (**`mfc0`**) until the update has taken effect in the core. The only hazard handled automatically by hardware in the 74K core is the **`mfc0`** instruction followed by the **`mtc0`** instruction. Release 2 requires that an **`ehb`** (Execution Hazard Barrier) instruction be placed between these two instructions when they address the same CPO register. This restriction is not required in the 74K core, though it would be prudent to use the **`ehb`** instruction to make the code more robust for future cores. For more information about the MIPS32® Release 2 Architecture guidelines on hazard barriers, refer to Section 2.3, "Hazards."

7.2 CP0 Register Descriptions

This section contains descriptions of each CP0 register. The registers are listed in numerical order, first by register number, then by select field number.

R/W Access Types

For each register described below, field descriptions include the read/write access properties of the field and the reset state of the field. The read/write access properties are described in Table 7.4.

Table 7.4 CP0 Register Field R/W AccessTypes

Notation	Hardware Interpretation	Software Interpretation
R/W	A field in which all bits are readable and writable by software and, potentially??, by hardware. Hardware updates of this field are visible by software reads. Software updates of this field are visible by hardware reads. If the reset state of this field is "Undefined", either software or hardware must initialize the value before the first read will return a predictable value. This should not be confused with the formal definition of **UNDEFINED** behavior.	
SO	Software Only. A field that is read and written by software but has no hardware effect. An example is the *DESAVE* register.	
R	A field that is either static or is updated only by hardware. If the Reset State of this field is either "0" or "Preset", hardware initializes this field to zero or to the appropriate state, respectively, on powerup. If the Reset State of this field is "Undefined", hardware updates this field only under those conditions specified in the description of the field.	A field to which the value written by software is ignored by hardware. Software may write any value to this field without affecting hardware behavior. Software reads of this field return the last value updated by hardware. If the Reset State of this field is "Undefined," software reads of this field result in an **UNPREDICTABLE** value except after a hardware update done under the conditions specified in the description of the field.
W	A field that can be written by software but which cannot be read by software. Software reads of this field will return an **UNDEFINED** value.	
W0	Hardware can write 1's or 0's to this field.	Software writes will only cause the bit to be cleared. Software can never set this bit. An example is the *NMI* bit field in the *Status* register.
W1C	Hardware can write 1's or 0's to this field.	Software should write "1" to this bit to clear it. An example is the *I*, *R*, and *W* bit fields in the *WatchHi0-3* register.
0	A field that hardware does not update, and for which hardware can assume a zero value.	A field to which the value written by software must be zero. Software writes of non-zero values to this field may result in **UNDEFINED** behavior of the hardware. Software reads of this field return zero as long as all previous software writes are zero. If the Reset State of this field is "Undefined", software must write this field with zero before it is guaranteed to read as zero.
U	A field that is not read or written by hardware.	Software writes to this field will be ignored. Software reads of this field will return an **UNDEFINED** value.

Color Coding of Register Descriptions

The color codes used in the register descriptions to indicate the access types are summarized in Figure 7.1. A field with two access types (for example, R/W0) is uncolored,

Figure 7.1 Register Format Color Coding of Access Field Types

31				5	0
R/W	SO	R	WRITE HAS UNUSUAL EFFECT (W, WO, W1C)	0	U

Power-up State of CP0 Registers

The traditions of the MIPS architecture regard it as software's job to initialize CP0 registers. As a rule, only fields where a wrong setting could prevent the CPU from booting are specified to be brought to a particular state by reset; other fields—perhaps other fields in the same register—are undefined. This manual documents where a field has a forced-from-reset value; conversely, when no reset-time value is documented, that means the register comes up in an undefined state.

To ensure robust programs, you should initialize all CP0 register fields, except those in which a random value is known to be harmless.

A Note on Unused Fields in CP0 Registers

Unused fields in registers are marked either with the digit 0, an "X", or occasionally a "U". A field marked zero is expected to read zero; a field marked "U" is expected to read back whatever you last wrote to it; and if the field is marked "X", the value is unpredictable.

But again, for robustness, you should write unused fields **either** to a value you previously read from the same field or (if no such value is available) to zero.

7.2.1 Index (CP0 Register 0, Select 0): Index into TLB array

Index is used as the TLB index when reading or writing the TLB with **`tlbr`**/**`tlbw`** respectively. It is also set by a TLB probe (**`tlbp`**) instruction to return the location of an address match in the TLB.

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 when the TLB is implemented; it is reserved if the FM is implemented.

Figure 7.2 Index Register Format

31	30 … 6	5 … 0
P	0	Index

Table 7.5 Field Descriptions for Index Register

Name	Bit(s)	Description	Read/ Write	Reset State
P	31	This bit is automatically set when a **`tlbp`** search of the TLB fails to find a matching entry.	R/W	Undefined
Index	5:0	An index into the TLB used for **`tlbwi`**, **`tlbr`**. It's set by **`tlbp`** when it finds a matching entry.	R/W	Undefined

7.2.2 Random (CP0 Register 1, Select 0): Randomly Generated Index into the TLB Array

The *Random* register is a read-only register whose value is used to index the TLB during a **`tlbwr`** instruction. It provides a quick way of replacing a TLB entry at random. *Random* is a free counter cycling through the range of valid TLB indexes. The *Random* register is decremented by one almost every clock, wrapping after the value in the *Wired* register is reached. As a result, it will not take values less than the value programmed in *Wired*.

To 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.

Figure 7.3 Random Register Format

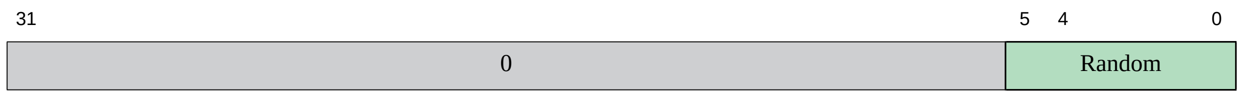


Table 7.6 Field Descriptions for Random Register

Name	Bit(s)	Description	Read/ Write	Reset State
Random	5:0	This field cycles "randomly" through the potential indices of the TLB, so its length varies with the TLB size (the diagram shows a max TLB size of 64 entries). It's usually a down counter, and starts off at the largest plausible index.	R	#TLB Entries — 1

7.2.3 EntryLo0-1 (CP0 Registers 2 and 3, Select 0): Output (physical) side of TLB entry

These registers hold and represent the contents of the physical (output) side of a TLB entry — each entry maps a pair of pages and *EntryLo0* and *EntryLo1* are for even-/odd-numbered virtual pages respectively. They're read during a **`tlbwr`** or **`tblw`** instruction, and written by a **`tlbr`**, and are not used for anything else.

Figure 7.4 EntryLo0, EntryLo1 Register Format

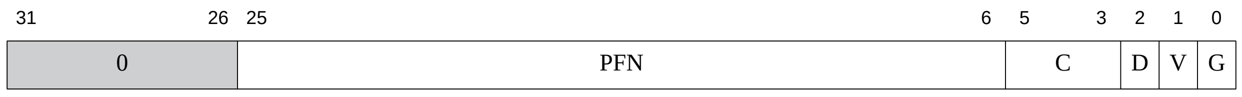


Table 7.7 Field Descriptions for EntryLo0-1 Register

Name	Bit(s)	Description	Read/ Write	Reset State
PFN	25:6	The "physical frame number" — traditional OS name for the high-order bits of the physical address. 20 bits of *PFN* together with 12 bits of in-page address make up a 32-bit physical address. The MIPS32® Architecture permits the *PFN* to be as large as 24 bits, but the 74K core has a 32-bit physical address bus.	R/W	Undefined
C	5:3	A code indicating how to cache this line: (see table below)	R/W	Undefined
D	2	The "dirty" flag. In hardware terms it's just a write-enable (when it's 0 you can't do a store using addresses translated here, you'll get an exception instead). However, software can use it to track pages which have been written to; when you first map a page you leave this bit clear, and then a first write causes an exception which you note somewhere in the OS' memory management tables (and of course remember to set the bit).	R/W	Undefined
V	1	The "valid" flag. You'd think it doesn't make much sense — why load an entry if it's not valid? But this is very helpful so you can make just one of a pair of pages valid.	R/W	Undefined
G	0	The "global" bit. This really belongs to the input side, and you don't really want two values for it. So you should always make sure this is the same in *EntryLo0* and *EntryLo1*.	R/W	Undefined

Code	Cached	How it writes	Notes
0	cached	write-through	An unusual choice for a high-speed CPU, probably only for debug
2	uncached		
3	cached	writeback	All normal cacheable areas
7	uncached	"Uncached Accelerated"	A special sort of write cycle which hardware tries to gather into bursts. Useful for high-bandwidth write-only hardware.

7.2.4 Context (CP0 Register 4, Select 0): Mixture of Pre-programmed and BadVAddr Bits which can act as an OS Page Table Pointer.

On any address-related exception (including all TLB-related exceptions) *Context* contains the useful mix of pre-programmed and borrowed-from-*BadVAddr* bits shown below.

Figure 7.5 Context Register Format

31 — 23	22 — 4	3 — 0
PTEBase	BadVPN2	0

Table 7.8 Field Descriptions for Context Register

Name	Bit(s)	Description	Read/ Write	Reset State
PTEBase	31:23	The *PTEBase* field is just software-writable and readable bits with no direct hardware effect. In a preferred scheme for software management of page tables, *PTEBase* can be set to the base address of a (suitably aligned) page table in memory; then the *BadVPN2* number (see below) comes from the virtual address associated with the exception—-it's just bits from *BadVAddr*, repackaged. In this case the virtual address bits are shifted such that each ascending 8Kbyte translation unit generates another step through a page table (assuming that each entry is 2&#215;32-bit words in size — reasonable since you need to store at least the two candidate *EntryLo0-1* values in it). An OS which can accept a page table in this format can contrive that in the time-critical simple TLB refill exception, *Context* automagically points to the right page table entry for the new translation. This is a great idea, but modern OS' tend not to use it — the demands of portability mean it's too much of a stretch to bend the page table information to fit this model.	R/W	Undefined
BadVPN2	22:4		R	Undefined

7.2.5 UserLocal (CP0 Register 4, Select 2): Address Causing the Last TLB-related Exception

UserLocal is a read-write 32-bit register that is not intepreted by the hardware and conditionally readable by software. This register is suitable for a kernel maintained thread ID whose value can be read by user-level code with **`rdhwr 29,`** as long as *HWRENA*$_{UL}$ is set.

Figure 7.6 IUserLocal Register Format

31 — 0
UserLocal

Table 7.9 UserLocal Register Field Description

Fields		Description	Read / Write	Reset State
Name	Bits			
UserLocal	31:0	Software information that is not interpreted by hardware	R/W	Undefined

7.2.6 PageMask (CP0 Register 5, Select 0): Control for Variable Page Size in TLB Entries

Every TLB entry has an independent virtual-address mask which allows it to ignore some address bits when deciding to match. By selectively ignoring lower page addresses, the entry can be made to match all the addresses in a "page" larger than 4KB.

Figure 7.7 PageMask Register Format

31 29	28 13	12 0
0	Mask	0

Table 7.10 Field Descriptions for PageMask Register

Name	Bit(s)	Description	Read/ Write	Reset State
Mask	28:13	Acts as a kind of backward mask, in that a 1 bit means "don't compare this address bit when matching this address". However, only a restricted range of *PageMask* values are legal (that's with "1"s filling the *PageMask*$_{Mask}$ field from low bits upward, two at a time):	R/W	Undefined

PageMask Value	Size of Each Output Page
`0x0000.0000`	4 Kbytes
`0x0000.6000`	16 Kbytes
`0x0001.E000`	64 Kbytes
`0x0007.E000`	256 Kbytes
`0x001F.E000`	1 Mbyte
`0x007F.E000`	4 Mbytes
`0x01FF.E000`	16 Mbytes
`0x07FF.E000`	64 Mbytes
`0x1FFF.E000`	256 Mbytes

Note that the uTLBs handle only 4Kbyte and 1Mbyte page sizes; other page sizes are down-converted to 4Kbyte or 1Mbyte as they are referenced. For other page sizes this may cause an unexpectedly high rate of uTLB misses, which could lead to a noticeable performance loss.

7.2.7 Wired (CP0 Register 6, Select 0): Controls Number of Fixed ("wired") TLB Entries

Wired can be set to a non-zero value to prevent the random replacement of that many TLB pages. It does this by preventing the *Random* register from taking values between 0 and the value of wired minus one: in turn that's done by arranging that the *Random* downcounter bounces back to its maximum value when it was previously equal to *Wired*.

Wired is set to zero at reset time.

Figure 7.8 Wired Register Format

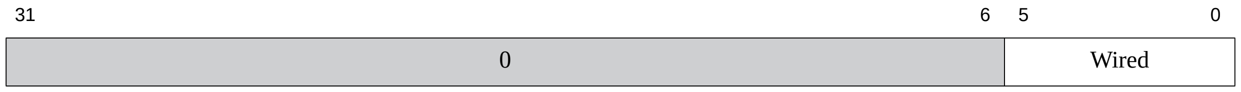


Table 7.11 Field Descriptions for Wired Register

Name	Bit(s)	Description	Read/ Write	Reset State
Wired	5:0	The oddly-named *Wired* controls *Random*'s behavior. *Random* is implemented as a full CPU clock-rate downcounter. It won't decrement below the current value of *Wired* (when it gets there it bounces off and starts again at the highest legal index). So in practice, when used inside the TLB refill exception handler, *Random* delivers a random index into the TLB somewhere between the value of *Wired* and the top. *Wired* can therefore be set to reserve some TLB entries from random replacement — a good place for an OS to keep translations which must never cause a TLB translation-not-present exception.	R/W	0

7.2.8 HWREna (CP0 Register 7, Select 0): Bitmask Limiting User-mode Access to rdhwr Registers

HWREna allows the OS to control which (if any) *hardware registers* are readable in user mode using **`rdhwr`**.

The low four bits [3:0] relate to the four registers required by the MIPS32® architecture standard. The two high bits [31:30] are available for implementation-dependent use.

The whole register is cleared to zero on reset, so that no hardware register is accessible without positive OS clearance.

Figure 7.9 HWREna Register Format

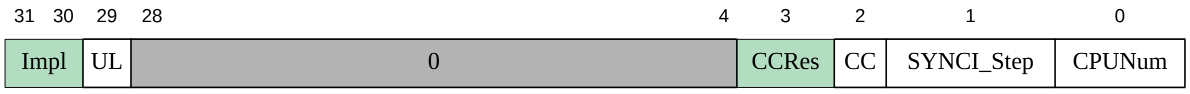


Table 7.12 Field Descriptions for HWREna Register

Name	Bit(s)	Description	Read/ Write	Reset State
Impl	31:30	Read 0. If there are any implementation-dependent hardware registers, you can control access to them here. Currently, no 74K family core has any such registers.	R	0

Table 7.12 Field Descriptions for HWREna Register

Name	Bit(s)	Description	Read/ Write	Reset State
UL	29	Set this bit to 1 to permit user programs to obtain the value of the *UserLocal* CP0 register through **`rdhwr 29`**	R/W	0
CCRes	3	Reads 0 if *Count* runs at the full clock rate (as it always does in 74K family CPUs); it would read 1 if *Count* ran at half speed.	R	0
CC	2	Set this bit 1 so a user-mode **`rdhwr 2`** can read out the value of the *Count* register.	R/W	0
SYNCI_Step	1	Set this bit 1 so a user-mode **`rdhwr 1`** can read the cache line size (actually, the smaller of the L1 I-cache line size and D-cache line size). That line size determines the step between successive uses of the **`synci`** instruction, which does the cache manipulation necessary to ensure that the CPU can correctly execute instructions which you just wrote.	R/W	0
CPUNum	0	Set this bit 1 so a user-mode **`rdhwr 0`** reads out the CPU ID number, as found in *EBase*$_{CPUNum}$.	R/W	0

7.2.9 BadVAddr (CP0 Register 8, Select 0): Address Causing the Last TLB-related Exception

BadVAddr is a plain 32-bit register, and is read-only.

It is set for the following exception types only: Address error (AdEL or AdES), TLB/XTLB Refill, TLB Invalid (TLBL, TLBS) and TLB Modified. There is more information in the notes to the *Cause*$_{ExcCode}$ field.

Figure 7.10 BadVAddr Register Format

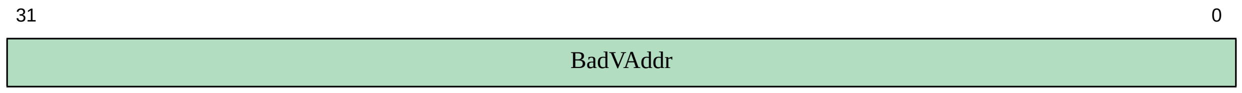


Table 7.13 BadVAddr Register Field Description

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

7.2.10 Count (CP0 Register 9, Select 0): Free-running Counter at Pipeline Speed

Count (a plain 32-bit register) counts up at the pipeline clock rate. *HWREna*$_{CCRes}$ flag reads 0 to indicate that the counting is at the pipeline clock rate as opposed to half-speed unlike some other cores.

Count may only ever stop in two circumstances. Firstly, some implementations may stop *Count* in the low-power mode entered through the **`wait`** instruction, but only if the $Cause_{DC}$ flag is set to 1. Secondly, you can arrange to stop *Count* in debug mode by setting $Debug_{CountDM}$

Count is writable and will carry on counting from whatever value is loaded into it. However, OS timers are usually implemented by leaving *Count* free-running and writing *Compare* as necessary.

Figure 7.11 Count Register Format

31 0
Count

Table 7.14 Count Register Field Description

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

7.2.11 EntryHi (CP0 Register10, Select 0): High-order Portion of TLB Entry

When reading and writing the TLB, *EntryHi* carries the "virtual-address side" information. During normal operation, $EntryHi_{ASID}$ defines the currently active address space.

Figure 7.12 EntryHi Register Format

31 13	12 8	7 0
VPN2	0	ASID

Table 7.15 Field Descriptions for EntryHi Register

Name	Bit(s)	Description	Read/ Write	Reset State
VPN2	31:13	$EntryHi_{VPN2}$ is the virtual address to be matched on a **`tlbp`**. It is also the virtual address to be written into the TLB on a **`tlbw`** and the destination of the virtual address on a **`tlbr`**. On a TLB-related exception the field *VPN2* is automagically set to the virtual address we were trying to translate when we got the exception. If — as is most often the case — the outcome of the exception handler is to find and install a translation to that address *VPN2* (and generally the whole of *EntryHi*) will turn out to already have the right values in it. It is written by software before a **`tlbp`** or **`tlbw`** and written by hardware in all other cases.	R/W	Undefined
ASID	7:0	This field does double-duty. It is used to stage data to and from the TLB, but in normal running software it's also the source of the current "ASID" value, used to extend the virtual address to make sure you only get translations for the current process.	R/W	0

7.2.12 Compare (CP0 Register 11, Select 0): Timer Interrupt Control

A plain 32-bit register, which may be written at any time. If the value of *Count* ever equals *Compare*, then the CPU activates the timer interrupt on the core output signal *SI_TimerInt*. The interrupt remains active until *Compare* is written again (typically, it's written with the counter value which corresponds to the next time you want to be notified.)

Figure 7.13 Compare Register Format

31 ... 0
Compare

Table 7.16 Compare Register Field Description

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

7.2.13 Status (CP0 Register 12, Select 0): Processor Status and Control

Status holds the basic processor operating mode (interruptibility, privilege levels, coprocessor accessibility). But it also has a number of informational and configuration bits, defined before there were ever any *Config* registers.

7.2.13.1 Interruptibility

Interrupts may be enabled **only** when:

$Status_{IE}$ == 1, $Status_{EXL}$ == 0, $Status_{ERL}$ == 0, and $Debug_{DM}$ == 0.

Even then you can disable interrupts individually using the $Status_{IM7-0}$ mask bits.

7.2.13.2 Privilege Levels

Debug Mode

The CPU is in Debug Mode if $Debug_{DM}$ is set, and can do anything.

Kernel Mode

The CPU is in kernel mode when $Debug_{DM}$ is zero and any of:
$Status_{EXL}$ == 1, $Status_{ERL}$ == 1, or $Status_{UM,SM}$ == 0.

In kernel mode the CPU has unrestricted access to all memory spaces (including, importantly, the "unmapped" regions kseg0 and kseg1), and to all the privileged (CP0) registers documented in this chapter, but it can't see some debug resources.

The processor resets into kernel mode because $Status_{ERL}$ is set from reset. And it goes into kernel mode on any exception because $Status_{EXL}$ is set (cache error exceptions set $Status_{ERL}$ instead.

In most OS', the processor only leaves kernel mode as a result of an **`eret`** instruction.

Supervisor Mode

The CPU is in supervisor mode when:
$Status_{EXL}$ == 0, *$Status_{ERL}$* == 0, *$Debug_{DM}$* == 0, and *$Status_{UM,SM}$* == 1.

In supervisor mode the CPU has access to the top half of the kseg2 region, sometimes known as kseg3. But it has no access to CP0 registers or most kernel memory. No known MIPS OS code uses supervisor mode, and it is not compatible (for example) with the "fixed mapping" MMU option.

User Mode

The CPU is in user mode when:
$Status_{EXL}$ == 0, *$Status_{ERL}$* == 0, *$Debug_{DM}$* == 0, and *$Status_{UM,SM}$* == 2.

In user mode the CPU has only access to the mapped kuseg address region. *Status*' fields are as follows:

7.2.13.3 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 7.14 Status Register Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15 … 8	7	6	5	4	3	2	1	0
CU3	CU2	CU1	CU0	RP	FR	RE	MX	PX	BEV	TS	SR	NMI	0	CEE	0	IM7-0	KX	SX	UX	UM	SM	ERL	EXL	IE

Table 7.17 Field Descriptions for Status Register

Name	Bit(s)	Description	Read/ Write	Reset State
CU3	31	Enables for different extension instruction sets. The *CU3-0* bits are for *co-processor* instruction sets, and are writable when such a coprocessor exists. Since no 74K family CPU has a co-processor 3, *$Status_{CU3}$* is hard-wired zero. *CU1* is most often used for a floating-point unit, if present, while *CU2* is reserved for a customer's coprocessor. Both become read-only and read zero if the corresponding coprocessor isn't fitted. Currently the 74K family of cores does not support coprocessor 2. Setting *$Status_{CU0}$* to 1 has the peculiar effect of allowing privileged instructions to work in user mode; not something a secure OS is likely to allow often. *MX* is set to 1 to enable instructions in *either* the MIPS DSP extension to the MIPS architecture, *or* the MDMX™ extension. The two may not be used together, and MDMX is unlikely to ever be available for 74K family core. But you can find out which by looking at *$Config3_{DSPP}$* (1 if MIPS DSP is implemented) and *$Config1_{MD}$* (1 if MIPS MDMX is implemented). *CEE* is 1 to enable instructions in the "CorExtend", user-definable instruction set. *$Config_{UDI}$* tells you whether your CPU has the CorExtend extension; but even then it may not use *CEE*. A user instruction set which uses only general-purpose registers and accumulators doesn't need disabling and may not use this bit.	R	0
CU2	30		R	0
CU1	29		R/W	Undefined
CU0	28		R/W	Undefined
MX	24		R/W	0
CEE	17		R/W	Undefined

Table 7.17 Field Descriptions for Status Register (Continued)

Name	Bit(s)	Description	Read/ Write	Reset State
RP	27	Reduced power — standard field. It's not connected inside the 74K core, but the state of the *RP* bit is available on the external core interface as the *SI_RP* signal. The 74K core uses clocks generated outside the core, and this could be used in your design to slow the input clock(s).	R/W	0
FR	26	If there is a floating point unit, set 0 for MIPS I compatibility mode (which means you have only 16 real FP registers, with 16 odd FP register numbers reserved for access to the high bits of double-precision values).	R/W	0
RE	25	Reverse endianness for instructions run in user mode. This feature is not supported in the 74K core and reads 0.	R	0
PX	23	Changes slightly to support 64-bit addressing, and these bits make that change for kernel-, supervisor- and user-privilege code respectively. For a 32-bit CPU, they are always zero. The $Status_{UX}$ bit has an additional role. When it's set zero, any attempt by a user-mode program to use an instruction specifically introduced for 64-bit operation results in an exception. With this bit clear, an operating system can run legacy 32-bit applications, guaranteeing that even accidental execution of data will produce precisely the same consequence as if you ran the software on a 32-bit CPU. That's sometimes unhelpful. If you want 32-bit addressing in user mode so $Status_{UX}$ is clear, but still want 64-bit instructions to handle data, you set $Status_{PX}$	R	0
KX	7		R	0
SX	6		R	0
UX	5		R	0
BEV	22	Boot Exception Vectors. When set to 1, relocates all *exception entry points* to near the reset-time start address.	R/W	1
TS	21	TLB Shutdown. Set if software attempts to create a duplicate TLB entry (which will also produce a "machine check" exception). Can be written back to zero, but never written to 1. The name of the field originated as a "TLB Shutdown" — historical MIPS CPUs quietly stopped translating addresses when they detected TLB abuse.	R/W	0
SR	20	Soft Reset. MIPS32 architecture "soft reset" bit: the 74K core's interface only supports a full external reset, so this always reads zero.	R	0
NMI	19	Non-maskable Interrupt. (read-only) — non-maskable interrupt shares the "reset" handler code, this field reads 1 when it was a NMI event which caused it.	R/W0	1 for NMI, 0 otherwise
IM7-0	15:8	Bitwise interrupt enable for the eight interrupt conditions also visible in $Cause_{IP7-0}$, **except** in the *"EIC" interrupt mode*. EIC mode is activated when $Config3_{VEIC}$ reads 1, and you set $Cause_{IV}$ and write a non-zero "vector spacing" into $IntCtl_{VS}$. In EIC mode *IM7-2* is recycled to become a 6-bit $Status_{IPL}$ ("interrupt priority level") field. An interrupt is only triggered when your interrupt controller presents an interrupt code which is numerically higher than the current value of $Status_{IPL}$. $Status_{IM1-0}$ always act as bitwise masks for the two software interrupt bits programmable at $CauseI_{P1-0}$.	R/W	Undefined

Table 7.17 Field Descriptions for Status Register (Continued)

Name	Bit(s)	Description	Read/ Write	Reset State
UM	4	Execution privilege level — basically user or kernel: | UM | SM | Mode | | 0 | 0 | kernel | | 0 | 1 | supervisor | | 1 | 0 | user | The intermediate "supervisor" privilege level is rarely used: but that's why this is a 2-bit field. Regardless of this field, the CPU is forced into kernel mode when either *EXL* or *ERL* is set.	R/W	Undefined
SM	3		R/W	Undefined
ERL	2	*EXL* is the regular exception mode bit, set automatically when the CPU takes an exception. *ERL* is the "error exception mode" bit, and is set following reset, an NMI, or a cache error exception. Either bit forces kernel mode and disables interrupts. There are some very special cases where nested exceptions are permitted, so an exception with *EXL* set does several strange things: a nested TLB Refill exception is sent to the general exception handler (not, as is usual, it's dedicated entry point), and on a nested exception *EPC*, *Cause*$_{BD}$ and *SRSCtl* are not overwritten. The result, broadly, is that when you return from the second exception you skip straight back to the code which was running before the first. For more details see the MIPS32 architecture documentation. The error level has its own return address: when *ERL* is set the **`eret`** instruction gets its address from *ErrorEPC*, not *EPC* as normal. Moreover, error level changes the memory map (in support of software fixing up cache errors), recycling kuseg as an uncached, unmapped window onto 512MB of physical memory.	R/W	1
EXL	1		R/W	Undefined
IE	0	Global interrupt enable, 0 to disable all interrupts. The **`di`**/**`ei`** instructions allow you to write this bit without affecting the rest of *Status*.	R/W	Undefined

7.2.14 IntCtl (CP0 Register 12, Select 1): Setup for Interrupt Vector and Interrupt Priority Features

Used to find out about interrupt wiring, and to set the "stride" for interrupt entry points in vectored-interrupt modes.

Figure 7.15 IntCtl Register Format

31 29	28 26	25 10	9 5	4 0
IPTI	IPPCI	0	VS	0

Table 7.18 Field Descriptions for IntCtl Register

Name	Bit(s)	Description	Read/ Write	Reset State
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.	R	Externally Set

Encoding	IP bit	Hardware Interrupt Source
2	2	HW0
3	3	HW1
4	4	HW2
5	5	HW3
6	6	HW4
7	7	HW5

Name	Bit(s)	Description	Read/ Write	Reset State
		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. 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.		
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. 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	is writable to give you software control of the vector spacing; if the value in *VS* is **VS**, you will get a spacing of $32 \times 2^{(VS-1)}$ bytes. Only values of 1, 2, 4, 8 and 16 work (to give spacings of 32, 64, 128, 256, and 512 bytes respectively). A value of zero gives a zero spacing, so all interrupts arrive at the same address — the legacy behaviour.	R/W	0

7.2.15 SRSCtl (CP0 Register12, Select 2): Shadow Register Set Selectors

Shadow sets are extra sets of general registers which can be selected when software takes an interrupt (or, rarely, some other exception).

Whether you have any shadow sets, and if so how many, is configurable by the SoC designer. If your CPU has shadow register sets, *SRSCtl*$_{HSS}$ will be non-zero. If no shadow sets are implemented, a read of this register returns all zeroes in *SRSCtl*$_{HSS}$.

Figure 7.16 SRSCtl Register Format

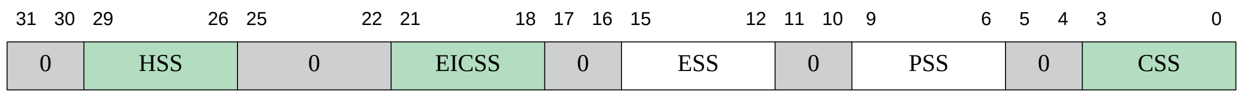


Table 7.19 Field Descriptions for SRSCtl Register

Name	Bit(s)	Description	Read/ Write	Reset State
HSS	29:26	The highest-numbered register set available on this processor(i.e. the number of available register sets minus one.) If it reads zero, your processor has just one set of GPR registers and no shadow-set facility. This field is read-only.	R	Preset
EICSS	21:18	In EIC mode, the external interrupt controller proposes a shadow register set number with each requested interrupt (nonzero IPL). When the processor takes an interrupt, the externally-supplied set number determines the next set and is made visible here in *SRSCtl*$_{EICSS}$ until the next interrupt. The CPU is in EIC mode if *Config3*$_{VEIC}$ (indicating the hardware is EIC-compliant), and software has set *Cause*$_{IV}$ to enable vectored interrupts. If the CPU is not in EIC mode, this field reads zero. In VI mode (no external interrupt controller, *Config3*$_{VInt}$ reads 1 and *Cause*$_{IV}$ has been set 1) the core sees only eight possible interrupt numbers; the *SRSMap* register contains eight 4-bit fields defining the register set to use for each of the eight interrupt levels. If you are remaining with "classic" interrupt mode (*Cause*$_{IV}$ is zero), it's still possible to use one shadow set for all exception handlers — including interrupt handlers — by setting *SRSCtl*$_{ESS}$ non-zero.	R	Undefined
ESS	15:12	This writable field is the software-selected register set to be used for "all other" exceptions; that's other than an interrupt in VI or EIC mode (both have their own special ways of selecting a register set). Unpredictable things will happen if you set *ESS* to a non-existent register set number (i.e., if you set it higher than the value in *SRSCtl*$_{HSS}$.	R/W	0

Table 7.19 Field Descriptions for SRSCtl Register (Continued)

Name	Bit(s)	Description	Read/ Write	Reset State
PSS	9:6	*CSS* is the register set currently in use, and is a read-only field. It's set on any exception, replaced by the value in $SRSCtl_{PSS}$ on an **`eret`**. *PSS* is the "previous" register set, which will be used following the next **`eret`**. It's writable, allowing the OS to dispatch code in a new register set; load this value and then execute an **`eret`**. If you write a larger number than the total number of implemented register sets the result is unpredictable. You can get at the values of registers in the previous set using **`rdpgpr`** and **`wrpgpr`**. Just a note: $SRSCtl_{PSS}$ and $SRSCtl_{CSS}$ are not updated by *all* exceptions, but only those which write a new return address to *EPC* (or equivalently, those occasions where the exception level bit $Status_{EXL}$ goes from zero to one). Exceptions where *EPC* is *not* written include: • Exceptions occurring with $Status_{EXL}$ already set; • Cache error exceptions, where the return address is loaded into *ErrorEPC*; • EJTAG debug exceptions, where the return address is loaded into *DEPC*.	R/W	0
CSS	3:0		R	0

7.2.16 SRSMap (CP0 Register 12, Select 3): Shadow Set Choice for Each Interrupt Level in VI Mode

This register's eight fields each proposes a shadow set to be used with the corresponding interrupt level, when in "vectored interrupt" (VI) mode. VI mode is active when $Config3_{VInt}$ reads 1, $Config3_{VEIC}$ reads zero and $Cause_{IV}$ is set 1.

This register is not implemented (and accesses to it have undefined results) if no shadow sets are provided: that is, if $SRSCtl_{HSS}$ reads zero.

Figure 7.17 SRSMap Register Format

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
SSV7	SSV6	SSV5	SSV4	SSV3	SSV2	SSV1	SSV0

Table 7.20 Field Descriptions for SRSMap Register

Name	Bit(s)	Description	Read/ Write	Reset State
SSV7	31:28	Shadow register set to be used in "VI" interrupt mode, when handling the interrupt for the respective *Cause*$_{IP7-0}$ bit. A zero shadow set number means not to use a shadow set, which is the default from reset. A number than the highest valid set (as found in *SRSCtl*$_{HSS}$) has unpredictable results: don't do that. If you are remaining with "classic" interrupt mode, it's still possible to use one shadow set for all exception handlers — including interrupt handlers — by setting *SRSCtl*$_{ESS}$ non-zero. In "EIC" interrupt mode, this register has no effect and the shadow set number to be used is determined by an input bus from the interrupt controller.	R/W	0
SSV6	27:24		R/W	0
SSV5	23:20		R/W	0
SSV4	19:16		R/W	0
SSV3	15:12		R/W	0
SSV2	11:8		R/W	0
SSV1	7:4		R/W	0
SSV0	3:0		R/W	0

7.2.17 Cause (CP0 Register 13, Select 0): Cause of Last General Exception

This register records information about the last exception, and is used by low-level exception handler code to decide what to do next. But it has a handful of writable fields too, detailed below.

Figure 7.18 Cause Register Format

31	30	29 28	27	26	25 24	23	22	21 16	15 10	9 8	7	6 2	1 0
BD	TI	CE	DC	PCI	0	IV	WP	0	IP7-2	IP1-0	0	ExcCode	0

Table 7.21 Field Descriptions for Cause Register

Name	Bit(s)	Description	Read/ Write	Reset State
BD	31	1 if the exception happened on an instruction in a branch delay slot; in this case *EPC* is set to restart execution at the branch, which is usually the correct thing to do. You need only consult *Cause*$_{BD}$ when you need to look at the instruction which caused the exception (perhaps to emulate it).	R	Undefined
TI	30	Last interrupt was from the on-core timer (see section below for *Count Compare*	R	Undefined
CE	29:28	If that was a "co-processor unusable" exception, this is the co-processor which you tried to use.	R	Undefined
DC	27	(writable) set 1 to disable the *Count* register during low-power operation following a **`wait`** instruction.	R/W	0
PCI	26	Last interrupt was an overflow from the performance counters, see the *PerfCnt* registers.	R	Undefined
IV	23	(writable) set 1 to use a special *exception entry point* for interrupts. It's quite likely that if you're doing this, you're also using multiple entry points for different interrupt levels, see the *IntCtl* register.	R/W	Undefined

Table 7.21 Field Descriptions for Cause Register (Continued)

Name	Bit(s)	Description	Read/ Write	Reset State
WP	22	(writable to zero) — remembers that a watchpoint triggered when the CPU couldn't take the exception because it was already in exception mode (or error-exception mode, or debug mode). Since this bit automagically causes the exception to happen again, it must be cleared by the watchpoint exception handler.	R/W	Undefined
IP7-2	15:10	So long as the CPU is not in EIC interrupt mode, this field reflects the current state of the interrupt request inputs to the core. When one of them is active and enabled by the corresponding *Status*$_{IM7-0}$ bit, an interrupt may occur. The CPU is in EIC mode if *Config3*$_{VEIC}$ (indicating the hardware is EIC-compliant), and software has set *Cause*$_{IV}$ to enable vectored interrupts. In that case this field is interpreted as an unsigned binary number, and is a snapshot of the value of the "interrupt priority level" (IPL) supplied by the interrupt controller. The snapshot is from the time when the CPU decided to take the interrupt exception. When the presented IPL is higher than the current interrupt priority level held in *StatusI*$_{M7-2}$, the CPU takes an interrupt. A zero level on the core inputs indicates no interrupt request. *IP1-0* are writable, and in fact always just reflect the value written here. They act as software interrupt bits masked by *StatusI*$_{M1-0}$ regardless of the interrupt mode.	R	Undefined
IP1-0	9:8		R/W	Undefined
ExcCode	6:2	What caused that last exception. Lots of values, listed in Table 7.22 below.	R	Undefined

Table 7.22 Exception Code values in ExcCode Field of Cause Register

Value	Code	What just happened?
0	`Int`	Interrupt
1	`Mod`	Store, but page marked as read-only in the TLB
2	`TLBL`	Load or fetch, but page marked as invalid in the TLB
3	`TLBS`	Store, but page marked as invalid in the TLB
4	`AdEL`	Address error on load/fetch or store respectively. Address is either wrongly aligned, or a privilege violation.
5	`AdES`	
6	`IBE`	Bus error signaled on instruction fetch
7	`DBE`	Bus error signaled on load/store (imprecise)
8	`Sys`	System call, i.e. **`syscall`** instruction executed.
9	`Bp`	Breakpoint, i.e. **`break`** instruction executed.
10	`RI`	Instruction code not recognized (or not legal)
11	`CpU`	Instruction code was for a co-processor which is not enabled in *Status*$_{CU3-0}$.
12	`Ov`	Overflow from a trapping variant of integer arithmetic instructions.
13	`Tr`	Condition met on one of the conditional trap instructions **`teq`** etc.

Table 7.22 Exception Code values in ExcCode Field of Cause Register

Value	Code	What just happened?
14	-	Reserved
15	`FPE`	Floating point unit exception — more details in the FPU control/status registers.
16-17	-	Available for implementation dependent use
19-22	-	Reserved
23	`WATCH`	Instruction or data reference matched a watchpoint
24	`MCheck`	"Machine check"
25	`Thread`	Thread-related exception, only for CPUs supporting the MIPS MT ASE. In that case the cause is further detailed in *VPEControl*$_{EXCPT}$.
26	`DSP`	Tried to run an instruction from the MIPS DSP ASE, but it's either not enabled or not available. In particular, *Status*$_{MX}$ is zero).
27-29	-	Reserved
30	`CacheErr`	Parity/ECC error somewhere in the core, on either instruction fetch, load or cache refill. In fact you never see this value in *Cause*$_{ExcCode}$; but some of the codes in this table including this one can be visible in the "debug mode" of the EJTAG debug unit — see and in particular the notes on the *Debug* register.
31	-	Reserved

7.2.18 EPC (CP0 Register 14, Select 0): Restart Address from Exception

After any normal exception (debug and error exceptions are different, see *DEPC* and *ErrorEPC* respectively), *EPC* holds the return address.

If the instruction we'd really like to return to is in a branch delay slot, *EPC* points to the branch instruction and *Cause*$_{BD}$ will be set. All MIPS branch instructions may be re-executed successfully, so returning to the branch is the right thing to do in this case.

Figure 7.19 EPC Register Format

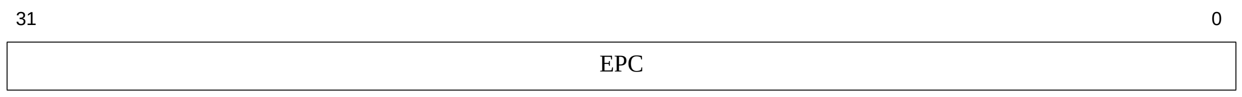


Table 7.23 EPC Register Field Description

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

7.2.19 PRId (CP0 Register 15, Select 0): Processor Identification and Revision

Identifies the CPU to software. It's appropriately printed as part of the start-up display by any software telling the world about its start-up; but when portable software is configuring itself around different CPU attributes, it's always preferable to sense those attributes directly — look in one of *Config*, *Config1-2*, *Config3* or perhaps a directed software probe.

Figure 7.20 PRId Register Format

31 … 24	23 … 16	15 … 8	7 … 0
CoOpt	CoID	Imp	Rev

Table 7.24 Field Descriptions for PRId Register

Name	Bit(s)	Description	Read/ Write	Reset State
CoOpt	31:24	Whatever is specified by the SoC builder who synthesizes the core — refer to your SoC manual. It should be a number between 0 and 127 — higher values are reserved by MIPS Technologies.	R	Preset
CoID	23:16	Identifies the company that designed or manufactured the processor. In the 74K, this field contains a value of 1 to indicate MIPS Technologies, Inc.	R	1
Imp	15:8	Identifies the type of processor. This field allows software to distinguish between the various types of MIPS Technologies processors. The value of this field is 0x97 for the 74K core.	R	0x97
Rev	7:0	The revision number of the core design.	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

7.2.20 EBase (CP0 Register 15, Select 1): Exception entry point base address and CPU/VPE ID

EBase does two vital jobs: one is to allow software to know which CPU it's running on and the other is to relocate the exception entry points. It is primarily supplied for multi-CPU systems .

The latter is necessary because CPUs sharing a memory map have their exception entry points in kseg0. By setting *EBase* differently on each CPU, you can give them distinct exception handlers.

Figure 7.21 EBase Register Format

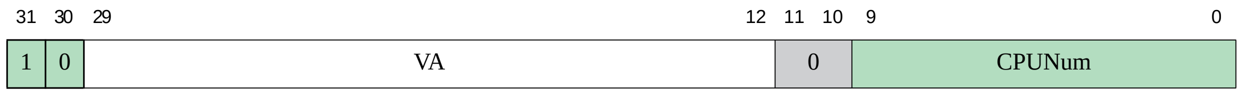


Table 7.25 Field Descriptions for EBase Register

Name	Bit(s)	Description	Read/ Write	Reset State
VA	29:12	The base address for the exception vectors, adjustable to a resolution of 4Kbytes. See *the exception entry points table* for how that moves all the exception entry points. The top two bits of this register must be `10` to make sure the exception vector ends up in the "kseg0" region, conventionally used for OS code. By setting *EBase* on any CPU to a unique value, that CPU can have its own unique exception handlers. Write this field only when *Status*$_{BEV}$ is set so that any exception will be handled through the ROM entry points (otherwise you would be changing the exception address under your own feet, and the results of that are undefined).	R/W	0x0000.0
CPUNum	9:0	This is just a single "CPU number" field (set by the core interface bus *SI_CPUNum*, which the SoC designer will tie to some suitable value).	R	Externally Set

7.2.21 Config (CP0 Register 16, Select 0): Legacy Configuration Register

The main role of the (several) configuration registers is to be a read-only repository of information about the core's resources, encoded so as to be useful to operating system initialization code.

But this original *Config* register acquired some writable fields. Typically, these select the sort of options you'd write once in initialization software and then never touch again.

Figure 7.22 Config Register Format

31	30 28	27 25	24	23	22	21	20 19	18	17	16	15	14 13	12 10	9 7	6 4	3	2 0
M	K23	KU	ISP	DSP	UDI	SB	0	MM	0	BM	BE	AT	AR	MT	0	VI	K0

Table 7.26 Field Descriptions for Config Register

Name	Bit(s)	Description	Read/ Write	Reset State
M	31	Reads 1 if *Config1* is available.	R	1
K23	30:28	If your CPU uses fixed mapping instead of having a TLB, you set the cacheability attributes of chunks of the memory map by writing these fields. They're encoded like *EntryLo0-1*$_C$. If you have a TLB, these fields are unused (but please write only zero to them). *Config*$_{K23}$ is for program addresses `0xC000.0000-0xFFFF.FFFF` (the "kseg2" and "kseg3" areas), while *Config*$_{KU}$ is for program addresses `0x0000.0000-0x7FFF.FFFF` (the "kuseg" area) From reset, both are "uncached" (code 2).	FMT: R/W TLB: R	FMT:2 TLB:0
KU	27:25		FMT: R/W TLB: R	FMT:2 TLB:0
ISP	24	Reads 1 if I-side scratchpad *(*ISPRAM*)* is fitted.	R	Preset
DSP	23	Reads 1 if D-side scratchpad *(*SPRAM*)* is fitted. (Don't confuse this with the MIPS DSP ASE, whose presence is indicated by *Config3*$_{DSPP}$.)	R	Preset
UDI	22	Reads 1 if your core implements user-defined "CorExtend" instructions.	R	Preset
SB	21	Read-only "SimpleBE" bus mode indicator, which reflects the core input signal *SI_SimpleBE*. If set, means that this core will only do simple partial-word transfers on its OCP interface; that is, the only partial-word transfers will be byte, aligned half-word, and aligned word. If zero, it may generate partial-word transfers with an arbitrary set of bytes enabled (which some memory controllers may not like).	R	Externally Set
MM	18	Writable: set 1 if you want writes resulting from separate store instructions in write-through mode merged into a single (possibly burst) transaction at the interface. This doesn't affect cache writebacks (which are always whole blocks together) or uncached writes (which are never merged).	R/W	1
BM	16	Read-only. Indicates whether your bus uses sequential or sub-block burst order; set by the core input signal *SI_SBlock* signal to match your system controller.	R	Externally Set
BE	15	Reads 1 for big-endian, 0 for little-endian, as selected by the core input *SI_Endian*.	R	Externally Set

Table 7.26 Field Descriptions for Config Register (Continued)

Name	Bit(s)	Description	Read/ Write	Reset State
AT	14:13	Reads 0 for MIPS32. 0 MIPS32 1 MIPS64 instruction set but MIPS32 address map 2 MIPS64 instruction set with full address map	R	0
AR	12:10	Reads 2 to reflect Release 2 of the MIPS32 architecture. Zero is for the original release.	R	1
MT	9:7	MMU type: 0 None 1 MIPS32/64 compliant TLB 2 "BAT" type 3 MIPS-standard fixed mapping All MIPS Technologies cores are type 1 or 3, as selected by your SoC builder.	R	Preset
VI	3	Reads 0 to indicate L1 I-cache is physically tagged. It would read 1 if the L1 I-cache were virtual (both indexed and tagged using virtual address).	R	0
K0	2:0	Is the fixed kseg0 region cached or uncached? And if cached, how exactly does it behave — this field is encoded just like the "cache coherency attribute" field of a TLB entry, as it shows up in the *EntryLo0-1*$_C$ field. The field powers up set to "uncacheable" (2). It’s very unusual to want to have kseg0 uncacheable, so this needs setting up in power-on software.	R/W	2

7.2.22 Config1-2 (CP0 Register 16, Select 1-2): MIPS32/64 Configuration Registers

These two registers tell you the size of the TLB, and the size and organization of L1, L2, and L3 caches (a zero "line size" is used to indicate a cache which isn’t there).

Config2 also has fields which tell you about the presence of some extensions to the base MIPS32 architecture that are implemented on this core.

7.2.22.1 Config1

This register displays the size and configuration of the TLB and primary caches, and the availability of some optional CPU features.

Figure 7.23 Config1 Register Format

31	30 25	24 22	21 19	18 16	15 13	12 10	9 7	6	5	4	3	2	1	0
M	MMUSize	IS	IL	IA	DS	DL	DA	C2	MD	PC	WR	CA	EP	FP

Table 7.27 Field Descriptions for Config1 Register

Name	Bit(s)	Description	Read/ Write	Reset State
M	31	Continuation bit, set to 1 to indicate that *Config2* is implemented.	R	1
MMUSize	30:25	The size of the TLB array (the array has MMUSize +1 entries).	R	Preset
IS	24:22	Fields for the L1 I-cache. All caches have the same triplet of fields, which report: S: Number of sets per way. Calculate as: $64 \text{ x } 2^S$ L: Line size. Zero means no cache at all, otherwise calculate as: 2×2^L A: Associativity/number of ways. Calculate as A + 1 So if (IS, IL, IA) is (2,4,3), you have 256 sets/way, 32 bytes per line, and 4-way set associativity, which is a 32Kbyte cache. 74K family cores always have 32-byte cache lines. The L1 caches are 4-way set associative and are 16KB, 32KB, or 64KB.	R	Preset
IL	21:19		R	4
IA	18:16		R	3
DS	15:13	For L1 D-cache: same encoding as $Config1_{IS,IL,IA}$.	R	Preset
DL	12:10		R	Preset
DA	9:7		R	3
C2	6	the absence of a coprocessor 2 (that would be a customer-designed coprocessor).	R	0
MD	5	0 to indicate that the MDMX ASE is not implemented in the floating point unit of the 74K core	R	0
PC	4	There is at least one performance counter implemented, see *PerfCnt0-3*.	R	1
WR	3	Reads 1 because the 74K core always has watchpoint registers, see *WatchLo0-3*/*WatchHi0-3*.	R	1
CA	2	Reads 1 because the MIPS16e compressed-code instruction set is available (as it is on most MIPS Technologies cores).	R	1
EP	1	Reads 1 because an *EJTAG debug unit* is always provided on MIPS Technologies cores.	R	1
FP	0	A floating point unit is attached.	R	Preset

7.2.22.2 Config2

Figure 7.24 Config2 Register Format

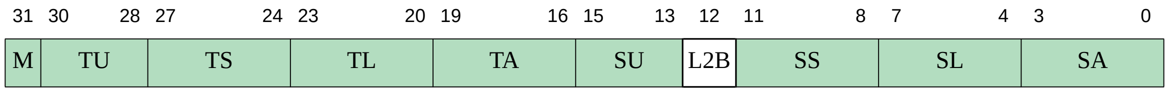


Table 7.28 Field Descriptions for Config2 Register

Name	Bit(s)	Description	Read/ Write	Reset State
M	31	Continuation bit, 1 if *Config3* is implemented.	R	1
TU	30:28	Reserved for extra control/status bits for an L3 cache, if fitted.	R	0
TS	27:24	If *Config2*$_{TL}$ is non-zero, your CPU has an L3 (tertiary) cache. Its size and shape are encoded just like *Config1*$_{IS,IL,IA}$ which see above. But no 74K family core is equipped for an L3 cache.	R	0
TL	23:20		R	0
TA	19:16		R	0
SU	15:13	Reserved for more secondary cache control/status bits, when required. Not used on the 74K family cores.	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 assertion its L2_Bypassed output pin. The value of L2_Bypassed is returned when L2B is read. When this bit is set through a write operation, a subsequent read of this bit will not indicate a 1, until the L2 has asserted the signal L2_Bypassed indicating that it has been bypassed.	R/W	0
SS	11:8	If *Config2*$_{SL}$ is non-zero, your CPU has an external L2 (secondary) cache. It's size and shape are encoded just like *Config1*$_{IS,IL,IA}$ above.	R	Preset
SL	7:4		R	Preset
SA	3:0		R	Preset

7.2.23 Config3 (CP0 Register 16, Select 3): Configuration register showing ASEs

Config3 provides information about the presence of optional extensions to the base MIPS32 architecture in addition to those specified in *Config2*.

Figure 7.25 Config3 Register Forma

31	30 14	13	12	11	10	9 7	6	5	4	3	2	1	0
M	0	ULRI	0	DSP2P	DSPP	0	VEIC	VInt	SP	0	MT	SM	TL

Table 7.29 Field Descriptions for Config3 Register

Name	Bit(s)	Description	Read/ Write	Reset State
M	31	Continuation bit, zero because there is no *Config4*.	R	0
ULRI	13	Reads 1 to indicate that the *UserLocal* Register is implemented	R	1

Table 7.29 Field Descriptions for Config3 Register (Continued)

Name	Bit(s)	Description	Read/ Write	Reset State
DSP2P	11	Reads 1 to indicate that Revision 2 of the MIPS DSP extension is implemented	R	1
DSPP	10	Reads 1 to indicate that the *MIPS DSP* extension is implemented.	R	1
VEIC	6	Read-only bit from the core input signal *SI_EICPresent* which should be set in the SoC to alert software to the availability of an *EIC*-compatible interrupt controller.	R	Externally Set
VInt	5	Reads 1 to indicate the CPU can handle vectored interrupts.	R	1
SP	4	Reads 0 to indicate the CPU does not support small (1Kbyte) pages.	R	0
MT	2	Reads 0 to indicate the CPU does not include the MIPS MT (multithreading) extension.	R	0
SM	1	Reads 0 to indicate the CPU does not include the instructions of the "SmartMIPS" ASE.	R	0
TL	0	Reads1 to indicate instruction trace is supported.	R	0

7.2.24 Config7 (CP0 Register 16, Select 7): CPU-specific Configuration

These registers control machine-specific features of the 74K core. A few of them are for hardware interface adaptation, but most are for chip or system test only. They default into a "safe" value, and most software—even bootstrap software—can and should ignore these registers completely,

7.2.24.1 Config7

CPU-specific one-time setup and basic information fields. The fields are described in Table 7.30. The remaining fields default to zero and are uncommonly set. It is therefore always safe *not* to write *Config7*. Some of these bits are for diagnostics and experimentation only:

Figure 7.26 Config7 Register Format

31	30	29	28	27	26	25	24	23	22	21	20 19	18	17	16	15 13	12 11	10 9	8	7	6	5	4	3	2	1	0
WII	FPFS	IHB	FPR1	SEHB	CP2IO	IAGN	IALU	DGHR	SG	SUI	0	HCI	FPR0	AR	0	PREF	0	ES	0	CP1IO	0	ULB	BP	RPS	BHT	SL

Table 7.30 Field Descriptions for Config7 Register

Name	Bit(s)	Description	Read/ Write	Reset State
WII	31	Wait IE Ignore. When this bit is set, an interrupt will unblock a **`wait`** instruction even if $Status_{IE}$ is preventing the interrupt from being taken. If *WII* reads 0, the 74K family of cores, will remain in the wait condition forever if entered with interrupts disabled. If set to 1,it allows OS code to avoid tricky race conditions.	R	1

Table 7.30 Field Descriptions for Config7 Register (Continued)

Name	Bit(s)	Description	Read/ Write	Reset State
FPFS	30	Fast Prepare for Store. When this bit is set **`pref 31`** will behave as specified, i.e. the prefetch instruction will only validate the data tag but not write 0's into the data cache. By default, this bit will be 0 and **`pref 31`** will behave like **`pref 30`**. This means that **`pref 31`** will validate the data tag and write 0's into the data cache array for the specified line	R/W	0
IHB	29	If IHB=1, the following behavior will be true: • When the core sees any explicit/implicit **`mtc0(cache,ll,mtc0, tlbop, eret, deret, sync`**`-in-debug-mode`**`, di, ei`**) followed by any implicit **`mfc0 (ehb, mfc0, eret, deret, di, ei)`**, the pipeline will behave as if an **`ehb`** is introduced implicitly prior to executing the **`mfc0`**. This ensures all state modification by **`mtc0`** is completely seen by **`mfc0`**. • Any **`jalr r31, jr r31`** instruction seen by the core when CP0 is usable (i.e CU0=1 or Kernel or Debug mode as defined in the PRA) will automagically treat those instructions as **`jalr.hb`** and **`jr.hb`**. If IHB=0 , the following behavior will be true: • Programmer is responsible for resolving hazards and put **`ehb`** or .**`hb`** where appropriate. Prior cores may have used some number of **`nops`** or **`ssnops`** to ensure that the effect of a CP0 modifying instruction is seen by a CP0 read instruction. 74K cannot guarantee such behavior with a small number of **`nops/ssnops`**. Per Release2, the programmer is expected to put in an explicit **`ehb`** or **`.hb`** where needed. If there is reason to believe that the programmer has not done this then this bit can be enabled to get correct operation.	R/W	0
SEHB	27	"Slow EHB": experimental mode to accelerate CP0 sequences using **`ehb`** If this bit is set, **`ehb`** will block issue of instructions from the instruction buffer until all older instructions have graduated and the pipe is empty. By default, **`ehb`** will block issue of instructions from the instruction buffer only if there are pending explicit CP0-modifying instructions in the pipe.	R/W	0
CP2IO	26	*CP1IO* Reserved for future use.	R/W	0
CP1IO	6	By default data sent from the core to a coprocessor block may be sent in an order reflecting the internal pipeline execution sequence. Set this bit to arrange that data will be sent only in instruction order to the FPU	R/W	0
IAGN	25	Selective control of out-of-order behavior: issue ALU-side or load/store-side instructions (respectively) in program order.	R/W	0
IALU	24		R/W	0
DGHR	23	Disables the use of any global history in the branch predictor.	R/W	0
SG	22	Set 1 to allow only one instruction to graduate per cycle. This has a negative impact on performance and should only be used for test purposes.	R/W	0

Table 7.30 Field Descriptions for Config7 Register (Continued)

Name	Bit(s)	Description	Read/ Write	Reset State
SUI	21	Strict Uncached Instruction (SUI) policy control. Run uncached instruction strictly in order and (as far as possible) unpipelined. This will be quite slow (the policy itself will introduce a 15-cycle bubble between each instruction), but you'll hardly notice because running uncached is already so slow. Only the branch-delay-slot instruction of a branch is fetched without this bubble. The advantage is that the CPU will not wander off speculatively fetching unwanted instructions from a (perhaps slow) boot memory.	R/W	0
HCI	18	Read-only field which is always zero on 74K family cores. It reads 1 for some software-simulated CPUs, to indicate that the software-modelled cache does not require initialization. Most software should ignore this bit.	R	0
FPR1, FPR0	28,17	Read-only fields. Indicates frequency of the core relative to FPU. • 2'b00: Core:FPU = 1:1 • 2'b01: Core FPU = 2:1 • 2'b10: Core:FPU = 3:2 • 2'b11: Reserved	R	Based on Hardware Present
AR	16	Read-only field, indicating that the D-cache is configured to avoid *cache aliases*. All the remaining fields are read/write, and control various functions. Only one of them is likely to find real system use:	R	Based on Hardware Present
PREF	12:11	These two bits control the extent of prefetching of Instructions into the Instruction Cache as indicated. • 2'b00: Prefetch 0 cache lines on an Icache miss in addition to fetching the missing cache line. i.e. Disable I-cache prefetching. • 2'b01: Prefetch 1 cache line (sequential next line) on an I-cache miss in addition to fetching the missing cache line. • 2'b10: Reserved • 2'b11: Prefetch 2 cache lines (sequential next 2 lines) on an Icache miss in addition to fetching the missing cache line.	R/W	01
ES	8	Defaults to zero. If set, the **`sync`** instruction will be signalled on the core's OCP interface as an "ordering barrier" transaction. The transaction is an extension to the OCP standards, and system controllers which don't support it will typically under-decode it as a read from the boot ROM area. But that's going to be quite slow: so set this bit only if your system understands the synchronizing transaction.	R/W	0
ULB	4	Set 1 to make all uncached loads blocking (a program usually only blocks when it uses the data which is loaded). You want to do this only when nothing else will work...	R/W	0
BP	3	When set, no branch prediction is done, and all branches and jump stall as above.	R/W	0
RPS	2	When set, the return address branch predictor is disabled, so **`jr $31`** is treated just like any other jump register. Instruction fetch stalls after the branch delay slot, until the jump instruction reaches the "EC" stage in the pipeline and can provide the right address.	R/W	0
BHT	1	When set, the branch history table is disabled and all branches are predicted taken. This bit is don't care if *Config7*$_{BP}$ is set.	R/W	0

Table 7.30 Field Descriptions for Config7 Register (Continued)

Name	Bit(s)	Description	Read/ Write	Reset State
SL	0	When set, disables non-blocking loads. Normally the 74K core will keep running after a load instruction even if it misses in the D-cache, until the data is used. With this disable bit set, the CPU will stall on any load D-cache miss.	R/W	0

7.2.25 WatchLo0-3 (CP0 Register 18, Select 0-3): Watchpoint Address and Qualifiers

Used in conjunction with *WatchHi0-3* respectively, each of these registers carries the virtual address and what-to-match fields for a CP0 watchpoint. *WatchLo0-1* are used for instruction side accesses and *WatchLo2-3* are used for data side accesses.

Figure 7.27 WatchLo Register Format

31 3	2	1	0
VAddr	I	R	W

Table 7.31 Field Descriptions for WatchLo0-3 Register

Name	Bit(s)	Description	Read/ Write	Reset State
VAddr	31:3	The address to match on, with a resolution of a doubleword.	R/W	Undefined
I	2	Accesses to match: I-fetches, Reads (loads), Writes (stores). *WatchLo0-1*$_R$ and *WatchLo0-1*$_W$ are fixed to zero, while *WatchLo2-3*$_I$ will be zero.	R/W	0
R	1		R/W	0
W	0		R/W	0

7.2.26 WatchHi0-3 (CP0 Register 19, Select 0-3): Watchpoint Control/Status

These registers provide the interface to a debug facility that causes an exception if an instruction or data access matches the address specified in the registers. Watch exceptions are not taken if the CPU is already in exception mode (that is if *Status*$_{EXL}$ or *Status*$_{ERL}$ is already set).

Watch events which trigger in exception mode are remembered, and result in a "deferred" exception, taken as soon as the CPU leaves exception mode.

WatchHi0-1 are used for instruction side accesses and *WatchHi2-3* are used for data side accesses.

This CP0 watchpoint system is independent of the EJTAG debug system (which provides more sophisticated hardware breakpoints).

The *WatchLo0-3* registers hold the address to match, while *WatchHi0-3* hold a bundle of control fields.

Figure 7.28 WatchHi Register Format

31	30	29 24	23 16	15 12	11 3	2	1	0
M	G	0	ASID	0	Mask	I	R	W

Table 7.32 Field Descriptions for WatchHi0-3 Register

Name	Bit(s)	Description	Read/ Write	Reset State
M	31	The *WatchHi0-3*$_{M}$ bit is set whenever there is one more watchpoint register pair to find; your software should use it (starting with *WatchHi0*) to figure out how many watchpoints there are. This field is set for *WatchHi0-2* and cleared on *WatchHii3.*	R	X
G	30	*WatchHi0-3*$_{ASID}$ matches addresses from a particular address space (the "ASID" is like that in TLB entries) — except that you can set *WatchHi0-3*$_{G}$ ("global") to match the address in any address space.	R/W	Undefined
ASID	23:16		R/W	Undefined
Mask	11:3	Implements address ranges. Set bits in *WatchHi0-3*$_{Mask}$ to mark corresponding *WatchLo0-3*$_{VAddr}$ address bits to be ignored when deciding whether this is a match.	R/W	Undefined
I	2	Read your *WatchHi0-3* after a watch exception, and these fields tell you what type of access (if anything) matched. Write a 1 to any of these bits in order to *clear* it (and therefore prevent the exception from immediately happening again). This behaviour is unusual among CP0 registers, but it is quite convenient: to clear a watchpoint of all the exception causes you've seen just read the value of *WatchHi0-3* and write it back again. *WatchHi0-1*$_{R}$ and *WatchHi0-1*$_{W}$ should always read 0 and *WatchHi2-3*$_{I}$ should always read 0	W1C	Undefined
R	1		W1C	Undefined
W	0		W1C	Undefined

7.2.27 Debug (CP0 Register 23, Select 0): EJTAG Debug Status/Control Register

Very little can be accessed outside of debug mode. In non-debug mode *Debug* may not be written at all, and only the *DM* bit and the *EJTAGver* field return valid data.

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 (a "nested exception"). Not all fields are valid in both circumstances: *Halt* and *Doze* are not defined after a nested exception; and the nested-exception-type field *DExcCode* is undefined from a debug exception. The fields are so numerous and the names are so long that the figure has to be divided into two parts.

Figure 7.29 Debug Register Format

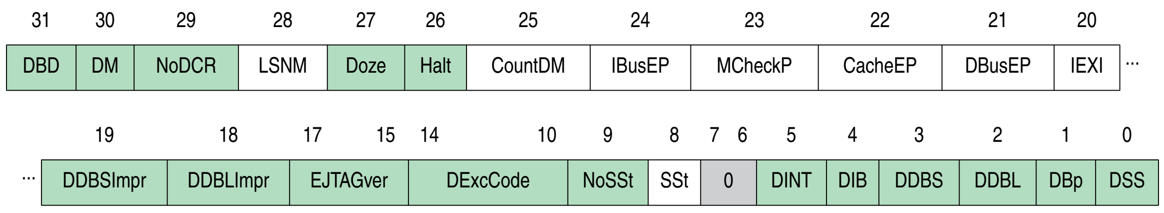


Table 7.33 Field Descriptions for Debug Register

Name	Bit(s)	Description	Read/ Write	Reset State
DBD	31	Exception happened in branch delay slot. When this happens *DEPC* will point to the branch instruction, which is usually the right place to restart.	R	Undefined
DM	30	Debug mode — set on any debug exception, cleared by `deret`.	R	0
NoDCR	29	Reads 0 to indicate the presence of a memory-mapped *DCR* register.	R	0
LSNM	28	Set this to 1 if you want debug-mode accesses to "dseg" addresses to be just sent to system memory. This makes most of the EJTAG unit's control system unavailable, so will probably only be done around a particular load/store.	R/W	0
Doze	27	Before the debug exception, CPU was in some kind of reduced power mode.	R	Undefined
Halt	26	Before the debug exception, the CPU was stopped — probably asleep after a `wait` instruction.	R	Undefined
CountDM	25	1 if and only if the count register continues to run in debug mode. Writable for the 74K core, so you get to choose. On other implementations it may be read-only, just informing you what the CPU does.	R/W	1
IBusEP	24	These "pending exception" flags remember exception events caused by instructions run in debug mode, but which have not happened yet because they are imprecise and $Debug_{IEXI}$ is set. Note that you can write them to 1 to any of these at any time, so they survive writes to the whole *Debug* register: but a write of zero to a field is ignored. They remain set until $Debug_{IEXI}$ is cleared explicitly, or implicitly by a `deret`. If it's the `deret` which clears such a bit, the exception is taken and the pending bit cleared: *IBusEP* would remember a bus error on instruction fetch. This exception is precise on the 74K core, so this can't happen and the field is always zero. *MCheckP* machine check condition (usually an illegal TLB update). As above, the machine check is always precise on the 74K core, so this is always zero. *CacheEP* remembers a cache parity error. *DBusEP* remembers a bus error on a data access.	R/W	0
MCheckP	23		R/W	0
CacheEP	22		R/W	0
DBusEP	21		R/W	0
IEXI	20	Set to 1 to defer imprecise exceptions. Set by default on entry to debug mode, cleared on exit, but writable. The deferred exception will come back when and if this bit is cleared: until then you can see that it happened by looking at the "pending" bits $Debug_{IBusEP, MCheckP, CacheEP, DBusEP}$	R/W	0
DDBSImpr	19	Imprecise store breakpoint. *DEPC* probably points to an instruction some time later in sequence than the store which triggered the breakpoint. The debugger or user (or both) have to cope as best they can.	R	0
DDBLImpr	18	Imprecise load breakpoint. (See note on imprecise store breakpoint, above).	R	0

Table 7.33 Field Descriptions for Debug Register (Continued)

Name	Bit(s)	Description	Read/ Write	Reset State
EJTAGver	17:15	Read-only — tells you which revision of the specification this implementation conforms to. On the 74K core it reads 3 for version 3.1. The full set of legal values are: 0 Version 2.0 and earlier 1 Version 2.5 2 Version 2.6 3 Version 3.1	R	3
DExcCode	14:10	Cause of any non-debug exception you just handled from within debug mode — following first entry to debug mode, this field is undefined. The value will be one of those defined for *Cause*$_{ExcCode}$	R	Undefined
NoSSt	9	Read-only — reads 0 because single-step is implemented (it always is on MIPS Technologies cores).	R	0
SSt	8	Set 1 to enable single-step.	R/W	0
DINT	5	Debug interrupt (from EJTAG pin).	R	Undefined
DIB	4	Instruction breakpoint.	R	Undefined
DDBS	3	Precise store breakpoint.	R	Undefined
DDBL	2	Precise load breakpoint.	R	Undefined
DBp	1	Any sort of debug breakpoint.	R	Undefined
DSS	0	Single-step exception.	R	Undefined

7.2.28 Trace Control Register (CP0 Register 23, Select 1)

The *TraceControl* register configuration is shown below.

Figure 7.30 TraceControl Register Format

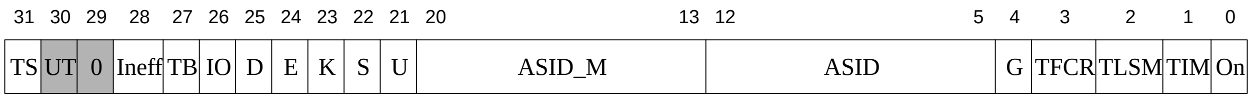

31 30 29 28 27 26 25 24 23 22 21 20 13 12 5 4 3 2 1 0
TS UT 0 Ineff TB IO D E K S U ASID_M ASID G TFCR TLSM TIM On

Table 7.34 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

Table 7.34 TraceControl Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bits			
UT	30	This bit is deprecated since there are now two explicit trace registers *UserTraceData1* and *UserTraceData2*. Previously this bit indicated 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.	0	Undefined
0	29	Reserved for future use; Must be written as zero; returns zero on read.	0	0
Ineff	28	When set to one, core-specific inefficiency tracing is enabled, and core-specific trace information is included in the trace stream. The inefficiency code replaces an "NI" and is interpreted in the trace stream with an expanded inscomp. The inscomp is expanded from 3b to 4b for all trace formats.	R/W	0
TB	27	Trace All Branch. When set to 1, this tells the processor to trace the PC value for all branches taken, not just the ones whose branch target address is statically unpredictable.	R/W	Undefined
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 also be one, and either the *G* bit must be one, or the current process ASID must match the *ASID* field in this register. When set to zero, trace is disabled in Debug Mode, regardless of the setting other bits.	R/W	Undefined
E	24	When set to one, 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. When set to zero, trace is disabled in Exception Mode, regardless of the setting of other bits.	R/W	Undefined
K	23	When set to one, 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. When set to zero, trace is disabled in Kernel Mode, regardless of the setting 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. When set to zero, trace is disabled in Supervisor Mode, regardless of other bits. If the processor does not implement Supervisor Mode, this bit is ignored on write and returns zero on read.	R/W	Undefined

Table 7.34 TraceControl Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bits			
U	21	When set to one, 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. When set to zero, trace is disabled in User Mode, regardless of the setting 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. If the processor does not implement the standard TLB-based MMU, this field is ignored on writes and returns zero on reads.	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. If the processor does not implement the standard TLB-based MMU, this field is ignored on writes and returns zero on reads.	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. If the processor does not implement the standard TLB-based MMU, this field is ignored on writes and returns 1 on reads. This causes all match equations to work correctly in the absence of an ASID.	R/W	Undefined
TFCR	3	When set, indicates to the PDtrace interface that the optional Fcr bit must be traced in the appropriate trace formats. If PC tracing is disabled, the full PC of the function call (or return) instruction must also be traced. Note that function call/return information is only traced if tracing is actually enabled for the current mode.	R/W	Undefined
TLSM	2	When set, this indicates to the PDtrace interface that information about data cache misses should be traced. If PC, load/store address, and data tracing are disabled (see the TraceControl2Mode field), the full PC and load/store address are traced for data cache misses. If load/store data tracing is enabled, the LSm bit must be traced in the appropriate trace format. Note that data cache miss information is only traced if tracing is actually enabled for the current mode.	R/W	Undefined
TIM	1	When set, this indicates to the PDtrace interface that the optional Im bit must be traced in the appropriate trace formats. If PC tracing is disabled, the full PC of the instruction that missed in the I-cache must be traced. Note that instruction cache miss information is only traced if tracing is actually enabled in the current mode.	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

7.2.29 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 7.31 TraceControl2 Register Format

31 30	29	28 21	20	19 12	11 7	6 5	4	3	2 1 0
SyPExt	CPUIdV	CPUId	TCV	TCNum	Mode	ValidModes	TBI	TBU	SyP

Table 7.35 TraceControl2 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
SyPExt	31:30	Extension to the *SyP* (sync period) field for implementations that need higher numbes of cycles between synchronization events. The value of *SyP* is extended by assuming that these two bits are juxtaposed to the left of the three bits of *SyP* (*SypExt*$_{SyP}$). When only *SyP* was used to specify the synchronization period, the value was 2x, where x was computed from *SyP* by adding 5 to the actual value represented by the bits. A similar formula is applied to the 5 bits just obtained by the juxtaposition of *SyPExt* and *SyP*. Sync period values greater than 2^{31} are UNPREDICTABLE. That is all values greater than 11010 (26+5=31) are UNPREDICTABLE. With SyPExt bits, a sync period range of 25 to 2^{31} cycles can be obtained.	R/W	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	0
CPUId	28:21	This field specifies the number of the VPE to trace when *CPUIdV* is set.	R/W	0
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	0
TCNum	19:12	Specifies the TC to trace when *TCV* is set. The right-most bits only are used.	R/W	0

Table 7.35 TraceControl2 Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bits			
Mode	11:7	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 tracing of a specific tracing mode 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. The table shows what trace value is turned on: **Bit** / **Trace the Following** 7 — PC 8 — Load address 9 — Store address 10 — Load data 11 — Store data	R/W	Undefined
ValidModes	6:5	This field specifies the subset of tracing that is supported by the processor. **Encoding** / **Meaning** 00 — PC tracing only 01 — PC and load and store address tracing only 10 — PC, load and store address, and load and store data 11 — Reserved	R	Preset
TBI	4	This bit indicates how many trace buffers are implemented by the TCB, as follows: **Encoding** / **Meaning** 0 — Only one trace buffer is implemented, and the TBU bit of this register indicates which trace buffer is implemented 1 — 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.	R	Undefined
TBU	3	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. **Encoding** / **Meaning** 0 — Trace data is being sent to an on-chip trace buffer 1 — Trace Data is being sent to an off-chip trace buffer This bit is loaded from *TCBCONTROLB*$_{OfC}$.	R	Undefined

Table 7.35 TraceControl2 Register Field Descriptions (Continued)

Fields				
Name	**Bits**	**Description**	**Read / Write**	**Reset State**
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. **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} This field is loaded from *TCBCONTROLA*$_{SyP}$.	R	Undefined

7.2.30 User Trace Data1 Register (CP0 Register 23, Select 3) and User Trace Data2 Register (CP0 Register 24, Select 3)

A software write to any bits in the *UserTraceData1* register or *UserTraceData2* register will trigger a trace record to be written with a type indicator TU1 or TU2 respectively.

These register are only implemented if the MIPS Trace capability is present.

Figure 7.32 User Trace Data1 / User Trace Data2 Register Format

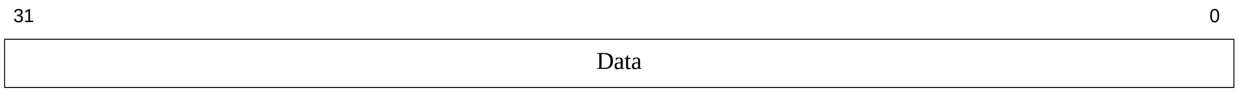


Table 7.36 UserTraceData1 / UserTraceData2 Register Field Descriptions

Fields				
Name	**Bits**	**Description**	**Read / Write**	**Reset State**
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

7.2.31 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. The Instruction 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 7.33 TraceIBPC Register Format

31 30	29	28	27 12	11 9	8 6	5 3	2 0
0	PCT	IE	0	IBPC$_3$	IBPC$_2$	IBPC$_1$	IBPC$_0$

Table 7.37 TraceIBPC Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	**Bits**			
0	31:30, 27:12	Reserved for future implementation.	R	0
PCT	29	Used to specify whether a performance counter trigger signal is generated when an EJTAG instruction breakpoint match occurs: **Encoding** / **Meaning** 0 — Disables performance counter trigger signal from instruction breakpoints 1 — Enables performance trigger signals from instruction breakpoints	R/W	0
IE	28	Used to specify whether the trigger signal from EJTAG instruction breakpoint should trigger tracing functions or not: **Encoding** / **Meaning** 0 — Disables trigger signals from instruction breakpoints 1 — Enables trigger signals from instruction breakpoints	R/W	0
IBPCn	3n+2:3n	The three bits are decoded to enable different tracing modes. Table 7.39 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.	R/W	0

7.2.32 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 7.34 TraceDBPC Register Format

31 30	29	28	27 6	5 3	2 0
0	PCT	DE	0	DBPC$_1$	DBPC$_0$

Table 7.38 TraceDBPC Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:30, 27:6	Reserved for future implementations.	R	0/1
PCT	29	Used to specify whether a performance counter trigger signal is generated when an EJTAG data breakpoint match occurs: **Encoding** / **Meaning** 0 — Disables performance counter trigger signal from data breakpoints 1 — Enables performance trigger signals from data breakpoints	R/W	0
DE	28	Used to specify whether the trigger signal from EJTAG data breakpoint should trigger tracing functions or not: **Encoding** / **Meaning** 0 — Disables trigger signals from data breakpoints 1 — Enables trigger signals from data breakpoints	R/W	0
DBPCn	3n+2:3n	The three bits are decoded to enable different tracing modes. Table 7.39 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.	R/W	0

Table 7.39 BreakPoint Control Modes: IBPC and DBPC

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 on, then there is no effect.
010	None	Reserved for future implementation.
100	Identical to trigger condition 000, and in addition, dump the full performance counter values into the trace stream	If tracing is currently on, dump the full values of all the implemented performance counters into the trace stream, and turn tracing off. If tracing is already off, then there is no effect.
101	Identical to trigger condition 001, and in addition, also dump the full performance counter values into the trace stream	Unconditionally start tracing if tracing was turned off. If tracing is already turned on, then there is no effect. In both cases, dump the full values of all the implemented performance counters into the trace stream.
110	Not used	Reserved for future implementations

7.2.33 DEPC (CP0 Register 24, Select 0): Restart Address from Last EJTAG Debug Exception

Points to the instruction to restart when you run an **`deret`** to leave debug mode. When *$Debug_{DBD}$* is set, it means that the "real" return address is in a branch delay slot, and *DEPC* points to the preceding branch.

Figure 7.35 DEPC Register Format

31 … 0
DEPC

Table 7.40 DEPC Register Formats

Field		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

7.2.34 Trace Control3 Register (CP0 Register 24, Select 2)

The *TraceControl3* register provides additional control and status information. Note that some fields in the *TraceControl3* 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 *TraceControl3* 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 7.36 TraceControl3 Register Format

31 … 14	13	12	11	10	9	8	7 … 2	1	0
0	PeCOvf	PeCFCR	PeCBP	PeCSync	PeCE	PeC	0	TRPAD	FDT

Table 7.41 TraceControl3 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:14	Reserved for future implementation.	R	0
PeCOvf	13	Trace performance counters when one of the performance counters overflows its count value. Enabled when set to 1.	R/W	0
PeCFCR	12	Trace performance counters on function call/return or on an exception handler entry. Enabled when set to 1.	R/W	0

Table 7.41 TraceControl3 Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bits			
PeCBP	11	Trace performance counters on hardware breakpoint match trigger. Enabled when set to 1.	R/W	0
PeCSync	10	Trace performance counters on synchronization counter expiration. Enabled when set to 1.	R/W	0
PeCE	9	Performance counter tracing enable. When set to 0, the tracing out of performance counter values as specified is disabled. To enable, this bit must be set to 1. This bit is used under software control. When trace is controlled by an external probe, this enabling is done via *TraceControl3*$_{PeCE}$.	R/W	0
PeC	8	Specifies whether or not Performance Control Tracing is implemented. This is an optional feature that may be omitted by implementation choice. Implemented when set to 1.	R/W	0
TrIDLE	2	Trace Unit Idle. This bit indicates if the trace hardware is currently idle (not processing any data). This can be useful when switching control of trace from hardware to software and vice versa. The bit is read-only and updated by the trace hardware.	R/W	0
TRPAD	1	Trace RAM Access Disable. Disables program software access to the on-chip trace RAM using load/store instructions. This bit is loaded from *TCBCONTROLB*$_{TRPAD}$.	R/W	0
FDT	0	Filtered Data Trace Mode Enable. When the bit is 0, this mode is disabled. When set to 1, this mode is enabled.	R/W	0

7.2.35 PerfCtl0-3 (CP0 Register 25, Select 0, 2, 4, 6): Performance Counter Control

Cores in the 74K family provide four "performance counters" — *PerfCnt0-3* — which can count your choice of a large range of events which might be of interest to a programmer.

If enabled by the corresponding *IE* flag, bit 31 of each counter can be wired to cause an interrupt. The OR of all the performance counter register interrupts becomes the core output *SI_PCI*, which is typically fed back again into an interrupt input, conventionally identified by $IntCtl_{IPPCI}$. However, systems using more sophisticated interrupt controllers may feed the performance counter interrupt into the interrupt controller, and then you'll have to read the interrupt controller documentation to figure out what happens to it.

Figure 7.37 PerfCtl0-3 Register Format

31	30 … 12	11 … 5	4	3	2	1	0
M	0	Event	IE	U	S	K	EXL

Table 7.42 Field Descriptions for PerfCtl0-3 Register

Name	Bit(s)	Description	Read/ Write	Reset State
M	31	Reads 1 if there is another *PerfCtl* register after this one. This field is set for *PerfCtl0-2* and cleared on *PerfCtl3.*	R	X
Event	11:5	Determines which event to count: available events are listed in Table 7.43, "Performance Counter Events and Codes".	R/W	Undefined
IE	4	Set to cause an interrupt when the counter "overflows" into its bit31. This can either be used to implement an extended count, or (by presetting the counter appropriately) to notify software after a certain number of events have occurred.	R/W	0
U	3	Count events in User mode, Supervisor mode, Kernel mode and Exception mode (i.e. when $Status_{EXL}$ is set) respectively. Set multiple bits to count in all cases.	R/W	Undefined
S	2		R/W	Undefined
K	1		R/W	Undefined
EXL	0		R/W	Undefined

Table 7.43 Performance Counter Events and Codes

Event No	Counter0/2	Counter1/3
0	Cycles	
1	Instructions graduated	
2	`jr $31` (return) instructions whose target is predicted	`jr $31` (return) predicted but guessed wrong
3	Cycles where no instruction is fetched because it has no "next address" candidate. This includes stalls due to register indirect jumps such as `jr,` stalls following a `wait` or `eret` and stalls dues to exceptions from instruction fetch	`jr $31` (return) instructions fetched and **not** predicted using RPS
4	ITLB accesses.	ITLB misses, which result in a JTLB access.

Table 7.43 Performance Counter Events and Codes (Continued)

Event No	Counter0/2	Counter1/3
5	Reserved	JTLB instruction access misses (will lead to an exception)
6	Instruction Cache accesses. 74K cores have a 128-bit connection to the I-cache and fetch 4 instructions every access. This counts **every** such access, including accesses for instructions which are eventually discarded. For example, following a branch which is incorrectly predicted, the 74K core will continue to fetch instructions, which will eventually get thrown away.	Instruction cache misses. Includes misses resulting from fetch-ahead and speculation.
7	Cycles where no instruction is fetched because we missed in the I-Cache.	Reserved
8	Cycles where no instruction is fetched because we are waiting for an I-fetch from uncached memory.	Reserved
9	Number of times the instruction fetch pipeline is flushed and replayed because the IFU buffers are full and unable to accept any instructions.	Reserved
10	Reserved	
11	Reserved	
12	Reserved	
13	Cycles where no instructions are brought into the IDU because the ALU instruction candidate pool is full.	Cycles where no instructions are brought into the IDU because the AGEN instruction candidate pool is full.
14	Cycles where no instructions can be added to the issue pool because we have run out of ALU completion buffers (CB's).	Cycles where no instructions can be added to the issue pool because we have run out of AGEN completion buffers (CB's).
15	Cycles where no instructions can be added to the issue pool, because we have used all the FIFO entries in the CLDQ, which keep track of data coming back from the FPU.	Cycles where no instructions can be added to the issue pool, because we have filled the "in order" FIFO used for coprocessor 1 instructions (IOIQ).
16	Cycles with no ALU-pipe issue: no instructions available.	Cycles with no AGEN-pipe issue: no instructions available.
17	Cycles with no ALU-pipe issue: we have instructions, but operands not ready.	Cycles with no AGEN-pipe issue: we have instructions, but operands not ready.
18	Cycles with no ALU-pipe issue: we have instructions, but some resource is unavailable. This includes: • Operands are not ready (same as event 17) • `div` in progress inhibits MDU instructions • CorExtend resource limitation.	Cycles with no AGEN-pipe issue: we have instructions, but some resource is unavailable. This includes: • Operands not ready (same as event 17) • Non-issued stores blocking ready to issue loads • Non-issued cacheops blocking ready to issue loads

Table 7.43 Performance Counter Events and Codes (Continued)

Event No	Counter0/2	Counter1/3
19	ALU-pipe bubble issued. This resulting empty pipe stage guarantees that some resource will be unused for a cycle, sometime soon. Used, for example, to guarantee an opportunity to write `mfc1` data into a CB. •	AGEN-pipe bubble issued. This resulting empty pipe stage guarantees that some resource will be unused for a cycle, sometime soon. Used, for example, to allow access to the data cache for refill or eviction. •
20	Cycles when only one instruction is issued.	Cycles when two instructions are issued (one ALU, one AGEN).
21	Cycles when instructions are issued out of order into the ALU pipe. i.e. instruction issued is not the oldest in the pool.	Cycles when instructions are issued out of order into the AGEN pipe. i.e. instruction issued is not the oldest in the pool.
22	Reserved	Reserved
23	Cacheable loads - Counts all accesses to the Dcache caused by load instructions. This count includes instructions that do not graduate.	All D-cache accesses (loads, stores, prefetch, cacheop etc.). This count includes instructions that do not graduate.
24	DCache writebacks	DCache misses. This count is per instruction at graduation and includes load, store, prefetch, `synci` and address based cacheops.
25	JTLB d-side (data side as opposed to instruction side) accesses	JTLB translation fails on d-side (data side as opposed to instruction side) accesses. This count includes instructions that do not graduate.
26	Load/store instruction redirects, which happen when the load/store follows too closely on a possibly matching cacheop.	The 74K core’s D-cache has an auxiliary virtual tag, used to pick the right line early. When (occasionally) the physical tag match and virtual tag match do not line up, it is treated as a cache miss - in processing the “miss” the virtual tag is corrected for future accesses. This event counts those bogus misses.
27	Reserved	Reserved •
28	L2 cache writebacks	L2 cache accesses
29	L2 cache misses	L2 cache miss cycles
30	Cycles Fill Store Buffer(FSB) <= 1/2 full	Cycles Fill Store Buffer(FSB) > 1/2 full
31	Cycles Load Data Queue (LDQ) <= 1/2 full	Cycles Load Data Queue(LDQ) > 1/2 full
32	Cycles Writeback Buffer(WBB) <= 1/2 full	Cycles Writeback Buffer(WBB) > 1/2 full
33	Reserved	Reserved
34	Reserved	Reserved
35	Replays following optimistic issue of instruction dependent on load which missed. Counted only when the dependent instruction graduates.	Floating Point Load instructions graduated.
36	`jr` (not `$31`) instructions graduated.	`jr $31` mispredicted at graduation
37	Integer Branch instructions graduated	Floating Point Branch instructions graduated

Table 7.43 Performance Counter Events and Codes (Continued)

Event No	Counter0/2	Counter1/3
38	Branch likely instructions graduated	Mispredicted Branch likely instructions graduated
39	Conditional branches graduated	Mispredicted Conditional branches graduated
40	Integer instructions graduated (includes `nop, ssnop, ehb` as well as all arithmetic, logic, shift and extract type operations).	Floating Point instructions graduated (but not counting Floating Point load/store)
41	Loads graduated (includes Floating Point)	Stores graduated (includes Floating Point). Of `sc` instructions, only successful ones are counted.
42	`j/jal` graduated	MIPS16e instructions graduated
43	no-ops graduated - included (`sll, nop, ssnop, ehb`).	integer multiply/divides graduated
44	DSP instructions graduated	ALU-DSP instructions graduated, result was saturated
45	DSP branch instructions graduated	MDU-DSP instructions graduated, result was saturated.
46	Uncached loads graduated.	Uncached stores graduated.
47	Reserved	Reserved
48	Reserved	Reserved
49	Reserved	Reserved
50	CP1 branches mispredicted.	Reserved
51	`sc` instructions graduated.	`sc` instructions failed.
52	`prefetch` instructions graduated.	`prefetch` instructions which did nothing, because they hit in the cache.
53	Cycles where no instructions graduated	Load misses graduated. Includes Floating Point Loads.
54	Cycles where one instruction graduated.	Cycles where two instructions graduated
55	Reserved	Reserved
56	Number of cycles no instructions graduated from the time the pipe was flushed because of a branch mispredict until the first new instruction graduates. This is an indicator graduation bandwidth loss due to mispredict.	Number of cycles no instructions graduated from the time the pipe was flushed because of a replay until the first new instruction graduates. This is an indicator graduation bandwidth loss due to replay. Often times this replay is a result of event 25 and therefore an indicator of bandwidth lost due to cache miss.
57	Reserved	Reserved
58	Exceptions taken	Replays initiated from graduation

Table 7.43 Performance Counter Events and Codes (Continued)

Event No	Counter0/2	Counter1/3
59	Implementation-specific CorExtend event. The integrator of this core may connect the core pin *UDI_perfcnt_event* to an event to be counted. This is intended for use with the CorExtend interface.	Implementation-specific system event. The integrator of this core may connect the core pin *SI_PCEvent* to an event to be counted.
60	Reserved	Reserved
61	Reserved	Reserved
62	Reserved	Implementation-specific DSPRAM event. The integrator of this core may connect the pin *SP_prf_c13_e62_xx* to the event to be counted
63	L2 single-bit errors corrected	Reserved

7.2.36 PerfCnt0-3 (CP0 Register 25, Select 1, 3, 5, 7): Performance Counters

General purpose event counters, which operate as directed by *PerfCtl0-3*.

Figure 7.38 Performance Counter Count Register

31 … 0
Counter

Table 7.44 Performance Counter Count Register Field Descriptions

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

31 … 0
Counter

7.2.37 ErrCtl (CP0 Register 26, Select 0): Software Parity Control and Test Modes for Cache RAM Arrays

Most of the fields of this register are for test software only. The MIPS32 Architecture defines this register as implementation-dependent, but most CPUs put the parity-enable control in the top bit. So running OS software is well advised to set this register to `0x8000.0000` to enable cache parity checking, or to zero to disable parity checking.

31	30	29	28	27	26	25	24	23	22	21	20	19	18 … 12	11 … 4	3 … 0
PE	PO	WST	SPR	PCO	ITC	LBE	WABE	L2P, L2EccEnable	PCD	DYT	SE	FE	0	PI	PD

Table 7.45 Field Descriptions for ErrCtl Register

Name	Bit(s)	Description	Read/ Write	Reset State
PE	31	1 to enable cache parity checking. Hard-wired to zero if parity isn't implemented.	R/W	0
PO	30	Parity Overwrite. Set 1 to set the parity bit regardless of parity computation, which is only for diagnostic/test purposes. After setting this bit you can use **`cache IndexStoreTag`** to set the cache data parity to the value currently in *PI* (for I-cache) or *PD* (for D-cache), while the tag parity is forcefully set from *ITagLo_P*/*DTagLo_P*.	R/W	0
WST	29	Write to 1 for test mode for **`cache IndexLoadTag`**/ **`cache IndexStoreTag`** instructions, which then read/write the cache's internal *way-selection RAM* instead of the cache tags.	R/W	0
SPR	28	Scratchpad RAM. When set, index-type cache instructions work on the *scratchpad/DSPRAM/ISPRAM,* if fitted.	R/W	0
PCO	27	Precode override. Used for diagnostic/test of the I-cache. When this bit is set, then the precode values in the ITagHi register are used instead of the hardware generated precode values. This applies to index store data cacheop operation.	R/W	0
ITC	26	Reserved	R/W	Undefined
LBE	25	Indicates whether a bus error (the last one, if there's been more than one) was triggered by a load or a write-allocate respectively. A write-allocate is where a cacheable write has missed in the cache, and the cache has read the line from memory. Where both a load and write-allocate are waiting on the same cache-line refill, both could be set. These bits are "sticky", remaining set until explicitly written zero.	R	Undefined
WABE	24		R	Undefined
L2P,L2Ecc Enable	23	L2 Present, L2EccEnable: Indicates whether ECC is enabled on the L2Cache if present. • 0: L2Presetnt & L2EccEnable = 0 • 1: L2Present & L2EccEnable = 1	R/W	0
DYT	21	Set 1 to arrange that **`cache`** load/store data operations work on the "dirty array" — the slice of cache memory which holds the "dirty"/ "stored-into" bits.	R/W	0
PCD	22	Precode Disable. When set, **`cache IndexStoreTag`** instructions do not update the corresponding precode field and precode parity in the instruction cache tag array.	R/W	0
SE	20	Indicates that a second cache error was detected before the first error was processed. This is an unrecoverable error. This bit is set when a cache error is detected while the *FE* bit is set. This bit is cleared on reset or when a cache error is detected with *FE* cleared.	R	0
FE	19	Indicates that this is the first cache error and therefore potentially recoverable. Error handling software should clear this bit when the error has been processed. This bit is cleared on reset. Refer to *SE* bit description for implications of this bit. Note that software can only write a 0 to this bit. A write value of 1 will not have any effect.	R/W	0
PI	11:4	Parity bits being read/written to caches (I- and D-cache respectively), when *PO* is set.	R/W	
PD	3:0		R/W	

7.2.38 CacheErr (CP0 Register 27, Select 0): Cache Parity Exception Status

Read-only register used to analyze the details of a parity error.

Figure 7.39 CacheErr Register Format

31	30	29	28	27 26	25	24	23	22	21 19	18	17	16 0
ER	EC	ED	ET	0	EB	EF	SP	EW	Way	DR		Index

Table 7.46 Field Descriptions for CacheErr Register

Name	Bit(s)	Description	Read/ Write	Reset State
ER	31	This bit reads 1 if the error was on a L1 data cache access and reads 0 other wise. For errors caused by L1 I-fetch or L2 or higher level cache accesses, this bit will read 0.	R	Undefined
EC	30	Reads 0 if L1 cache errors and 1 for higher-level caches.	R	Undefined
ED	29	Set for errors in data field and tag field respectively.	R	Undefined
ET	28		R	Undefined
EB/EM	25	• If *EC* equals 0 indicating an error in the L1 cache, this bit is *EB* indicating Error in Both caches. If data and instruction-fetch error are reported on the same instruction, it is unrecoverable. If so, the rest of the register reports on the instruction-fetch error. • If *EC* equals 1 indicating an error in the L2 or higher cache, this bit is *EM* indicating Error in Multiple locations.	R	Undefined
EF	24	Unrecoverable (fatal) error (other than the *EB* type above). Some parity errors can be fixed by invalidating the cache line and relying on good data from memory. But if this bit is set, all is lost... It's one of the following: 7.46.1 Dirty parity error in dirty victim 7.46.1 Line being displaced from cache ("victim") has a tag parity error, so we don't know whether to write it back, or whether the writeback location (which needs a correct tag) would be correct. 7.46.2 The victim's tag indicates it has been written by the CPU since it was obtained from memory (the line is "dirty" and needs a write-back), but it has a data parity error. 7.46.3 Writeback store miss and *CacheErr*$_{EW}$ error. 7.46.4 At least one more cache error happened concurrently with or after this one, but before we reached the relative safety of the cache error exception handler. 7.46.5 If *EC* equals 0 and a second L2 error happens when an earlier L2 error is pending.	R	Undefined
SP	23	Error affecting a *scratchpad RAM* access.	R	Undefined
EW	22	Parity error on way-selection RAM array.	R	Undefined
Way	21:19	• If *EC* equals 0, bit 19 is unused. Bits 21:20 indicate the way-number of the cache entry where the error occurred. It is not valid if a Scratchpad RAM error is detected (SP=1). • If *EC* equals 1,indicating an L2 or higher level cache error, Bits 21:19 indicate the way-number of the cache entry where the error occurred.	R	Undefined
DR	18	A 1 bit indicates that the reported error affected the cache line "dirty" bits. This bit is only meaningful in case of an L1 data cache access.	R	Undefined

Table 7.46 Field Descriptions for CacheErr Register (Continued)

Name	Bit(s)	Description	Read/ Write	Reset State
Index	16:0	The cache index (within the cache way) or Scratchpad RAM index of the entry where the error occurred. Unlike some other cores, the low bits are meaningful and the index is not aligned as if a byte address. The index is cache line aligned. The index-type **`cache`** instruction will need an "index" with the way bits glued on top of this cache-entry field; you know how to put that together, because the shape of the cache is defined in the *Config1-2* registers.	R	Undefined

7.2.39 ITagLo (CP0 Register 28, Select 0): Read/write Interface for Load/Store Tag Cacheops

These registers are a staging location for cache tag information being read/written with **`cache`** load-tag/store-tag operations.

The interpretation of this register changes depending on the setting s of $ErrCtl_{WST}$ and $ErrCtl_{SPR}$.

- Default cache interface mode ($ErrCtl_{WST}$ = 0, $ErrCtl_{SPR}$ = 0)
- Diagnostic "way select test mode" ($ErrCtl_{WST}$ = 1, $ErrCtl_{SPR}$ = 0)
- For scratchpad memory setup ($ErrCtl_{WST}$ = 0, $ErrCtl_{SPR}$ = 1)

See the diagrams below for a description.

7.2.39.1 ITagLo ($ErrCtl_{WST}$ = 0, $ErrCtl_{SPR}$ = 0)

In this mode, this register is a staging location for cache tag information being read/written with **`cache`** load-tag/store-tag operations — routinely used in cache initialization.

Figure 7.40 ITagLo Register Format ($ErrCtl_{WST}$ = 0, $ErrCtl_{SPR}$ = 0)

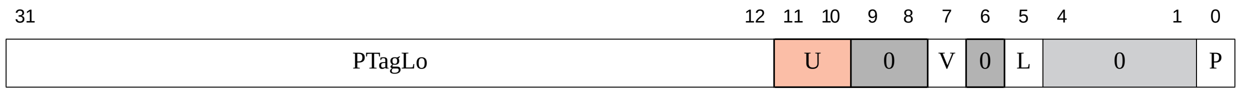


Table 7.47 Field Descriptions for ITagLo Register

Name	Bit(s)	Description	Read/ Write	Reset State
PTagLo	31:12	The cache address tag — a physical address because the 74K core's caches are physically tagged. It holds bits `31-12` of the physical address — the low 12 bits of the address are implied by the position of the data in the cache.	R/W	Undefined
V	7	1 if this cache entry is valid (set zero to initialize the cache).	R/W	Undefined

Table 7.47 Field Descriptions for ITagLo Register (Continued)

Name	Bit(s)	Description	Read/ Write	Reset State
L	5	1 to lock this cache entry, preventing it from being replaced by another line when there's a cache miss. Done when you have data so critical that it must be in the cache: it's quite costly, reducing the efficiency of the cache for memory data competing for space at this index.	R/W	Undefined
P	0	Parity bit over the cache tag entries (excluding the *D* bit).	R/W	Undefined

7.2.39.2 ITagLo-WST(ErrCtlWST = 1, ErrCtlSPR = 0)

The way-select RAM is an independent slice of the cache memory (distinct from the tag and data arrays). Test software can access either by **`cache`** load-tag/store-tag operations when $ErrCtl_{WST}$ is set. Then you get the data in these fields.

Figure 7.41 ITagLo Register Format ($ErrCtl_{WST}$ = 1, $ErrCtl_{SPR}$ = 0)

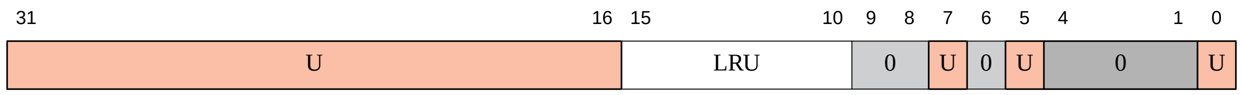


Table 7.48 Field Descriptions for ITagLo-WST Register

Name	Bit(s)	Description	Read/ Write	Reset State
LRU	15:10	When you read or write the tag in way select test mode (that is, with $ErrCtl_{WST}$ set) this field reads or writes the LRU ("least recently used") state bits, held in the way select RAM.	R/W	Undefined

7.2.39.3 ITagLo-WST($ErrCtl_{WST}$ = 0, $ErrCtl_{SPR}$ = 1)

In this mode, the ITagLo register becomes the interface to the instruction scratchpad RAM.

Figure 7.42 ITagLo Register Format ($ErrCtl_{WST}$ = 0, $ErrCtl_{SPR}$ = 1)

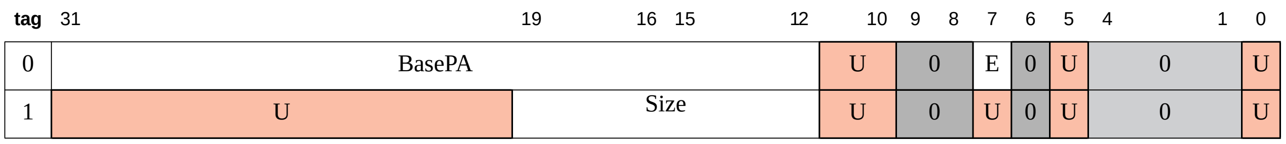


Table 7.49 Field Descriptions for ITagLo-SPR Register

Name	Bit(s)	Description	Read/ Write	Reset State
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

Table 7.49 Field Descriptions for ITagLo-SPR Register (Continued)

Name	Bit(s)	Description	Read/ Write	Reset State
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.	R/W	Undefined

7.2.40 IDataLo (CP0 Register 28, Select 1): Read/write Interface for I-cache Special Cacheops

Staging registers for special **`cache`** which load or store data from or to the cache line. Two registers (*IDataHi*, *IDataLo*) are needed because the 74K core loads I-side data at least 64-bits at a time.

Figure 7.43 IDataLo Register Format

31	0
DATA	

Table 7.50 IDataLo 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

7.2.41 DTagLo (CP0 Register 28, Select 2): Read/Write Interface for Load/Store Tag Cacheops

These registers are a staging location for cache tag information being read/written with **`cache`** load-tag/store-tag operations.

The interpretation of this register changes depending on the settings of *ErrCtl*$_{WST}$ and *ErrCtl*$_{SPR}$.

- Default cache interface mode (*ErrCtl*$_{WST}$ = 0, *ErrCtl*$_{DYT}$ = 0, *ErrCtl*$_{SPR}$)= 0)
- Diagnostic "way select test mode" (*ErrCtl*$_{WST}$ = 1, *ErrCtlDYT* = 0, *ErrCtl*$_{SPR}$) = 0)
- Diagnostic "dirty array test mode" (*ErrCtl*$_{WST}$ = 0, *ErrCtl*$_{DYT}$ = 1, *ErrCtl*$_{SPR}$) = 0)
- For scratchpad memory setup (*ErrCtl*$_{WST}$ = 0, *ErrCtl*$_{DYT}$ = 0, *ErrCtl*$_{SPR}$)= 1)

7.2.41.1 DTagLo (ErrCtl$_{WST}$ = 0, ErrCtl$_{DYT}$ = 0, ErrCtl$_{SPR}$ = 0)

In this mode, this register is a staging location for cache tag information being read/written with **`cache`** load-tag/store-tag operations — routinely used in cache initialization.

Figure 7.44 DTagLo Register Format (ErrCtl$_{WST}$ = 0, ErrCtl$_{DYT}$ = 0, ErrCtl$_{SPR}$ = 0)

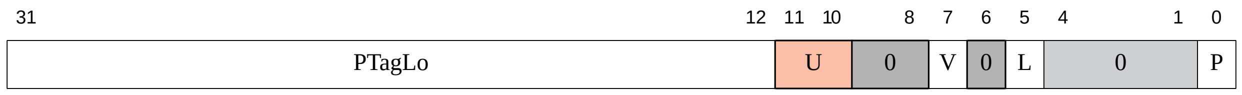


Table 7.51 Field Descriptions for DTagLo Register

Name	Bit(s)	Description	Read/ Write	Reset State
PTagLo	31:12	The cache address tag — a physical address because the 74K core's caches are physically tagged. It holds bits `31-12` of the physical address — the low 12 bits of the address are implied by the position of the data in the cache.	R/W	Undefined
U	11	This bit always gets the virtual address bit [11] of the tag if the index load tag cache instruction is executed.	R/W	Undefined
V	7	1 if this cache entry is valid (set zero to initialize the cache).	R/W	Undefined
L	5	1 to lock this cache entry, preventing it from being replaced by another line when there's a cache miss. Done when you have data so critical that it must be in the cache: it's quite costly, reducing the efficiency of the cache for memory data competing for space at this index.	R/W	Undefined
P	0	Parity bit over the cache tag entries (excluding the *D* bit).	R/W	Undefined

7.2.41.2 DTagLo-WST(ErrCtl$_{WST}$ = 1, ErrCtl$_{DYT}$ = 0, ErrCtl$_{SPR}$= 0)

The way-select RAM is an independent slice of the cache memory (distinct from the tag and data arrays). Test software can access either by **`cache`** load-tag/store-tag operations when *ErrCtl$_{WST}$* is set: then you get the data in these fields.

Figure 7.45 DTagLo Register Format (ErrCtl$_{WST}$ = 1, ErrCtl$_{DYT}$ = 0, ErrCtl$_{SPR}$ = 0)

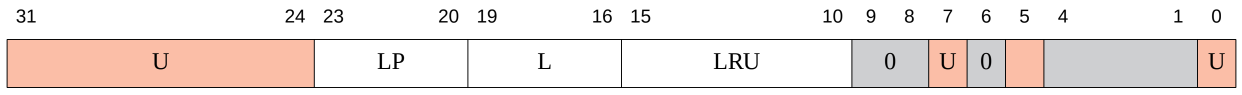


Table 7.52 Field Descriptions for DTagLo-WST Register

Name	Bit(s)	Description	Read/ Write	Reset State
LP	23:20	Cache-line locking control bits, held in the way select RAM, and parity over them.	R/W	Undefined
L	19:16		R/W	Undefined
LRU	15:10	When you read or write the tag in way select test mode (that is, with *ErrCtl$_{WST}$* set) this field reads or writes the LRU ("least recently used") state bits, held in the way select RAM.	R/W	Undefined

7.2.41.3 DTagLo-DYT(ErrCtl$_{WST}$ = 0, ErrCtl$_{DYT}$ = 1, ErrCtl$_{SPR}$) = 0)

The dirty RAM is another slice of the cache memory (distinct from the tag and data arrays). Test software ca access either by **`cache`** load-tag/store-tag operations when *ErrCtl$_{DYT}$* is set: then you get the data in these fields.

Figure 7.46 Field Descriptions for DTagLo-DYT Register

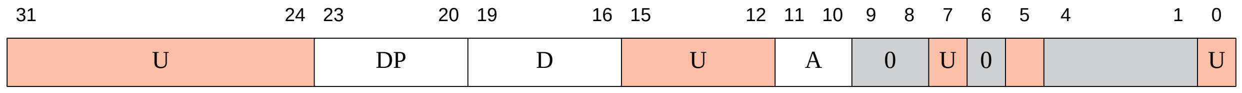


Table 7.53 Field Descriptions for DTagLo-DYT Register

Name	Bit(s)	Description	Read/ Write	Reset State
DP	23:20	Cache line "dirty" bits (and parity across them).	R/W	Undefined
D	19:16		R/W	Undefined
A	11:10	Cache line "alias" bits.	R/W	Undefined

7.2.41.4 DTagLo-SPT(ErrCtl$_{WST}$ = 0, ErrCtl$_{DYT}$ = 0, ErrCtl$_{SPR}$) = 1)

If your CPU has *scratchpad* RAM, you will need to initialize and manage it using **`cache`** load/store operations while *ErrCtl$_{SPR}$* is set. The tag load/store operations are used to read and write control registers: and then you see these fields.

Figure 7.47 DTagLo Register Format (ErrCtl$_{WST}$ = 0, ErrCtl$_{DYT}$ = 0, ErrCtl$_{SPR}$) = 1)

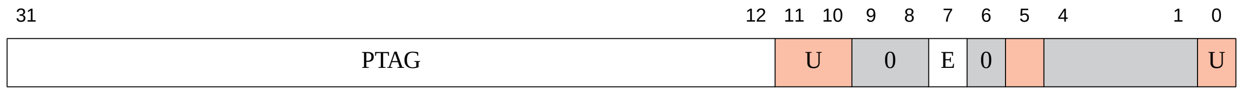


Table 7.54 Field Descriptions for DTagLo-SPT Register

Name	Bit(s)	Description	Read/ Write	Reset State
PTAG	31:12	Scratchpad control. Sets base address.	R/W	Undefined
E	7	Scratchpad control enable.	R/W	Undefined

7.2.42 DDataLo (CP0 Register 28, Select 3): Low-order Data Read/Write Interface for D-cache

On 74K family cores, test software can read or write cache data using a **`cache`** index load tag/index store data instruction. Which word of the cache line is transferred depends on the low address fed to the **`cache`** instruction.

Figure 7.48 DDataLo Register Format

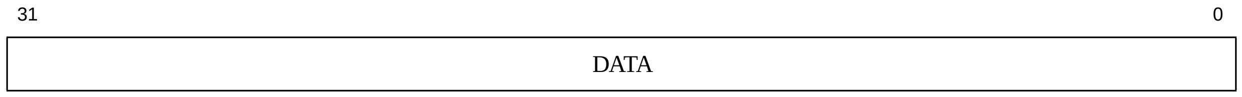


Table 7.55 DDataLo Register Field Description

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

7.2.43 L23TagLo (CP0 Register 28, Select 4): L2 and L3 Cacheop Tag Use

This register in the 74K core is implemented to support access to external L2 cache tags via **`cache`** instructions. The definition of the fields of this 32 bit register are defined by the SoC designer. Refer to the section on L2 Transactions in the document ““MIPS32® 74K™ Processor Core Family Integrator’s Guide, MD00499” for further information on using this register.

Figure 7.49 L23TagLo Register Format

31 … 0
DATA

7.2.44 L23DataLo (CP0 Register 28, Select 5): Low-order Data Read/Write Interface for L2 or L3 cache

On 74K family cores, test software can read or write cache data using a **`cache`** index load/store data instruction. Which word of the cache line is transferred depends on the low address fed to the **`cache`** instruction.

Figure 7.50 L23DataLo Register Format

31 … 0
DATA

Table 7.56 L23DataLo 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

7.2.45 ITagHi (CP0 Register 29, Select 0): I-cache Predecode Bits

This register represents the I-cache Predecode bits and is intended for diagnostic use only

Figure 7.51 ITagHi Register Format

31 … 25	24 … 18	17 … 11	10 … 4	3	2	1	0
PREC_67	PREC_45	PREC_23	PREC_01	P_67	P_45	P_23	P_01

Table 7.57 Field Descriptions for ITagHi Register

Name	Bit(s)	Description	Read/ Write	Reset State
PREC_67	31:25	74K family cores do some decoding of instructions when they're loaded into the I-cache, which helps speed instruction dispatch. When you use **`cache`** tag load/store instructions, you see that information here. The individual *PREC* fields hold precode information for pairs of adjacent instructions in the I-cache line, and the *P* fields hold parity over them.	R/W	Undefined
PREC_45	24:18		R/W	Undefined
PREC_23	17:11		R/W	Undefined
PREC_01	10:4		R/W	Undefined
P_67	3		R/W	Undefined
P_45	2		R/W	Undefined
P_23	1		R/W	Undefined
P_01	0		R/W	Undefined

7.2.46 IDataHi (CP0 Register 29, Select 1): High-order Data Read/write Interface for I-cache Special Cacheops

Staging registers for special **`cache`** which load or store data from or to the cache line. Two registers (*IDataHi*, *IDataLo*) are needed because the 74K core loads I-cache data at least 64-bits at a time.

Figure 7.52 IDataHi Register Format

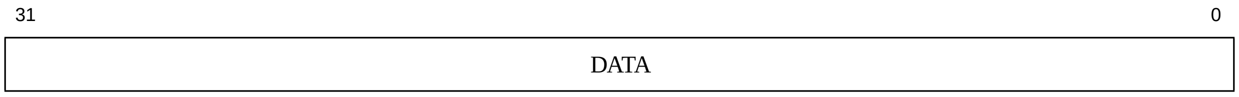


Table 7.58 IDataHi Register Field Description

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

7.2.47 DTagHi (CP0 Register 29, Select 2): D-cache Virtual Index (including ASID)

More cache tag bits for the 74K core's dual-tagged L1 D-cache. For diagnostics only.

Figure 7.53 DTagHi Register Format

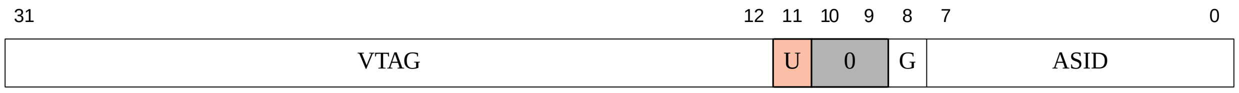


Table 7.59 Field Descriptions for DTagHi Register

Name	Bit(s)	Description	Read/ Write	Reset State
VTAG	31:12	74K family cores have a dual-tagged D-cache, combining a virtual tag for fast lookup with a physical tag to avoid *aliases*. Bit[11] always gets virtual address[11] of the tag when index load tag cache instruction is executed. These fields store the information required to match a virtual address: the virtual address itself, the ASID (tracking the "address space identifier" maintained in *EntryHi*$_{ASID}$) and a global ("*G*") bit which can be set to make it not necessary to match the ASID.	R/W	Undefined
U	11		R/W	Undefined
G	8		R/W	Undefined
ASID	7:0		R/W	Undefined

7.2.48 L23DataHi (CP0 Register 29, Select 5): High-order Data Read/Write Interface for L2 or L3 cache

On 74K family cores, test software can read or write cache data using a **`cache`** index load/store data instruction. Which word of the cache line is transferred depends on the low address fed to the **`cache`** instruction.

Figure 7.54 L23DataHi Register Format

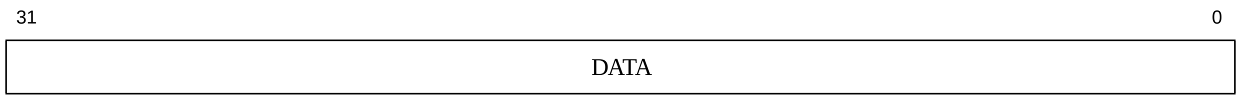


Table 7.60 L23DataHi Register Field Description

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

7.2.49 ErrorEPC (CP0 Register 30, Select 0): Restart Location from Reset or Cache Error Exception

This full 32-bit register is filled with the restart address on a cache error exception or any kind of CPU reset — in fact, any exception which sets *Status*$_{ERL}$ and leaves the CPU in "error mode".

Figure 7.55 ErrorEPC Register Format

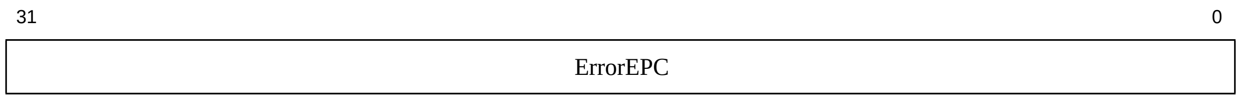


Table 7.61 ErrorEPC Register Field Description

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

7.2.50 DESAVE (CP0 Register 31, Select 0): Scratch Read/Write Register for EJTAG Debug Exception Handler

Software-only register, with no hardware effect. Provided because the debug exception handler can't use the *k0-1* GP registers, used by ordinary exception handlers to bootstrap themselves: but a debug handler can save a GPR into *DESAVE*, and then use that GPR register in code which saves everything else.

Figure 7.56 DeSave Register Format

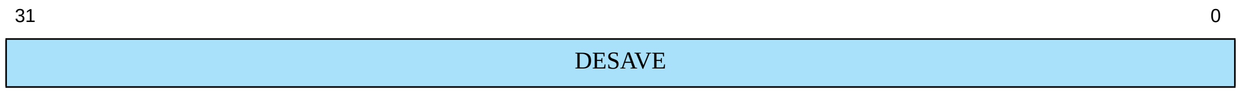


Table 7.62 DeSave Register Field Description

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
DESAVE	31:0	Debug exception save contents.	SO	Undefined

Chapter 8

Hardware and Software Initialization of the 74K™ Core

A 74K 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 74K 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:

- *Random* - cleared to maximum value on Reset (TLB/MMU only)
- *Wired* - cleared to 0 on Reset (TLB/MMU only)
- *StatusBEV* - set to 1 on Reset
- *StatusTS* - cleared to 0 on Reset
- *StatusNMI* - cleared to 0 on Reset
- *StatusERL* - set to 1 on Reset
- *StatusRP* - cleared to 0 on Reset
- *WatchLoI,R,W* - cleared to 0 on Reset
- *Config* fields related to static inputs - set to input value by Reset
- *ConfigK0* - set to 010 (uncached) on Reset
- *ConfigKU* - set to 010 (uncached) on Reset (FMT/MMU only)

- *ConfigK23* - set to 010 (uncached) on Reset (FMT/MMU only)
- *DebugDM* - cleared to 0 on Reset (unless EJTAGBOOT option is used to boot into Debug Mode, as described in Chapter 11, “EJTAG Debug Support in the 74K™ Core” on page 223.
- *DebugLSNM* - cleared to 0 on Reset
- *DebugIBusEP* - cleared to 0 on Reset
- *DebugDBusEP* - cleared to 0 on Reset
- *DebugIEXI* - cleared to 0 on Reset
- *DebugSSt* - 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 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 (for example, those defining the bus mode and cache size) 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 parts of the device, as described below.

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 74K 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 because the instruction cache initialization must 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 tnterrupts are used.
- *Compare*: Should be set to a known value if timer tnterrupts are used. Note that the write to *Compare* will also clear any pending timer interrupts, so *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.

Chapter 9

Caches of the 74K™ Core

This chapter describes the caches present in a 74K processor core. It contains the following sections:

9.1 Cache Configurations

A 74K 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 can be configured at build time to be 0, 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: data, tag, and way-select.

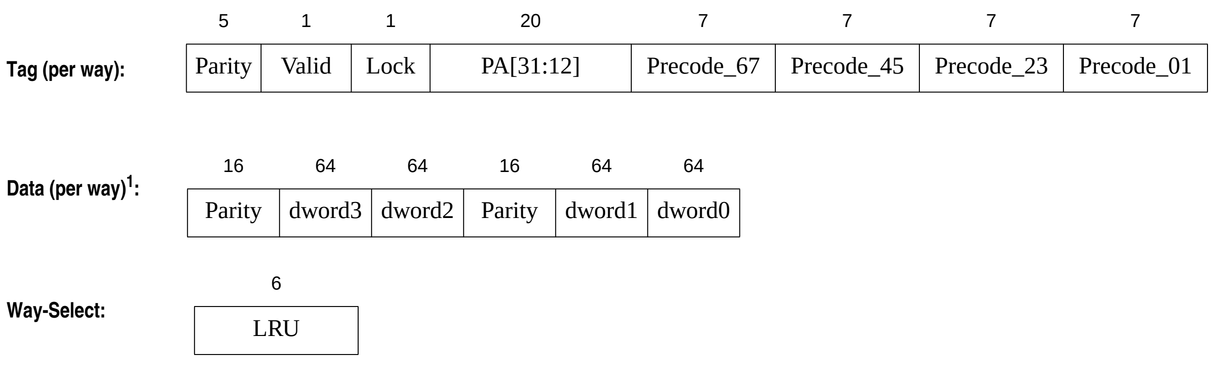
Table 9.1 Instruction Cache Attributes

Attribute	With Parity	Without Parity
Size	0, 16, 32, 64KB	
Line Size	32B	
Number of Cache Sets	128, 256, 512	
Associativity	4 way	
Replacement	LRU	
Cache Locking	per line	
Data Array		

Table 9.1 Instruction Cache Attributes (Continued)

Attribute	With Parity	Without Parity
Read Unit	144b x 4	128b x 4
Write Unit	144b	128b
Tag Array		
Read Unit	55b x 4	50b x 4
Write Unit	55b	50b
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 74K 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. Virtual aliasing comes into effect only for cache sizes that are larger than 16KB. The 16KB cache size does not suffer from virtual aliasing, because the way size equals the minimum page size.

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.

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 16 parity bits—one for each byte of the 128b data. The tag array has 5 parity bits for each tag—one for each of the 4 precode fields and one for the physical tag, lock, and valid bits. The LRU array does not have any parity.

9.3 Data Cache

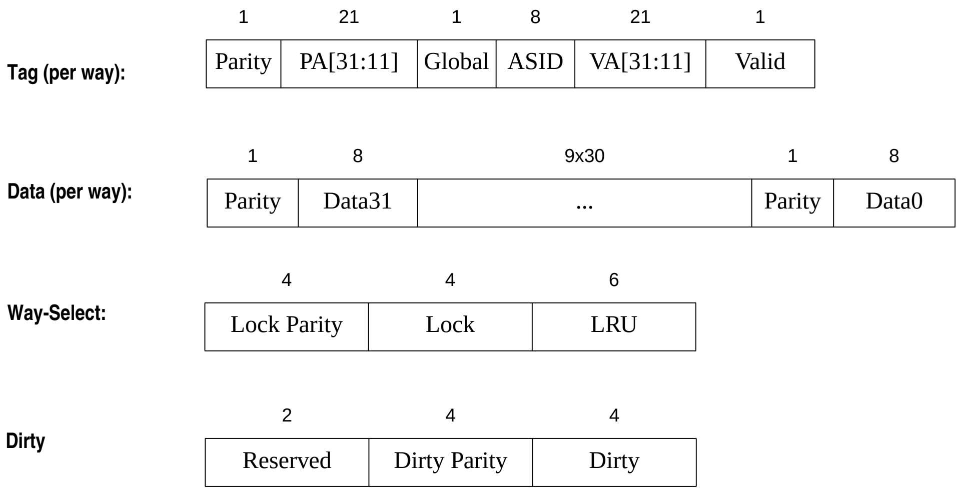
The data cache is similar to the instruction cache, with a few key differences. The data cache does not contain any precode information. To handle store bytes, the data array is byte-accessible, and the optional data parity is 1 bit per byte. The way-select array for the data cache holds the lock bits (and optional lock parity bits) for each cache line, in addition to the LRU information. The lock bits indicate the cache lines that have been locked using the CACHE instruction. There is a separate dirty array to hold the dirty bits of cache lines.

Table 9.2 shows the key characteristics of the data cache. Figure 9.2 shows the format of an entry in the arrays comprising the data cache: tag, data, way-select, and dirty.

Table 9.2 Data Cache Attributes

Attribute	With Parity	Without Parity
Size	0, 16, 32, 64KB	
Line Size	32B	
Number of Cache Sets	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	53b x 4	52b x 4
Write Unit	53b	52b
Way-Select Array		
Read Unit	14b	10b
Write Unit	1-14b	
Dirty Array		
Read Unit	10b	6b
Write Unit	1-10b	

Figure 9.2 Data Cache Organization



9.3.1 Virtual Aliasing

As indicated in the corresponding section on the instruction cache, virtual aliases can occur because the Data cache is indexed using virtual address bits. As in the instruction cache, virtual aliasing occurs only if the data cache is configured to be larger than 16KB.

Software must avoid virtual aliases in the data cache, though the core can be configured to use physical addresses to access the cache to achieve the same end goal.

9.3.2 Parity

Parity protection of the data cache arrays can optionally be included. The data array requires a parity bit for each byte, corresponding 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 lock bit, but the LRU bits are not covered by parity. The dirty array also has a parity bit for each dirty bit.

9.4 Uncached Accelerated Stores

Uncached Accelerated gathering is supported for word and doubleword stores only.

Gathering of uncached accelerated stores will start on cache-line aligned addresses, i.e. 32 byte aligned addresses. Uncached accelerated word or doubleword stores that do not to meet the conditions required to start gathering will be treated like regular uncached stores.

When 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 doubleword stores to the same cache line will write sequentially into this buffer, regardless 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 doubleword-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. Halfword Stores
  2. Unaligned Stores
  3. Store conditionals
- In order for software to execute correctly 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. In addition, the data cache has a dirty array. 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 bits of the physical address (bits [31:12] for instruction cache, bits[31:11] for data cache), 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 74K 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.

- *Writeback 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 in the cache. If it is, 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 74K™ Core" on page 85 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 7.52 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 74K 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

Because the caches are virtually indexed and physically tagged, a phenomenon known as *virtual aliasing* can occur for some cache sizes. 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. Virtual aliasing can 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 74K core. Because there are no direct writes to the I-cache in the MIPS architecture, aliasing is usually an issue only for the D-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 74K 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 13, “74K™ Processor Core Instructions” on page 303.

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 7.2.37 “ErrCtl (CP0 Register 26, Select 0): Software Parity Control and Test Modes for Cache RAM Arrays”.) Similarly, the *SPR* bit in the *ErrCtl* register will cause Index Load Tag and Index Store Tag instructions to read the pseudo-tags associated with the scratchpad

RAM array. Note that when the *WST* 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
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, some of which are described in the following subsections.

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 *ITagLo and ITagHi* registers into the selected tag entry. Index Load Tag will read the selected tag entry into the *ITagLo and ITagHi registers*.

If parity is implemented, the parity bits can be tested as normal bits by setting the *PO* (parity override) bit in the *ErrCtl* register. This will override the parity calculation and use the parity bits in *ITagLo and ItagHi* as the parity values.

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 Index Store Data instruction can optionally update the corresponding precode field in the tag array.The precode bits in the array are updated if the *PCD* bit in the *ErrCtl* register is zero when we execute the Index Store Data instruction. The precode value is generated by the hardware automatically if the PCO bit in the *ErrCtl* register is zero. Otherwise, the corresponding precode value (PREC_01/PREC_23/PREC_45/PREC_67) from the *ITagHi* register is used in updating the tag array.

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 *IDataLo* and *IDataHi* 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 lock and LRU bits can be tested using the same mechanism as the I-cache WS array.

9.7.6 D-cache DirtyArray

The testing of this array is also done through Index Load Tag and Index Store Tag CACHE instructions. By setting the *DYT* bit in the *ErrCtl* register, these operations will read and write the dirty array instead of the tag 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 when 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.

A 74K processor contains no direct hardware support for managing coherency with respect to its caches, so it must be handled via the 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. However, because of the presence of write buffers, there could be a delay in the actual write to memory. So if another memory master updates cacheable memory that could also be in the core's caches, 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 (until explicitly evicted). 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 they are explicitly forced by certain flavors of the CACHE or PREF instructions.

The SYNC instruction may also be useful to software in enforcing memory coherence, because it flushes the core's write buffers.

Chapter 10

Power Management in the 74K™ Core

A 74K 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 two mechanisms for system level low-power support, which are discussed in the following sections:

- Section 10.1 “Register-Controlled Power Management”
- Section 10.2 “Instruction-Controlled Power Management”

10.1 Register-Controlled Power Management

The *RP* (Reduced Power) 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 on the *SI_RP* output signal. Three additional pins— *SI_EXL*, *SI_ERL*, and *EJ_DebugM*—support the power-management functions 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 to reduce the clock frequency and place the core into power-down mode.

If an interrupt occurs 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. Setting 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. When *SI_EXL* is asserted, 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. The external agent can then 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, and 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

The second mechanism for invoking power-down mode 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. When 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 the state of the chip.

Chapter 11

EJTAG Debug Support in the 74K™ Core

The EJTAG debug logic in the 74K processor core is compliant with MIPS® EJTAG Specification Version 4.12 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*) 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 74K 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

31 30	29	28 18	17	16	15	14	13 11	10	9	8 6	5	4	3	2	1	0
Res	ENM	Res	DB	IB	IVM	DVM	0	CBT	PCS	PCR	PCSE	INTE	NMIE	NMIP	SRE	PE

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 which the processor is running in kernel and Debug Mode: **Encoding** / **Meaning** 0: Little endian 1: Big endian	R	Preset
Res	28:18	Reserved	R	0
DB	17	Indicates if data hardware breakpoint is implemented: **Encoding** / **Meaning** 0: No data hardware breakpoint implemented 1: Data hardware breakpoint implemented	R	Preset

Table 11.1 Debug Control Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	**Bit(s)**			
IB	16	Indicates if instruction hardware breakpoint is implemented: **Encoding** / **Meaning** 0: No instruction hardware breakpoint implemented 1: Instruction hardware breakpoint implemented	R	Preset
Res	13:11	Reserved	R	0
IVM	15	Indicates if inverted data value match on data hardware breakpoints is implemented: **Encoding** / **Meaning** 0: No inverted data value match on data hardware breakpoints implemented 1: Inverted data value match on data hardware breakpoints implemented	R	0
DVM	14	Indicates if a data value store on a data value breakpoint match is implemented: **Encoding** / **Meaning** 0: No data value store on a data value breakpoint match implemented 1: Data value store on a data value break-point match implemented	R	0
CBT	10	Indicates if complex breakpoint block is implemented: **Encoding** / **Meaning** 0: No complex breakpoint block imple-mented 1: Complex breakpoint block imple-mented	R	Preset
PCS	9	Indicates if the PC Sampling feature is implemented.: **Encoding** / **Meaning** 0: No PC Sampling implemented 1: PC Sampling implemented	R	Preset
PCR	8:6	PC Sampling Rate: Values from 0 to 7 map to 2^5 to 2^{12} cycles respectively. That is, a PC sample is written out every 32, 64, 128, 256, 512, 1024, 2048, or 4096 cycles. The external probe or soft-ware is allowed to set this value to the desired sample rate	R/W	7

Table 11.1 Debug Control Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
PCSE	5	Indicates if PC Sampling is enabled: **Encoding** | **Meaning** 0 | PC Sampling not enabled 1 | PC Sampling enabled This bit is set to 0 following Reset. It must be set by software to enable PC sampling.	R/W	0
INTE	4	Interrupt Enable in Normal Mode. This bit provides the hardware and software interrupt enable for non-debug mode, in addition to other masking mechanisms in conjunction with other disable mechanisms: **Encoding** | **Meaning** 0 | Interrupt disabled 1 | Interrupt enabled depending on other enabling mechanisms	R/W	1
NMIE	3	Non-Maskable Interrupt (NMI) enable for Non-Debug Mode: **Encoding** | **Meaning** 0 | NMI disabled 1 | NMI enabled	R/W	1
NMIP	2	Indication for pending NMI: **Encoding** | **Meaning** 0 | No NMI pending 1 | NMI pending	R	0
SRE	1	Soft Reset Enable This bit allows the system to mask soft resets. The core does not internally mask resets. Rather the state of this bit appears on the *EJ_SRstE* external output signal, allowing the system to mask soft resets if desired.: **Encoding** | **Meaning** 0 | Soft reset masked for soft reset sources dependent on implementation 1 | Soft reset is fully enabled Bit is read-only (R) and reads as zero if not implemented.	R/W	1

Table 11.1 Debug Control Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
PE	0	Probe Enable This bit reflects the value of the *ProbEn* bit in the EJTAG Control register: **Encoding** / **Meaning**: 0 — No access should occur to the dmseg segment 1 — Probe services accesses to the dmseg segment Bit is read-only (R) and reads as zero if not implemented.	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 in 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 74K core: instruction breakpoints and data breakpoints.

A core may be configured with the following breakpoint options:

- No breakpoints
- Four instruction and 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. 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_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. 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 =
          ( ( ( 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

	Breakpoints that Match		Update of BS Bits for Matching Data Breakpoints	
Instruction	**Without Value Compare**	**With Value Compare**	**Without Value Compare**	**With Value Compare**
Load/Store	One or more	None	*BS* bits set for all	(No matching breakpoints)
Load	One or more	One or more	*BS* bits set for all	Unchanged *BS* bits since load of data value does not occur so match of the breakpoint cannot be determined
Load	None	One or more	(No matching breakpoints)	*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 breakpoints)	*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 an 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 *DDBSImpr* 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 *DDBSImpr* bits and *DDBLImpr*/*DDBSImpr* in the *Debug* register 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 *DDBSImpr* bits and *DDBSImpr*/*DDBSImpr* bits are accessed for read or write. This delay ensures that these bits are fully updated.

Any *DDBSImpr* bit set prior to the match and debug exception are kept set, because only debug software can clear the *DDBSImpr* 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.

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
n is breakpoint number in range 0 to 3		

Table 11.7 Addresses for Instruction Breakpoint Registers

Offset in drseg	Register Mnemonic	Register Name and Description
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

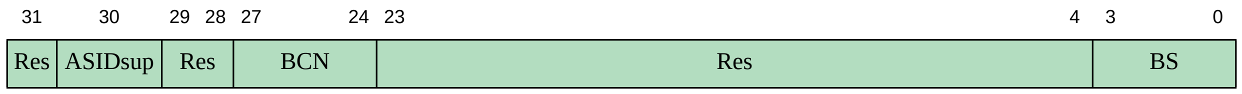


Table 11.8 IBS Register Field Descriptions

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. 0: No ASID compare. 1: ASID compare (*IBASIDn* register implemented).	R	Fixed MMU - 0 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

11.2.8.2 Instruction Breakpoint Address n (IBAn) Register

Compliance Level: Implemented only for implemented instruction breakpoints.

The *Instruction Breakpoint* Addr*ess n* (*IBAn*) register has the address used in the condition for instruction breakpoint *n*.

Figure 11.3 IBAn Register Format

31 1	0
IBA	ISA

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

31 1	0
IBM	ISAM

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: 0: Corresponding address bit not masked. 1: Corresponding address bit masked.	R/W	Undefined
ISAM	0	Instruction breakpoint ISA mode mask for condition: 0: ISA mode considered for match condition 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

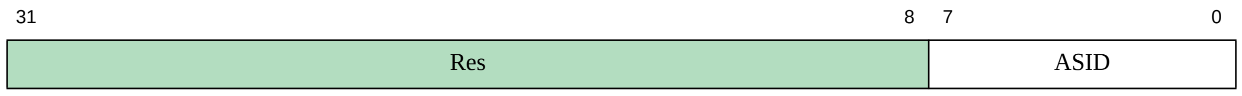


Table 11.11 IBASIDn Register Field Descriptions

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

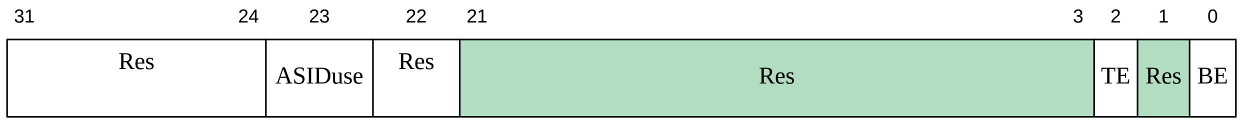


Table 11.12 BCn Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
Res	31:24	Must be written as zero; returns zero on read.	R	0
ASIDuse	23	Use ASID value in compare for instruction breakpoint n: 0: Don't use ASID value in compare 1: Use ASID value in compare	R/W	Undefined
Res	22	Must be written as zero; returns zero on read	R	0
Res	21:3	Must be written as zero; returns zero on read.	R	0
TE	2	Use instruction breakpoint n as triggerpoint: 0: Don't use it as triggerpoint 1: Use it as triggerpoint	R/W	0
Res	1	Must be written as zero; returns zero on read.	R	0

Table 11.12 BCn Register Field Descriptions

Fields				
Name	Bits	Description	Read / Write	Reset State
BE	0	Use instruction breakpoint n as breakpoint: 0: Don't use it as breakpoint 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 contains implementation and status information about the data breakpoints.

The *ASIDsup* field indicates whether ASID compares are supported.

Figure 11.7 DBS Register Format

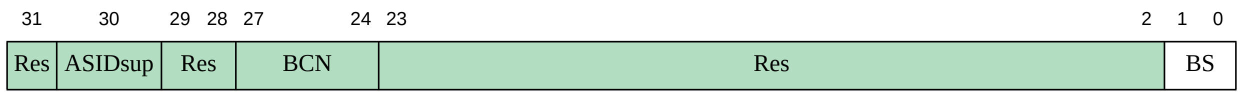


Table 11.14 DBS Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
Res	31	Must be written as zero; returns zero on read.	R	0
ASIDsup	30	Indicates that ASID compares are supported in data breakpoints. 0: Not supported 1: Supported	R	TLB MMU - 1 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

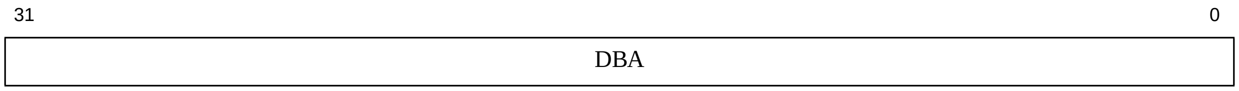


Table 11.15 DBAn Register Field Descriptions

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

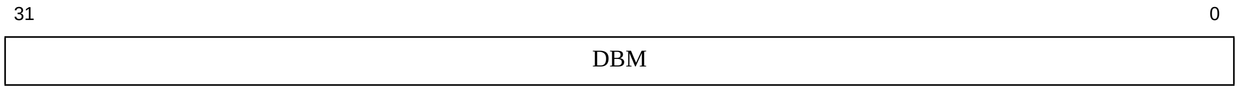


Table 11.16 DBMn Register Field Descriptions

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

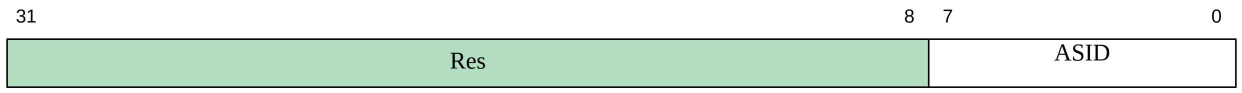


Table 11.17 DBASIDn Register Field Descriptions

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

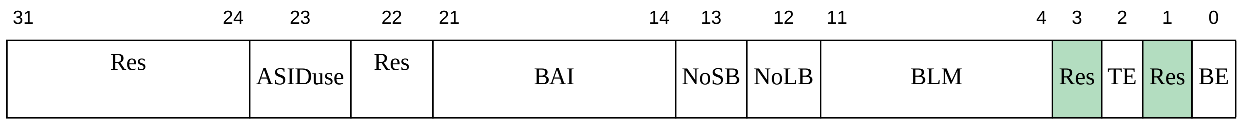


Table 11.18 DBCn Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	**Bits**			
Res	31:24	Must be written as zero; returns zero on reads.	R	0

Table 11.18 DBCn Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bits			
ASIDuse	23	Use ASID value in compare for data breakpoint n: 0: Don't use ASID value in compare 1: Use ASID value in compare	R/W	Undefined
Res	22	Must be written as zero; returns zero on reads	R	0
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. 0: Condition depends on access to corresponding byte 1: Access for corresponding byte is ignored	R/W	Undefined
NoSB	13	Controls if condition for data breakpoint is not fulfilled on a store transaction: 0: Condition may be fulfilled on store transaction 1: Condition is never fulfilled on store transaction	R/W	Undefined
NoLB	12	Controls if condition for data breakpoint is not fulfilled on a load transaction: 0: Condition may be fulfilled on load transaction 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.: 0: Compare corresponding byte lane 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: 0: Don't use it as triggerpoint 1: Use it as triggerpoint	R/W	0
Res	1	Must be written as zero; returns zero on reads.	R	0
BE	0	Use data breakpoint n as breakpoint: 0: Don't use it as breakpoint 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

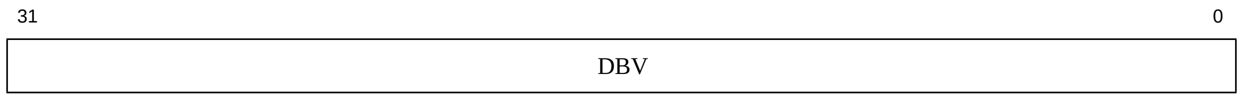


Table 11.19 DBVn Register Field Descriptions

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 Input clock used to shift data into or out of the Instruction or data registers. The *TCK* clock is independent of the processor clock, so the EJTAG probe can drive *TCK* independently of the processor clock frequency. The core signal for this is called *EJ_TCK*
TMS	I	Test Mode Select Input The *TMS* input signal is decoded by the TAP controller to control test operation. *TMS* is sampled on the rising edge of *TCK*. The core signal for this is called *EJ_TMS*
TDI	I	Test Data Input Serial input data (*TDI*) is shifted into the Instruction register or data registers on the rising edge of the *TCK* clock, depending on the TAP controller state. The core signal for this is called *EJ_TDI*
TDO	O	Test Data Output 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. The core signal for this is called *EJ_TDO* with output enable controlled by *EJ_TDOzstate*.
TRST_N	I	Test Reset Input (Optional pin) The *TRST_N* pin is an active-low signal for asynchronous reset of the TAP controller and instruction in the TAP module, independent of the processor logic. The processor is not reset by the assertion of *TRST_N*. The core signal for this is called *EJ_TRST_N* This signal is optional, but power-on reset must apply a low pulse on this 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 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* state by a 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.

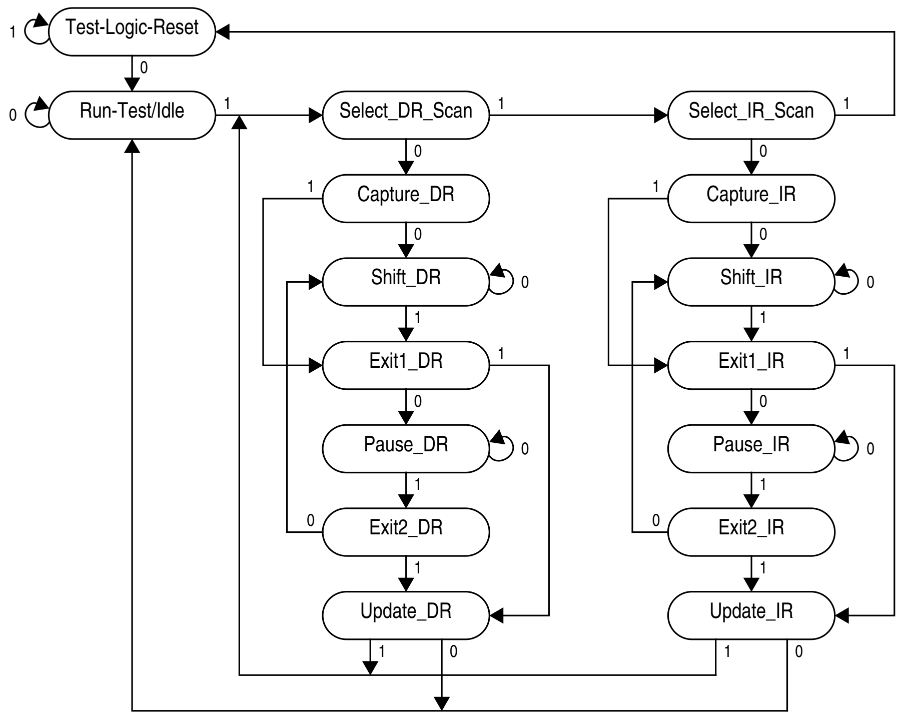
When test access is required, a protocol is applied via the *TMS* and *TCK* inputs, causing the TAP to exit the *Test-Logic-Reset* state and move through the appropriate states. From the *Run-Test/Idle* state, an Instruction register scan or a data register scan can be issued to transition the TAP through 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
0x16	TCBCONTROLE	Selects the *TCBTCONTROLE* register in the Trace Control Block
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.

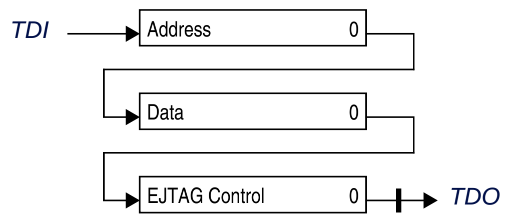
11.3.3.6 CONTROL Instruction

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

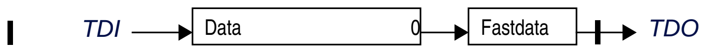
When the NORMALBOOT instruction is given and the Update-IR state is left, then the reset value of the *ProbTrap*, *ProbEn* and *EjtagBrk* bits in the *EJTAG Control* register are set to 0 after reset.

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 unit is present. If no TRU is present, 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 unit is present. If no TRU 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 unit is present. If no TRU is present, then this instruction will select the *Bypass* register.

11.3.3.14 TCBCONTROLE Instruction

This instruction is used to select the *TCBCONTROLE* register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace unit is present. If no TRU is present, then this instruction will select the *Bypass* register.

11.3.3.15 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 unit is present. If no TRU 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.

11.3.3.16 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*
- *Device Identification*
- *Implementation*
- *EJTAG Control* (*ECR*)
- *Processor Access Address*
- *Processor Access Data*
- *FastData*

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

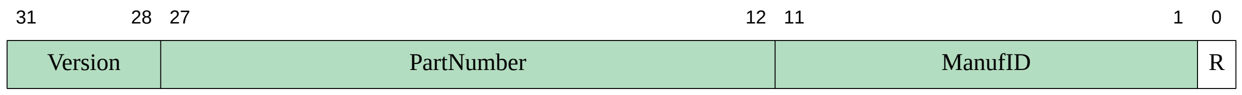


Table 11.23 Device Identification Register

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
Version	31:28	**Version** (4 bits) This field identifies the version number of the processor derivative.	R	*EJ_Version[3:0]*
PartNumber	27:12	**Part Number** (16 bits) This field identifies the part number of the processor derivative.	R	*EJ_PartNumber[15:0]*
ManufID	11:1	**Manufacturer Identity** (11 bits) Accordingly to IEEE 1149.1-1990, the manufacturer identity code shall be a compressed form of the JEDEC Publications 106-A.	R	*EJ_ManufID[10:0]*
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

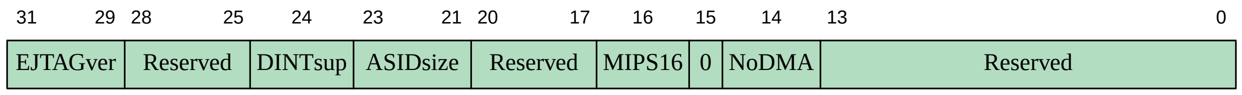


Table 11.24 Implementation Register Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
EJTAGver	31:29	EJTAG Version 4.14	R	3
Reserved	28:25	Reserved	R	0
DINTsup	24	DINT Signal Supported from Probe This bit indicates if the DINT signal from the probe is supported: 0: DINT signal from the probe is not supported 1: Probe can use DINT signal to make debug interrupt.	R	*EJ_DINTsup*
ASIDsize	23:21	Size of ASID field in implementation: 0: No ASID in implementation 2: 8-bit ASID 1,3: Reserved	R	TLB MMU- 2 FM MMU- 0
Reserved	20:17	Reserved	R	0
MIPS16	16	Indicates whether MIPS16 is implemented 0: No MIPS16 support 1: MIPS16 implemented	R	1

Table 11.24 Implementation Register Descriptions (Continued)

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

31	30 29	28 24	23	22	21	20	19	18	17	16	15	14	13	12	11 4	3	2 0
Rocc	Psz	Res	Res	Doze	Halt	PerRst	PRnW	PrAcc	Res	PrRst	ProbEn	ProbTrap	Res	EjtagBrk	Res	DM	Res

Table 11.25 EJTAG Control Register Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
Rocc	31	Reset Occurred The bit indicates if a CPU reset has occurred: 0: No reset occurred since bit last cleared. 1: Reset occurred since bit last cleared. The *Rocc* bit will keep the 1 value as long as reset is applied. This bit must be cleared by the probe, to acknowledge that the incident was detected. The *EJTAG Control* register is not updated in the *Update-DR* state unless *Rocc* is 0, or written to 0. This is in order to ensure proper handling of processor access.	R/W	1

Table 11.25 EJTAG Control Register Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
Psz[1:0]	30:29	Processor Access Transfer Size These bits are used in combination with the lower two address bits of the Address register to determine the size of a processor access transaction. The bits are only valid when processor access is pending.	R	Undefined

PAA[1:0]	Psz[1:0]	Transfer Size
00	00	Byte (LE, byte 0; BE, byte 3)
01	00	Byte (LE, byte 1; BE, byte 2)
10	00	Byte (LE, byte 2; BE, byte 1)
11	00	Byte (LE, byte 3; BE, byte 0)
00	01	Halfword (LE, bytes 1:0; BE, bytes 3:2)
10	01	Halfword (LE, bytes 3:2; BE, bytes 1:0)
00	10	Word (LE, BE; bytes 3, 2, 1, 0)
00	11	Triple (LE, bytes 2, 1, 0; BE, bytes 3, 2,1)
01	11	Triple (LE, bytes 3, 2, 1; BE, bytes 2, 1, 0)
All others		Reserved

Note: LE=little endian, BE=big endian, the byte# refers to the byte number in a 32-bit register, where byte 3 = bits 31:24; byte 2 = bits 23:16; byte 1 = bits 15:8; byte 0=bits 7:0, independently of the endianess.

Name	Bit(s)	Description	Read / Write	Reset State
Res	28:24	Reserved	R	0
Res	23	Reserved	R	0
Doze	22	Doze state The Doze bit indicates any kind of low-power mode. The value is sampled in the Capture-DR state of the TAP controller: 0: CPU not in low-power mode. 1: CPU is in low-power mode Doze includes the Reduced Power (RP) and WAIT power-reduction modes.	R	0
Halt	21	Halt state The Halt bit indicates if the internal system bus clock is running or stopped. The value is sampled in the Capture-DR state of the TAP controller: 0: Internal system clock is running 1: Internal system clock is stopped	R	0
PerRst	20	Peripheral Reset When the bit is set to 1, it is only guaranteed that the peripheral reset has occurred in the system when the read value of this bit is also 1. This is to ensure that the setting from the *TCK* clock domain gets effect in the CPU clock domain, and in peripherals. When the bit is written to 0, then the bit must also be read as 0 before it is guaranteed that the indication is cleared in the CPU clock domain also. This bit controls the *EJ_PerRst* signal on the core.	R/W	0

Table 11.25 EJTAG Control Register Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
PRnW	19	Processor Access Read and Write This bit indicates if the pending processor access is for a read or write transaction, and the bit is only valid while PrAcc is set: 0: Read transaction 1: Write transaction	R	Undefined
PrAcc	18	Processor Access (PA) Read value of this bit indicates if a Processor Access (PA) to the EJTAG memory is pending: 0: No pending processor access 1: Pending processor access The probe's software must clear this bit to 0 to indicate the end of the PA. Write of 1 is ignored. A pending 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. The FASTDATA access can clear this bit.	R/W0	0
Res	17	Reserved	R	0
PrRst	16	Processor Reset (Implementation dependent behavior) When the bit is set to 1, then it is only guaranteed that this setting has taken effect in the system when the read value of this bit is also 1. This is to ensure that the setting from the *TCK* clock domain gets effect in the CPU clock domain, and in peripherals. When the bit is written to 0, then the bit must also be read as 0 before it is guaranteed that the indication is cleared in the CPU clock domain also. This bit controls the *EJ_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 This bit indicates to the CPU if the EJTAG memory is handled by the probe so processor accesses are answered: 0: The probe does not handle EJTAG memory transactions 1: The probe does handle EJTAG memory transactions It is an error by the software controlling the probe if it sets the Prob-Trap bit to 1, but resets the *ProbEn* to 0. The operation of the processor is UNDEFINED in this case. The *ProbEn* bit is reflected as a read-only bit in the *ProbEn* bit, bit 0, in the *Debug Control Register* (*DCR*). The read value indicates the effective value in the DCR, due to synchronization issues between *TCK* and CPU clock domains; however, it is ensured that change of the *ProbEn* prior to setting the *EjtagBrk* bit will have effect for the debug handler executed due to the debug exception. The reset value of the bit depends on whether the EJTAGBOOT indication is given or not: No EJTAGBOOT indication given: 0 EJTAGBOOT indication given: 1	R/W	0 or 1 from EJTAGBOOT
ProbTrap	14	Probe Trap This bit controls the location of the debug exception vector: 0: In normal memory 0xBFC0.0480 1: In EJTAG memory at 0xFF20.0200 in dmseg Valid setting of the *ProbTrap* bit depends on the setting of the *ProbEn* bit, see comment under *ProbEn* bit. The *ProbTrap* should not be set to 1, for debug exception vector in EJTAG memory, unless the *ProbEn* bit is also set to 1 to indicate that the EJTAG memory may be accessed. The read value indicates the effective value to the CPU, due to synchronization issues between *TCK* and CPU clock domains; however, it is ensured that change of the *ProbTrap* bit prior to setting the *EjtagBrk* bit will have effect for the *EjtagBrk*. The reset value of the bit depends on whether the EJTAGBOOT indication is given or not: No EJTAGBOOT indication given: 0 EJTAGBOOT indication given: 1	R/W	0 or 1 from EJTAGBOOT
Res	13	Reserved	R	0
EjtagBrk	12	EJTAG Break Setting this bit to 1 causes a debug exception to the processor, unless the CPU was in debug mode or another debug exception occurred. When the debug exception occurs, the processor core clock is restarted if the CPU was in low-power mode. This bit is cleared by hardware when the debug exception is taken. The reset value of the bit depends on whether the EJTAGBOOT indication is given or not: No EJTAGBOOT indication given: 0 EJTAGBOOT indication given: 1	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 This bit indicates the debug or non-debug mode: 0: Processor is in non-debug mode 1: Processor is in debug mode The bit is sampled in the *Capture-DR* state of the TAP controller.	R	0
Res	2:0	Reserved	R	0

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

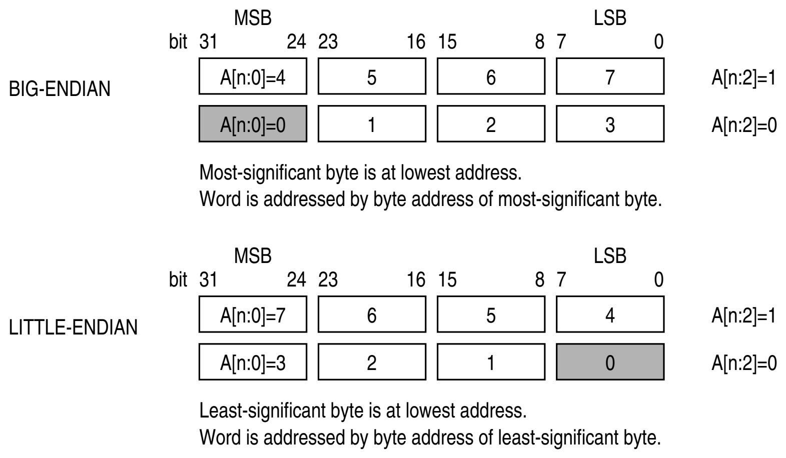
The *Processor Access Data* (*PAD*) register is used to provide data value to and from a processor access. The length of the Data register is 32 bits, and this register is selected by shifting in the DATA instruction.

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 PAD Register


MSB
LSB
bit 31 24 23 16 15 8 7 0
BIG-ENDIAN
A[n:0]=4
5
6
7
A[n:2]=1
A[n:0]=0
1
2
3
A[n:2]=0
Most-significant byte is at lowest address.
Word is addressed by byte address of most-significant byte.
MSB
LSB
bit 31 24 23 16 15 8 7 0
LITTLE-ENDIAN
A[n:0]=7
6
5
4
A[n:2]=1
A[n:0]=3
2
1
0
A[n:2]=0
Least-significant byte is at lowest address.
Word is addressed by byte address of least-significant byte.

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

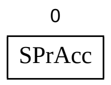


Table 11.26 Fastdata Register Field Description

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. 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 (previ-ous) 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 1, 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 SampleRate). 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

48 41	40 33	32 1	0
TC (for MIPS MT processors only)	ASID	PC	New

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

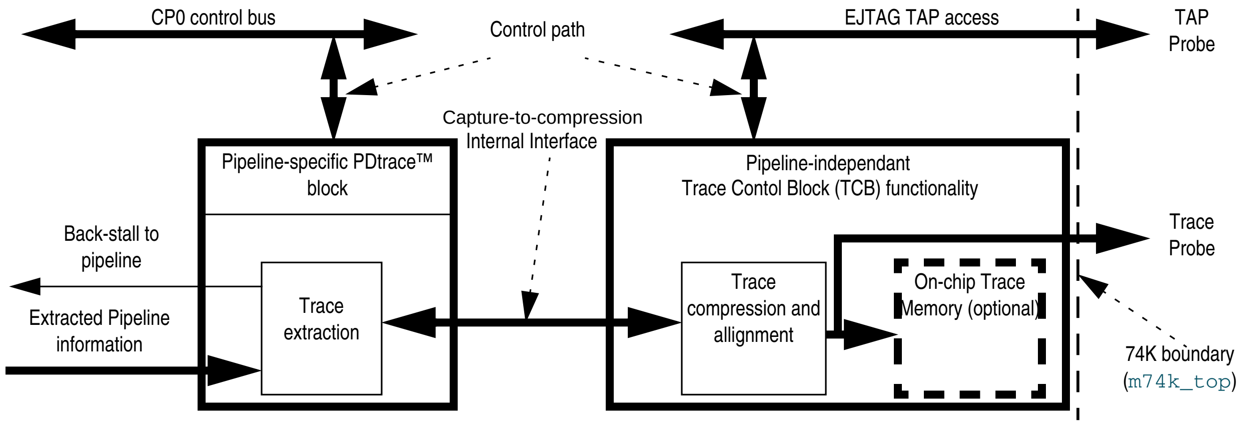
MIPS Trace enables the ability to trace program flow, load/store addresses, and load/store data, in addition to optional tracing of performance counters and core-specific inefficiencies. Several run-time options exist for the level of information which is traced, for example, tracing only when in specific processor modes (e.g., User Mode or Kernel Mode).

MIPS Trace is an optional block in the 74K core. If MIPS Trace is not implemented, the remainder of this chapter is irrelevant. When MIPS Trace is implemented, the *CP0 Config3*$_{TL}$ bit is set by hardware when the core is configured.

The pipeline-specific architecture of MIPS Trace is specified in the *PDtrace™ Interface and Trace Control Block Specification* [7]. There are two primary functional blocks: the PDtrace capture block and blocks that implement the Trace Control Block (TCB) functionality. The PDtrace capture block extracts the trace information from the end of the processor pipeline from the in-order graduation stage and stores the information in an internal FIFO called the Unified FIFO. The capture block then presents the data from the Unified FIFO to the PDtrace compression block. The compression block along with the TCB Registers implement what was previously known as the Trace Control Block (TCB). Though there is not an explicit module called TCB, the functionality of the TCB, as specified in [7] has been completely implemented and integrated into the PDtrace unit. Thus, it is no longer a customer option to implement a custom TCB.

Note that the generic pin interface, as defined in the retired document *PDtrace™ Interface Specification*, that was used to “communicate” between the capture and the TCB functionality is deprecated. The interface is replaced by an internal interface that is called the capture-to-compression interface. This interface is embedded inside the 74K core, and will not be discussed in detail here. Suffice it to say that the internal interface embodies all the functionality, as described in the document *PDtrace™ Interface and Trace Control Block Specification*. 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 Functional Blocks in the 74K™ Core


CP0 control bus
Control path
EJTAG TAP access
TAP Probe
Capture-to-compression Internal Interface
Pipeline-specific PDtrace™ block
Pipeline-independant Trace Contol Block (TCB) functionality
Back-stall to pipeline
Extracted Pipeline information
Trace extraction
Trace compression and allignment
On-chip Trace Memory (optional)
Trace Probe
74K boundary (m74k_top)

To some extent, the two modules provide similar trace control features, but the access to these features is quite different. The PDtrace software controls can only be reached through access to CP0 registers. The PDtrace hardware controls can only be reached through EJTAG TAP access. The selection of one of these controls determines what is traced from the core pipeline, and the information presented in the internal capture-to-compression interface.

Before describing the MIPS Trace implemented in the 74K 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 used to control for the trace. 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 TCBRegs 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 to toggle *TCK* atleast once before it is turned on. It is assumed that the EJTAG probe will be connected while using trace, and the probe’s reset sequence would toggle *TCK*. Note that to extract trace data from the trace compression block, $TCBCONTROLB_{EN}$ 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 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 the 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. The sync instruction is not a load and not a store.

All the instruction flags above are combined into one, 3-bit value, called the “instruction completion” 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.

1. An SC (Store Conditional) instruction is flagged as a store instruction, even if the load-locked bit prevented the actual store. Thus the SC does not have special handling and is treated as any other store; it is up to the reconstruction software to determine if the SC succeeded or failed.

Trace regeneration software must 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.

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 processes, 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 optional performance counter trace is enabled, when the specific events as defined in Section 11.7.11 "Performance Counter Tracing" occur, upto four performance counter registers are traced.

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

Two special CP0 registers, *UserTraceData1* and *UserTraceData2,* can generate a data trace. When either of these registers 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. Since writing these registers is performed via an MTC0 operation, only one register is updated in any given cycle. Thus in the same cycle, only one of the UserTraceData registers is traced. However in back to back cycles, the tracing of the two registers can alternate, and is handled correctly.

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 to the Trace Compression and Control Block. 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 three ways:

- Set/clear the *TCBCONTROLB*$_{EN}$ bit via an EJTAG TAP operation.
- Initialize a TCB trigger to set/clear the *TCBCONTROLB*$_{EN}$ bit.
- Use the drseg mapping of *TCBCONTROLB* to clear *TCBCONTROLB*$_{EN}$ via a load to drseg space. See Section 11.7.15 "Memory-mapped Access to On-Chip Trace RAM" for special access rules.

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. That is a TF6 is inserted 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 by the PDtrace capture block. 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 Instruction and Data Cache Miss Tracing

It is possible to embed information about Instruction and/or Data cache misses into the trace. There are limitations in the core’s ability to track this and put useful information into the trace.

For the instruction cache miss indicator:

- The instruction cache miss indicator is based on whether the instruction is pulled from the cache or the fill buffer. On a cache miss, the fetch is restarted when the data comes back from the BIU and the instructions will come from the Fill Buffer. The miss flag is only set for the first fetch that hits out of the FB to avoid marking every fetch from the line a miss. However, two instructions can be fetched per cycle and both will be marked as a miss. If branching to the middle of a dword though, only 1 miss will be seen.

- The IFU can prefetch down a speculative path which might not be immediately executed. These speculative fetches are filled into the cache. Subsequently, when the code accesses the same address, it is possible that the instruction will hit in the cache even if that instruction was being executed for the very first time.

For the data cache miss indicator:

- PDtrace instruction capture is done at the end of the GRU (graduation) pipe. However at this point, the cache miss info is not accurate. Hit indication is accurate, but the miss indication is not. The miss could turn into a hit after it enters the LSU graduation buffer. Thus, this miss indicator is instead sent with the data value.

  - For loads, this allows an accurate miss indication as the miss state must be resolved before we have the data.

  - For stores, the miss indicator is also sent with the data value. The store data value is captured when the store instruction exist the LSU graduation buffer.

11.7.11 Performance Counter Tracing

The optional feature of dumping performance counter values through the trace stream provides the ability to correlate performance counter events to the specific instruction execution path. *TraceControl3*$_{PeC}$ indicates if this optional feature is implemented. Furthermore the feature is enabled via *TraceControl3*$_{PeCE}$ / *TCBCONTROLE*$_{PeCE}$. The performance counters are traced out based on four specific events. When an enabled event occurs upto four performance counters are traced and in addition the traces a fifth value. The fifth value is the cycle count register. Tracing the cycle count register is unique to the core. Control over what particular performance counter is traced is specified by bit *PCTD* in each *Performance Counter Control* Register. If set to zero (default setting), tracing is enabled for this performance counter, and if set to one, tracing is disabled for this performance counter. Disabling tracing for each individual performance counter in their respective control register will still generate data, that is the cycle counter is traced out regardless of the individual performance counter control *PCTD* bit. In the case where more than one event occurs in the same cycle, the performance counter values are traced only once for that cycle.

1. Synchronization counter expiration will trigger tracing of the performance counter values. This is controlled by *TraceControl3*$_{PeCSync}$ / *TCBCONTROLE*$_{PeCSync}$.

2. Hardware trace breakpoint will trigger tracing of the performance counter values. This is contingent on several control bit settings. The TE bit in the breakpoint control register should be set. This allows a trigger signal to be sent to the Trace Unit. When set *TraceControl3*$_{PeCBP}$ / *TCBCONTROLE*$_{PeCBP}$ act as the enable for performance counter tracing. Additionally the generation of a performance counter trigger is controlled by setting active both *TraceIBPC*$_{PCT}$ and *TraceIBPC*$_{IE}$, and, or setting active hi both *TraceDBPC*$_{PCT}$ and *TraceDBPC*$_{IE}$ Furthermore the BreakPointControl field for the specific hardware breakpoint in *TraceIBPC* or *TraceDBPC* must be encoded as 3'b100 or 3'b101 to allow performance counter values into the trace stream.

3. Function call, function return or exception occurrence will trigger tracing of the performance counter values.This is controlled by *TraceControl3*$_{PeCFCR}$ / *TCBCONTROLE*$_{PeCFCR}$.

4. An overflow of an active performance counter will trigger tracing of the performance counter values. This is controlled by *TraceControl3*$_{PeCOvf}$ / *TCBCONTROLE*$_{PeCOvf}$.

The Performance counter data will always use a TF3 with the PCV bit set to one. If the traced data is not Performance counter data, and performance counter tracing is enabled, then the PCV bit will be zero.

11.7.12 Filtered Data Trace Mode

This mode is used to support tracing of events in an application code on a Linux system. This type of instrumented code tracing is primarily used for performance analysis although it can also be used for event logging and debug. Filtered data tracing mode provides a mechanism to do low overhead event tracing from user application code since the UserTraceData registers require a kernel call from user mode.

In this mode, data load and store addresses are compared to the hardware data breakpoint address, if the addresses match, the data value associated with that match along with the address are traced out.

This mode works even when data address and/or value tracing is turned on. However, the general usage model is when both PC and Data trace are turned off since it may not always be possible to identify data that was traced due to a match vs. the regular data stream. This mode is used to shadow one or more static (fixed-address) variables. When there is a store to the variable, the store value is captured into the trace. Since there are generally two or more data triggers/watchpoints, the trace will need to uniquely identify the shadowed variable by also tracing out the associated address.

Filtered Data Trace mode is controlled by T*raceControl2*$_{FDT}$ / *TCBCONTROLB*$_{FDT}$.

11.7.13 PC tracing off

PC tracing turned off is simply to disable PC tracing which is controlled by T*raceControl2*$_{Mode.PC}$ / *TCBCONTROLC*$_{Mode.PC}$. Turning off this bit has more implications than simply not tracing the PC. There is some special behavior which is contingent on the setting of other mode bits.

1. PC tracing off, *TLSM*=1 (T*raceControl*$_{TLSM}$ / *TCBCONTROLA*$_{TLSM}$), Address tracing=0, Data tracing=0. For data cache misses trace out full PC and full address and the associated instruction completions. Instruction completion information not associated with a data cache miss will not appear in trace memory.

2. PC tracing off, *TLSM*=1, One or both of these modes is enabled {Address tracing, Data tracing}: PC is not traced out, only what is enabled gets traced out, for example if address tracing is enable, then the full address is traced out. Instruction completion information not associated with a traced address or a traced data will not appear in trace memory.

3. PC tracing off, *TIM*=1 (*TraceControl*$_{TIM}$ / *TCBCONTROLA*$_{TIM}$), or TFCR=1 (*TraceControl*$_{TFCR}$ / *TCBCONTROLA*$_{TFCR}$): If an instruction cache miss or function call/return occurs, then the full PC is traced along with the corresponding instruction completion information.

4. PC tracing off, *TLSM*=0, *TIM*=0, *TFCR*=0: all trace messages related to instructions are disabled. TF6 with no-trace counts can still be generated if Cycle Accurate mode is enabled. TF2 should never be generated.

In addition, PC-sync messages are globally disabled. The reconstruction software would need a PC-sync in the case of *TLSM*=1 if the PCs traced out were delta PCs. However given that the full PC is traced, there is no need for the PC-Sync message.

With PC tracing disabled there is a significant decrease in the instruction completion information that is traced. Only if the PC, address, or data have been traced out will the corresponding instruction completion also be traced, else the instruction completion is dropped.

11.7.14 TMOAS Handling

The *MIPS PDtrace™ Specification* requires a TMOAS transaction to be inserted into the trace stream. TMOAS transactions are used to record processor mode change, start or end of the tracing activity, overflow of the internal buffers in the PDtrace unit, or periodical synchronization. The following is a summary of the cases where a TMOAS transaction is generated:

- **Start of Tracing,** When tracing is first started, or when it is re-started after a break, some basic information is needed first to allow external software to identify the trace start point in the static program image, and make some reasonable conclusions about the processor mode at the start of tracing. At the start of the tracing, a TMOAS record is sent out the same time as the first completed instruction. This trace record type shows the processor mode and the ASID value of the currently executing processor. This record is followed by a trace of the full PC value for the first instruction traced.

- **Trace Synchronization,** The synchronization tracing function is triggered when the internal synchronization counter overflows based on the synchronization period bits as set in the (*TraceControl2*$_{SyP}$/*TCBCONTROLA*$_{SyP}$). Similar to the start of tracing, when the synchronization period is reached, a TMOAS record is first sent, followed by a full PC value. Note that the TMOAS associated with synchronization is sent only when the IPC instruction has been identified, to prevent other TType records between the TMOAS and the full PC trace for the synchronization.

- **Trace Overflow and Restart.** The trace unit's internal FIFO or buffers are used to hold address and data values waiting to be compressed, formatted, and traced out of the processor. It is possible to have a program sequence that overflows one or more of these FIFOs. In the situation that the FIFO overflows, the core is essentially losing trace data and hence the output gets illogical and no longer is a true representation of the program execution sequence. In this situation, abandon tracing in the current cycle, discard all entries in the FIFO, and restart tracing from the next completed instruction in the following cycle. In this situation, a TMOAS record is first sent after the overflow.

- **Tracing During Processor Mode Changes.** During normal execution, the processor will change its operation mode frequently. For example, when executing user-level code, an interrupt may cause the processor to jump to kernel mode to service the interrupt. When the interrupt has been serviced, the processor will switch back to user mode. A mode change is indicated in the tracing logic by tracing out a TMOAS for TType. In the situation that the mode change affects tracing, for example, the tracing system has been set up to trace only in user mode and not in kernel mode, then the interrupt service routine should not be traced. Upon jumping to kernel mode, the core tracing logic will add a TMOAS as the last record. When jumping from a non-tracing mode to a tracing mode, the first record output is TMOAS to indicate the mode change. This is followed by a full PC value of the first instruction in the tracing mode. This will enable the external trace reconstruction software to re-synchronize itself and track program execution in the desired mode.

Figure 11.24 and Table 11.28 are the bit field description for a TMOAS record. A TMOAS record is usually associates with an instruction, except for the case of a trace end TMOAS, where a TMOAS is sent out because the processor enters a non-tracing mode. In this case, the TMOAS is not associated with any instruction since the processor is not tracing, some of the field in the TMOAS record can be invalid data, for example, the ISAM field can be undertemi-nistic. This should not present an issue for the software since this TMOAS is only used as an indication that the trace has ended.

.

Figure 11.24 A TMOAS Trace Record

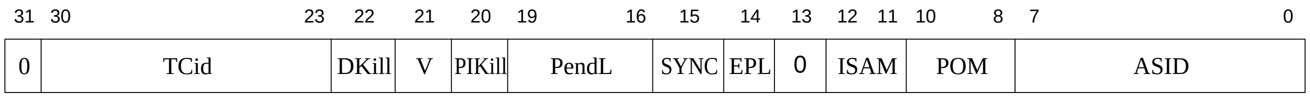


Table 11.28 TMOAS Trace Record Field Descriptions

Fields		
Name	Bits	Description
TCid	30..23	Only required if the processor implements MT, otherwise reserved.
DKill	22	Only required if the processor implements MT, otherwise reserved.
V	21	Only required if the processor implements MT, otherwise ignored.
PIKill	20	Only required if the processor implements MT, otherwise ignored.
PendL	19:16	This field is valid only when *SYNC* is 1, see below. When *SYNC* is 1, this field indicates the number of outstanding loads and stores at the IPC cycle. If the number of loads/store is zero, then all data transmissions' TDs after that are ignored until the next load/store instruction, at which point counting is restarted. Such TD transmissions are from store instructions which could not complete before the IPC signal was sent. Note that a sync happens with an InsComp value of IPC. Depending on whether or not there is data buffered up internally waiting to be sent out, the accompanying TMOAS may not be sent until several cycles later. In the meantime, any data sent in between the IPC and the TMOAS record may be ignored (at trace start or after an overflow) since this belongs to load and store instructions that happened before the sync. Now, if there are any load or store instructions between the IPC and the TMOAS, then the data for this will only be seen after the TMOAS is transmitted, since they would get buff-ered behind the TMOAS.
SYNC	15	When 0, this record was sent when the *ASID*, *POM*, or *ISAM* changed. When 1, this record was sent for a synchronization event.
EPL	14	When 1, the *PendL* field is to be interpreted as (*PendL* + 16). When 0, the *PendL* field is interpreted by itself. This is introduced in PDtrace rev. 6.00

Fields		Description
Name	Bits	
ISAM	12:11	(see value table below)

Value	In Architecture Mode
00	MIPS32
01	MIPS64
10	MIPS16e from MIPS32 mode
11	MIPS16e from MIPS64 mode

Name	Bits	Description
POM	10:8	(see value table below)

Value	Description
000	Kernel Mode (*EXL* = 0, *ERL* = 0)
001	Exception Mode (*EXL* = 1, *ERL* = 0)
010	Exception Mode (*EXL* = don't care, *ERL* = 1)
011	Debug Mode
100	Supervisor Mode
101	User Mode
110	Reserved
111	Reserved

Name	Bits	Description
ASID	7:0	The ASID of the current process. If the processor does not implement the standard TLB-based MMU, this field is always traced as a zero because the *EntryHi* register, and hence the *ASID*, is not defined.
0	31,13	Reserved for future use

11.7.15 Memory-mapped Access to On-Chip Trace RAM

The main access mode to the on-chip trace memory is done by the TAP Probe using the EJTAG Tap access port to the Trace unit. An alternate method to access the on-chip trace memory is provided. The on-chip trace memory can be accessed directly by software using load and store instructions. Access is provided by mapping the TCB registers to drseg address space, which allows them to be accessed by software in debug mode. Since the TCB registers accessed indirectly via *TCBData* by the TAP Probe are mapped directly to drseg, the *TCBData* register does not need to be mapped.

The mapped drseg registers are shown in Table 11.29. These mappings are “active” only when an external probe is

Table 11.29 Mapping TCB Registers in drseg

Offset in drseg	Register Name	Description
0x3000	*TCBControlA*	The TCBControlA register. See Section 11.9.1 “TCBCONTROLA Register” for more details about register contents.
0x3008	*TCBControlB*	The TCBControlB register. See Section 11.9.2 “TCBCONTROLB Register” for more details about register contents.
0x3010	*TCBControlC*	The TCBControlC register. See Section 11.9.4 “TCBCONTROLC Register” for more details about register contents.
0x3020	*TCBControlE*	The TCBControlE register. See Section 11.9.5 “TCBCONTROLE Register” for more details about register contents.
0x3028	*TCBConfig*	The TCBConfig register. See Section 11.9.6 “TCBCONFIG Register (Reg 0)” for more details about register contents.
0x3100	*TCBTW*	Trace Word read register. This register holds the Trace Word just read from on-line trace memory. See Section 11.9.7 “TCBTW Register (Reg 4)” for more details about register contents.
0x3108	*TCBRDP*	Trace Word Read pointer. It points to the location in the on-line trace memory where the next Trace Word will be read. A TW read has the side-effect of post-incrementing this register value to point to the next TW location. (A maximum value wraps the address around to the beginning of the trace memory). See Section 11.9.8 “TCBRDP Register (Reg 5)” for more details about register contents.
0x3110	*TCBWRP*	Trace Word Write pointer. It points to the location in the on-line trace memory where the next new Trace Word will be written. See Section 11.9.9 “TCBWRP Register (Reg 6)” for more details about register contents.
0x3118	*TCBSTP*	Trace Word Start Pointer. It points to the location of the oldest TW in the on-chip trace memory. See Section 11.9.10 “TCBSTP Register (Reg 7)” for more details about register contents.
0x3120	*BKUPRDP*	This is not a TCB register, but needed on a reset to save the *TCBRDP* value before that register is reset to 0. This allows the software that comes up after a (hard or soft) reset to know the last-known good value of *TCBRDP* before system crash, and potentially read the trace memory from or to the appropriate trace memory location.
0x3128	*BKUPWRP*	This is not a TCB register, but needed on a reset to save the *TCBWRP* value before that register is reset to 0. This allows the software that comes up after a (hard or soft) reset to know the last-known good value of *TCBWRP* before system crash, and potentially read the trace memory from or to the appropriate trace memory location.
0x3130	*BKUPSTP*	This is not a TCB register, but needed on a reset to save the *TCBSTP* value before that register is reset to 0. This allows the software that comes up after a (hard or soft) reset to know the last-known good value of *TCBSTP* before system crash, and potentially read the trace memory from or to the appropriate trace memory location.
0x3200-0x3238	*TCBTrigX*	The *TCBTrigX* set of registers. The number of implemented registers is determined by the value in *TCBCONFIG*$_{TRIG}$. See Section 11.9.11 “TCBTRIGx Register (Reg 16-23)” for more details about register contents.

either not present, or not enabled (i.e., the *ProbEN* bit in the *EJTAG Control* Register or *ECR* is set to zero). If the mappings are active, writes to the TCB registers via drseg are enabled (so long as these writes are otherwise permitted). If the mappings are inactive, writes to the TCB registers via drseg are ignored. Note that a hardware probe could set the *ProbEN* bit to zero and still access the TCBControl registers. Writing the TCB registers via the probe and drseg simultaneously will result in unpredictable behavior. Software should not rely on reads from the TCB registers via drseg to return reliable data when the mappings are inactive. If the mappings are active on reset (i.e., *ProbEN*=0), software is responsible for initializing all control register fields, except for $TCBCONTROLA_{On}$ and $TCBCONTROLB_{En}$.Those control bits are set to zero on a core reset if the drseg mappings are active.

On-chip trace memory can be read by doing a load instruction to *TCBTW*. Accessing the *TCBTW* has the side effect of automatically incrementing the value of *TCBRDP* to the next trace word. The trace memory cannot be written to via this mechanism. Software can also do direct loads and stores to *TCBRDP* and *TCBWRP* at the beginning of the trace memory dump function. Note that writing to these registers in the middle of the trace logic writing into this memory can result in UNPREDICTABLE results and junked values in the trace memory.

Whether or not software has access to on-chip trace memory is controlled via one bit $TCBCONTROLB_{TRPAD}$. This is a control *DISABLE* bit. The bit in *TCBCONTROLB* is mirrored in *TraceControl3*. To access the on-chip memory control registers, namely the memory pointers, the *TCBTW*, and both of the backup pointer bits,*TRPAD* and *ProbEN*, must be zero. To access the other registers, it is sufficient to set the *ProbEN* bit to zero. Regardless of the setting of *ProbEN* and *TRPAD*, all the registers listed in Table 11.29 can be read out by software.

Tracing is stopped when the system crashes and an exception handler is invoked. The last known good values of *TCBRDP*, *TCBWRP*, and *TCBSTP* are saved in the backup registers shown in the table. Software should not rely on *TCBRDP*, *TCBWRP*, and *TCBSTP* holding their last known good values across a reset, and should use the backup registers for this purpose.

11.7.16 Core-Specific Event Inefficiency Tracing

It is possible the trace can relay some hints as to the reason for loss of execution performance. This is done by enabling the core-specific inefficiency tracing via $TraceControl_{bit28}$ / $TCBCONTROLA_{bit28}$. The inefficiency is determined at the same point that the PC, address, data, etc. is captured from the core pipeline. That is, the inefficiency is determined in the last pipestage of the graduation. An inefficieny code applies only when there is no instruction graduating. In other words, the inefficiency replaces an “NI”.

When inefficiency tracing is enabled, the instruction completion indicator in all trace formats that have an instruction completion field will increase from three bits to four bits. Reconstruction software will look for the extra bit, and when the msb of the instruction completion is set to 1’b1, which indicates that the inefficiency code is valid. If the msb of the instruction completion is set to 1’b0, then a valid instruction has graduated and there is no inefficiency.

The inefficiency events defined are the following:

1. Load/store cache miss is the reason for the “NI”.
2. Branch/return misprediction is the reason for the “NI”.
3. Replay of a load consumer, or a branch likely, or a cacheop is the reason for the “NI”.
4. Graduation stall, due to the backpressure of the Load Store Graduation Buffer (LSGB) full or other stall from the core is the reason for the “NI”.

11.7.17 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.This prevents the reconstrution software from incorrectly detecting an end of trace.

11.7.18 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 11.30 and described in Chapter 7, “CP0 Registers of the 74K™ Core” on page 143.

Table 11.30 A List of Coprocessor 0 Trace Registers

Register Number	Sel	Register Name	Reference
23	1	*TraceControl*	Section 7.2.28 “Trace Control Register (CP0 Register 23, Select 1)”
23	2	*TraceControl2*	Section 7.2.29 “Trace Control2 Register (CP0 Register 23, Select 2)”
24	2	*TraceControl3*	Section 7.2.34 “Trace Control3 Register (CP0 Register 24, Select 2)”
23	3	*UserTraceData1*	Section 7.2.30 “User Trace Data1 Register (CP0 Register 23, Select 3) and User Trace Data2 Register (CP0 Register 24, Select 3)”
24	3	*UserTraceData2*	Section 7.2.30 “User Trace Data1 Register (CP0 Register 23, Select 3) and User Trace Data2 Register (CP0 Register 24, Select 3)”
23	4	*TraceIBPC*	Section 7.2.31 “TraceIBPC Register (CP0 Register 23, Select 4)”
23	5	*TraceDBPC*	Section 7.2.32 “TraceDBPC Register (CP0 Register 23, Select 5)”Section 7.2.31 “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 11.31 and Table 11.32. These registers are accessed via the EJTAG TAP interface.

Table 11.31 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 11.32. 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
0x16	*TCBCONTROLE*	Control Register in the TCB used to control tracing for the performance counter tracing feature. See Section 11.9.5 "TCBCONTROLE Register".	Yes

Table 11.32 Registers Selected by TCBCONTROLB

TCBCONTROLB$_{REG}$ Field	Name	Reference	Implemented
0	*TCBCONFIG*	Section 11.9.6 "TCBCONFIG Register (Reg 0)"	Yes
4	*TCBTW*	Section 11.9.7 "TCBTW Register (Reg 4)"	Yes if on-chip memory exists. Otherwise No
5	*TCBRDP*	Section 11.9.8 "TCBRDP Register (Reg 5)"	
6	*TCBWRP*	Section 11.9.9 "TCBWRP Register (Reg 6)"	
7	*TCBSTP*	Section 11.9.10 "TCBSTP Register (Reg 7)"	
16-23	*TCBTRIGx*	Section 11.9.11 "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 11.33.

Figure 11.25 TCBCONTROLA Register Format

31 30	29 27	26	25 24	23	22 20	19	18	17	16	15	14	13	12 5	4	3	2	1	0
SyPExt	Impl	0	VModes	ADW	SyP	TB	IO	D	E	S	K	U	ASID	G	TFCR	TLSM	TIM	On

Table 11.33 TCBCONTROLA Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	**Bits**			
SyPExt	31:30	Extension to the *SyP* (sync period) field for implementations that need higher numbes of cycles between synchronization events. The value of *SyP* is extended by assuming that these two bits are juxtaposed to the left of the three bits of *SyP* (*SypExt*.*SyP*). When only *SyP* was used to specify the synchronization period, the value was 2^x, where xwas computed from *SyP* by adding 5 to the actual value represented by the bits. A similar formula is applied to the 5 bits obtained by the juxtaposition of *SyPExt* and *SyP*. Sync period values greater than 2^{31} are UNPREDICTABLE. That is all values greater than 11010 (26+5=31) are UNPREDICTABLE. With *SyPExt* bits, a sync period range of 2^5 to 2^{31} cycles can be obtained.	R/W	0
Impl	29	Reserved for implementation specific use.	R	0
Ineff	28	Core-specific inefficiency tracing is enabled. If enabled core-specific trace information is included in the trace stream. The inefficiency code replaces an “NI” and is interpreted in the trace stream with an expanded inscomp. The inscomp is expanded from 3b to 4b for all trace formats.	R/W	0
Impl	27	Reserved for implementation specific use.	R	0
0	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: **Encoding** — **Meaning** 00 — PC tracing only 01 — PC and Load and store address tracing only 10 — PC, load and store address, and load and store data. 11 — Reserved This field is preset to the value of *ValidModes*.	R	10
ADW	23	The data value width used in the trace formats. 0: The width is 16 bits wide. 1: The width is is 32 bits wide.	R	1

Table 11.33 TCBCONTROLA Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bits			
SyP	22:20	Used to indicate the synchronization period. The period (in cycles) between which the periodic synchronization information is to be sent is defined as shown in the table below. **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}	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.	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.	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. When set to zero, trace is disabled in Debug mode, irrespective of other bits.	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.	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.	R/W	Undefined
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.	R/W	Undefined

Table 11.33 TCBCONTROLA Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bits			
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.	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.	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.	R/W	Undefined
TFCR	3	When set, this indicates to the PDtrace interface that the optional Fcr bit must be traced in the appropriate trace formats. If PC tracing is disabled, the full PC of the function call (or return) instruction must also be traced. Note that function call/return information is only traced if tracing is actually enabled for the current mode.	R/W	Undefined
TLSM	2	When set, this indicates to the PDtrace interface that information about data cache misses should be traced. If PC, load/store addresses and data tracing are disabled (see *TraceControl*$_{Mode}$ field), the full PC and load/store address are traced for data cache misses. If load/store data tracing is enabled, the *LSm* bit must be traced in the appropriate trace format. Note that data cache miss information is only traced if tracing is actually enabled for the current mode.	R/W	Undefined
TIM	1	When set, this indicates to the PDtrace interface that the optional Im bit must be traced in the appropriate trace formats. If PC tracing is disabled, the full PC of the instruction that missed in the I-cache must be traced. Note that instruction cache miss information is only traced if tracing is actually enabled in the current mode.	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. When set to one, tracing is enabled whenever the other enabling functions are also true.	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 11.34.

Figure 11.26 TCBCONTROLB Register Format

31	30 28	27 26	25 21	20	19	18	17	16	15	14	13 12	11	10 8	7	6 3	2	1	0
WE	0	TWSrcWidth	REG	WR	0	TRPAD	FDT	RM	TR	BF	TM	TLSIF	CR	Cal	TWSrcVal	CA	OfC	EN

Table 11.34 TCBCONTROLB Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
WE	31	Write Enable. Only when set to 1 will the other bits be written in *TCBCONTROLB*. This bit will always read 0.	R	0
0	30:28	Reserved. Must be written as zero; returns zero on read.	R	0
TWSrcWidth	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. **Encoding** / **Meaning**: 00 — Zero source field width 01 — 2- bit source field width 10 — 4- bit source field width 11 — Reserved for future use This field can either be 00, 01,or 10 for the 74K core.	R	00
REG	25:21	Register select: This field select the registers accessible through the *TCBDATA* register. Legal values are shown in Table 11.32.	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	Reserved. Must be written as zero; returns zero on read.	R	0
TRPAD	18	Trace RAM access disable bit, disables program software access to the on-chip trace RAM using load/store instructions. When this bit is set, that is the access is disabled, then software access to the on-chip memory is disabled. If probe access is not provided in the implementation, then this register bit must be tied to zero value to allow software to control access.	R/W	0
FDT	17	Filtered Data Trace Mode enable bit. When the bit is 0, this mode is disabled, reset value is disable. When set to 1, this mode is enabled. This mode is described in Section 11.7.12 "Filtered Data Trace Mode"	R/W	0
RM	16	Read on-chip trace memory. 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. 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.] When the write pointer is reached, this bit is automatically reset to 0, and the *TCBTW* register will read all zeros. 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*. This bit is reserved if on-chip memory is not implemented.	R/W1	0

Table 11.34 TCBCONTROLB Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bits			
TR	15	Trace memory reset. 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. This bit is automatically de-asserted back to 0, when the reset is completed. 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") This bit is cleared when writing 1 to the TR bit This bit is reserved if on-chip memory is not implemented.	R	0
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. **TM** / **Trace Mode**: 00 — Trace-To 01 — Trace-From 10 — Reserved 11 — Reserved 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. In Trace-From mode, the on-chip trace memory is filled from the point that the core starts tracing until the on-chip trace memory is full. In both cases, de-asserting the EN bit in this register will also stop fill to the trace memory. If a *TCBTRIGx* trigger control register is used to start/stop tracing, then this field should be set to Trace-To mode. 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 11.35. **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. This bit is reserved if off-chip trace option is not implemented.	R/W	100

Table 11.34 TCBCONTROLB Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bits			
Cal	7	Calibrate off-chip trace interface. 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. Calibrations pattern 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 This pattern is replicated for every 4 bits of *TR_DATA* pins. **Note:** The clock source of the TCB and PIB must be running. 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. When set to 1, the trace will include stall information. When set to 0, the trace will exclude stall information, and remove bit zero from all transmitted TF's. The stall information included/excluded is: • TF6 formats with TCBcode 0001 and 0101. • All TF1 formats.	R/W	0
OfC	1	If set to 1, trace is sent to off-chip memory using *TR_DATA* pins. If set to 0, trace info is sent to on-chip memory. This bit is read only if a single memory option exists (either off-chip or on-chip only).	R/W	Preset

Table 11.34 TCBCONTROLB Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bits			
EN	0	Enable trace. 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. When set to 1, trace information is sampled on the output pins (capture-to-compression interface) or written into the onchip trace memory. 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. When set to 0, trace information on the output pins are 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 11.35 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*$_{\text{REG}}$ field; see Table 11.32. Regardless of which register or data entry is accessed through *TCBDATA*, the register is only written if the *TCBCONTROLB*$_{\text{WR}}$ bit is set. For read-only registers, the *TCBCONTROLB*$_{\text{WR}}$ is a don't care. If sofware is accessing the onchip trace memory to read out the trace words, then *TCBDATA* is not used for the indirect read of the *TCBTW*. Instead software can read from *TCBTW* directly.

The format of the *TCBDATA* register is shown below, and the field is described in Table 11.36. The width of *TCBDATA* is 64 bits when on-chip trace words (TWs) are accessed (*TCBTW* access).

Figure 11.27 TCBDATA Register Format

31(63) ... 0
Data

Table 11.36 TCBDATA Register Field Descriptions

Fields		Description	Read/Write	Reset State
Names	Bits			
Data	31:0 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 11.37.

Figure 11.28 TCBCONTROLC Register Format

30 30	29 28	27 23	22	21 15	14	13	12 9	8 5	4 2	1	0
Res	NumDO	Mode	Res	Res	Res	Res	Res	Res	Res	Res	Res

Table 11.37 TCBCONTROLC Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
Res	31:30	Reserved for future use. Must be written as zero; returns zero on read.	0	0
NumDO	29:28	Specifies the number of bits needed by this implementation to specify the DataOrder: 00 - Four bits 01 - Five bits 10 - Six bits 11 - Eight bits	R	Preset
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. **Bit # Set** / **Trace The Following** 0 - PC 1 - Load address 2 - Store address 3 - Load data 4 - Store data 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.	R/W	0

Table 11.37 TCBCONTROLC Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bits			
Res	22:0	Reserved for future use. Must be written as zero; returns zero on read.	0	0

11.9.5 TCBCONTROLE 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, *TCBCONTROLE*, 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 *TCBCONTROLE* register.

The *TCBCONTROLE* register is written by an EJTAG TAP controller instruction, *TCBCONTROLE* (0x16).

The format of the *TCBCONTROLE* register is shown below, and the fields are described in Table 11.38.

Figure 11.29 TCBCONTROLE Register Format

31 9	8	7 6	5	4	3	2	1	0
0	TdIDLE	0	PecOvf	PeCFCR	PeCBP	PeCSync	PeCE	PeC

Table 11.38 TCBCONTROLE Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:9	Reserved for future use. Must be written as zero; returns zero on read.	0	0
TrIDLE	8	Trace Unit Idle. This bit indicates if the trace hardware is currently idle (not processing any data). This can be useful when switching control of trace from hardware to software and vice versa. The bit is read-only and updated by the trace hardware.	R	1
0	7:6	Reserved for future use; Must be written as zero; returns zero on read. (Hint to architect, Reserved for future expansion of performance counter trace events).	0	0
PeCOvf	5	Trace performance counters when one of the performance counters overflows its count value. Enabled when set to 1.	R/W	0
PeCFCR	4	Trace performance counters on function call/return or on an exception handler entry. Enabled when set to 1.	R/W	0
PeCBP	3	Trace performance counters on hardware breakpoint match trigger. Enabled when set to 1.	R/W	0
PeCSync	2	Trace performance counters on synchronization counter expiration. Enabled when set to 1.	R/W	0

Table 11.38 TCBCONTROLE Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bits			
PeCE	1	Performance counter tracing enable. When set to 0, the tracing out of performance counter values as specified is disabled. To enable, this bit must be set to 1. This bit is used under software control. When trace is controlled by an external probe, this enabling is done via the *TCB Control* register.	R/W	0
PeC	0	Specifies whether or not Performance Control Tracing is implemented. This is an optional feature that may be omitted by implementation choice. See Section 11.7.11 “Performance Counter Tracing” for details.	R	Preset

11.9.6 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 11.39.

Figure 11.30 TCBCONFIG Register Format

31	30 25	24 21	20 17	16 14	13 11	10 9	8 6	5	4	3 0
CF1	0	TRIG	SZ	CRMax	CRMin	PW	PiN	OnT	OfT	REV

Table 11.39 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 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. The size in bytes is given by $2^{(SZ+8)}$, implying that the minimum size is 256 bytes and the largest is 8Mb. This bit is reserved if on-chip memory is not implemented.	R	Preset
CRMax	16:14	Off-chip Maximum Clock Ratio. 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 11.35. This bit is reserved if off-chip trace option is not implemented.	R	Preset

Table 11.39 TCBCONFIG Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bits			
CRMin	13:11	Off-chip Minimum Clock Ratio. 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 11.35. 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.	R	Preset

PW	Number of bits used on TR_DATA
00	4 bits
01	8 bits
10	16 bits
11	Reserved

Name	Bits	Description	Read / Write	Reset State
		This field is preset based on input signals to the TCB and the actual capability of the TCB. This bit is reserved if off-chip trace option is not implemented.		
PiN	8:6	Pipe number. 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 4.1 must has a value of 1 for this field.	R	1

11.9.7 TCBTW Register (Reg 4)

The *TCBTW* register is used to read Trace Words from the on-chip trace memory. The Trace Word (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. The *TCBTW* register is mapped to offset 0x3100 in drseg. An access to offset 0x3100 automatically causes the read pointer to be incremented. The use of load half-word or load byte instructions can lead to unpredictable results, and is not recommended. The results of attempting to write to trace memory by an explicit store instruction targeting *TCBTW* are unpredictable.

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 11.40.

Figure 11.31 TCBTW Register Format

63 … 0
Data

Table 11.40 TCBTW Register Field Descriptions

Fields		Description	Read / Write	Reset State
Names	Bits			
Data	63:0	Trace Word (TW)	R/W	0

11.9.8 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 11.41. 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 TCBRDP Register Format

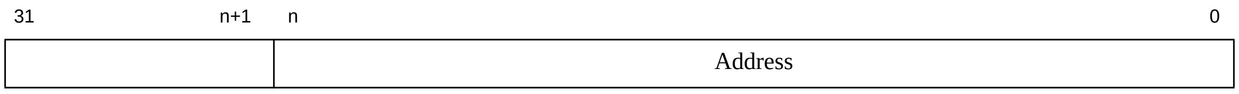


Table 11.41 TCBRDP Register Field Descriptions

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

11.9.9 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 11.42. 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 11.33 TCBWRP Register Format

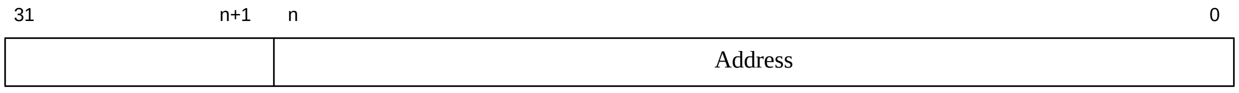


Table 11.42 TCBWRP Register Field Descriptions

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

11.9.10 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 11.43. 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 11.34 TCBSTP Register Format

31 n+1	n 0
0	Address

Table 11.43 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

11.9.11 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*$_{\text{TRIG}}$ 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 11.44.

Figure 11.35 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 11.44 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. 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. The write value of this bit always controls the behavior of this trigger. 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
CHTri	5	When set, this Trigger will fire when a rising edge on *TC_ChipTrigIn* is detected. The write value of this bit always controls the behavior of this trigger. 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. The write value of this bit always controls the behavior of this trigger. 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 11.44 TCBTRIGx Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Names	**Bits**			
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. **Type** / **Trigger action**: 00 — **Trigger Start:** Trigger start-point of trace. 01 — **Trigger End:** Trigger end-point of trace. 10 — **Trigger About:** Trigger center-point of trace. 11 — **Trigger Info:** No action trigger, only for trace info. 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. 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. 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. The write value of this bit always controls the behavior of this trigger. 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
TR	0	Trigger happened. When set, this trigger fired since the *TR* bit was last written 0. This bit is used to inspect whether the trigger fired since this bit was last written zero. 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. 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

11.9.12 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 74K 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 sections Section 7.2.31 “TraceIBPC Register (CP0 Register 23, Select 4)” and Section 7.2.32 “TraceDBPC Register (CP0 Register 23, Select 5)” for details on 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
    (
        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
    ($TCBCONTROLA_G$ 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:

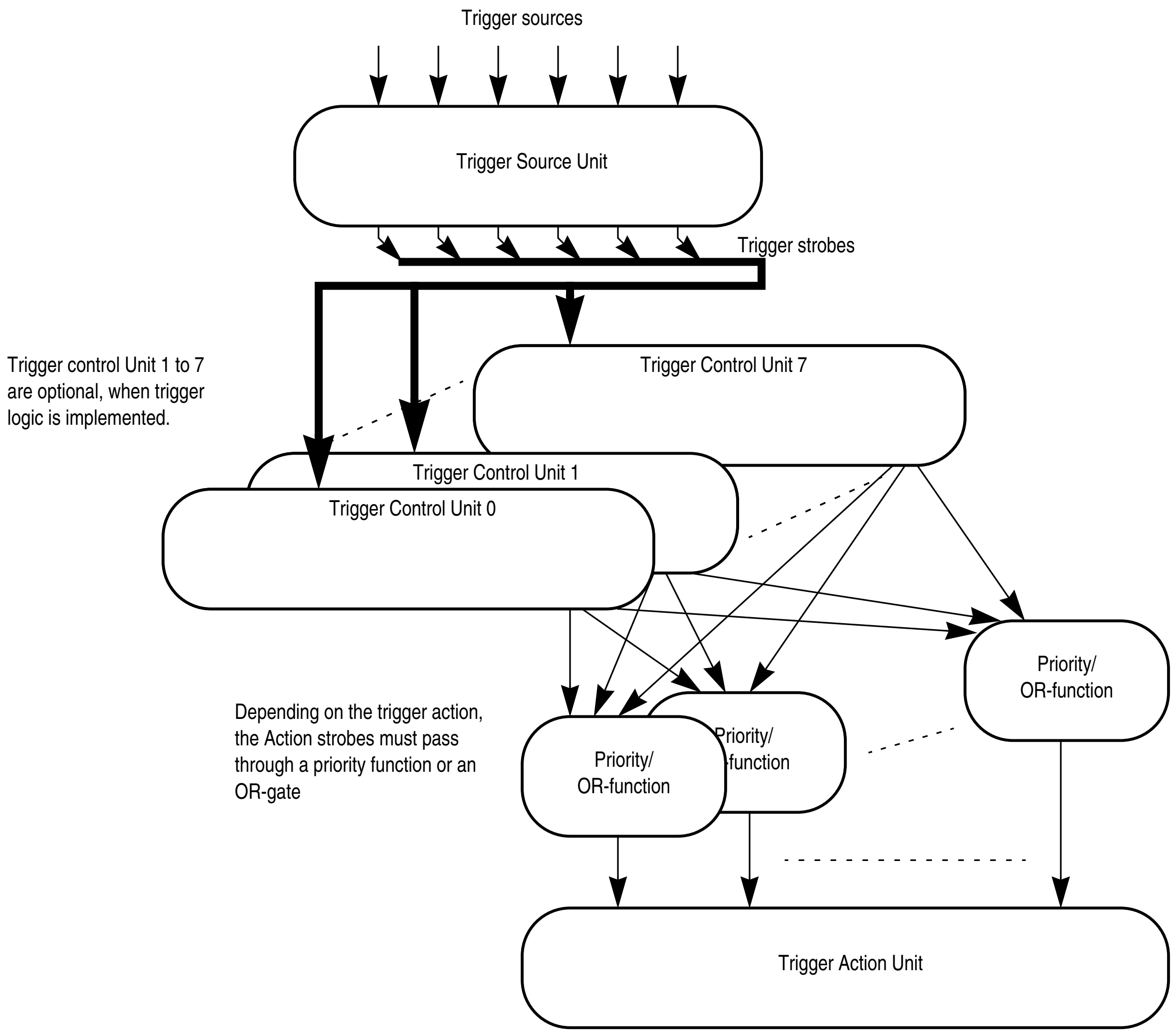
1. A common Trigger Source detection unit.

2. 1 to 8 separate Trigger Control units.

3. A common Trigger Action unit.

Figure 11.36 show the functional overview of the trigger flow in the TCB.

Figure 11.36 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 11.9.11 "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 11.9.11 "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 capture module and the TCB module contain a FIFO. This might seem like extra overhead, but there are good reasons for this. All the information that needs to be captured from the core pipeline is stored in a structure called the Unified Fifo (UFIFO). The Unified Fifo holds PC, load/store address values (delta or full), load/store data, processor mode changes (which are in the form of a TMOAS message), and performance counter data. The capture module reads out two UFIFO entries per cycle and sends to the TCB. The two transactions are translated into two PDtrace™ defined trace formats. These trace messages are then fed to the compression datapath. The compression datapath generates a PDtrace defined trace word.To keep the throughput, the compression datapath is wide enough to sustain the generation of upto two trace words per cycle. The trace words are stored in a buffer called the TraceWord Fifo (TWFIFO). One trace word is read out from the TWFIFO and sent to the offchip memory interface, or upto 2 trace words are read out per cycle and sent to the onchip trace memory. The buffer will advance to the next trace word, once a read acknowledge is received from the onchip or offchip memory interface.

In the TCB, the on-chip trace memory is defined as a 128-bit wide synchronous memory running at core-clock speed. In this case the TWFIFO needs only four entries to guarantee it will not overflow. The TWFIFO could be filled in such a way that only 64bits of the 128bits is written to memory, thus the four entry requirement. For off-chip trace going through the Trace Probe, the FIFO is much more important, due to a limited number of pins (4, 8 or 16) between the Probe and actual memory. Also the speed of the Trace Probe interface can be different (either faster or slower) from that of the 74K core. So for off-chip tracing, a deeper TCB TWFIFO is desirable.

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 *TraceControlIO* 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. In 74K core, the UFifo has 64 entries, while TWFIFO has 11 entries. Almost always the TWFIFO will fill up first and be the cause of the back-stall to the core pipeline.

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

As mentioned earlier, the buffer in the TCB (TWFIFO) is 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 TWFIFO 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 are two options:

1. Allow the overflow to happen, and thereby lose some information from the trace data.

2. Prevent the overflow by asserting a stall-signal back to the core. 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 unsupported 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 11.35 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 TWFIFO and not the PDtrace UFIFO. The extra bit therefore only affects the likelihood of the TCB TWFIFO 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.

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 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

Instruction Set Overview

This chapter provides an overview of the 74K™ core instruction set, including the instruction formats and the basic instruction types.

This chapter discusses the following topics:

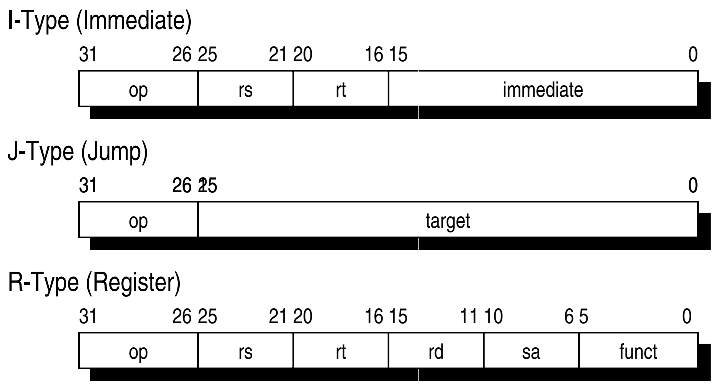
- Section 12.1 "CPU Instruction Formats"
- Section 12.2 "Load and Store Instructions"
- Section 12.3 "Computational Instructions"
- Section 12.4 "Jump and Branch Instructions"
- Section 12.5 "Control Instructions"
- Section 12.6 "Coprocessor Instructions"

Refer to Chapter 13, "74K™ Processor Core Instructions" on page 303 for a complete listing and description of those instructions whose behavior differs in the 74K processor core. The complete MIPS32 instruction set is described in Volume II of the MIPS32® Architecture For Programmers.

12.1 CPU Instruction Formats

A CPU instruction consists of a single 32-bit word, aligned on a word boundary. There are three instruction formats: immediate (I-type), jump (J-type), and register (R-type). The use of a small number of instruction formats simplifies instruction decoding, allowing the compiler to synthesize more complicated (and less frequently used) operations and addressing modes from these three formats as needed. The instruction formats are shown in Figure 12.1.

Figure 12.1 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

12.2 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.

12.2.1 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 74K 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.

12.2.2 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, defines the bytes accessed within the addressed word, as shown in Table 12.1. Only the combinations shown in Table 12.1 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 in the same way as load/store references of those sizes.

Table 12.1 Byte Access Within a Doubleword

	Low-Order Address Bits			Bytes Accessed															
				Big Endian (63----------------31-------------------0)								Little Endian (63----------------31-------------------0)							
Access Type	**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							

12.3 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

12.3.1 Cycle Timing for Multiply and Divide Instructions

Multiply instruction in the integer pipeline are transferred to the multiplier while 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 74K™ Core” on page 37 for more information on instruction latency and repeat rates.

12.4 Jump and Branch Instructions

Jump and branch instructions change the control flow of a program. All jump and branch instructions occur with a delay of one instruction: that is, the instruction immediately following the jump or branch (the instruction in the so-called *delay slot*) always executes while the target instruction is being fetched from storage.

12.4.1 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 use 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*.

12.4.2 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.

12.5 Control Instructions

Control instructions allow the software to initiate traps; they are always R-type.

12.6 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 13, “74K™ Processor Core Instructions” on page 303 for a listing of CP0 instructions.

Chapter 13

74K™ Processor Core Instructions

This chapter supplements the *MIPS32® Architecture Reference Manual* by describing instruction behavior that is specific to the 74K processor core. The chapter contains the following sections:

- Section 13.1 "Understanding the Instruction Descriptions"
- Section 13.2 "74K™ Opcode Map"
- Section 13.3 "Floating Point Unit Instruction Format Encodings"
- Section 13.4 "MIPS32™ Instruction Set for the 74K™ Core"

The 74K processor core also supports the instructions in the MIPS DSP ASE Revision 2 and the MIPS16e ASE. The MIPS DSP ASE Revision 2 instruction set is described in Chapter 4, "The MIPS® DSP Application-Specific Extension to the MIPS32® Instruction Set" on page 81. The MIPS16e ASE instruction set is described in Chapter 14, "MIPS16e™ Application-Specific Extension to the MIPS32® Instruction Set" on page 343.

13.1 Understanding the Instruction Descriptions

Refer to *Volume II* of the *MIPS32® Architecture Reference Manual* for more information about the instruction descriptions. That document contains a description of the instruction fields, a definition of terms, and a description function notation.

13.2 74K™ Opcode Map

Table 13.1 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 are 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 13.2 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 13.3 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 13.4 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 13.5 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 13.6 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 13.7 MIPS32 *MOVCI* Encoding of tf Bit

tf	*bit 16*	
	0	1
	MOVF	MOVT

Table 13.8 MIPS32 *SRL* Encoding of Shift/Rotate

tf	*bit 21*	
	0	1
	SRL	ROTR

Table 13.9 MIPS32 *SRLV* Encoding of Shift/Rotate

tf	*bit 6*	
	0	1
	SRLV	ROTRV

Table 13.10 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 13.11 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 13.12 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 13.13 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 13.14 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 13.15 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 13.16 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 13.17 MIPS32 *COP1* Encoding of tf Bit When rs=*S or D,* Function=*MOVCF*

tf	*bit 16*	
	0	1
	MOVF.fmt	MOVT.fmt

Table 13.18 *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 ∇	*	*	*	*	*	*
7	111	NMSUB.S ∇	NMSUB.D ∇	*	*	*	*	*	*

1. COP1X instructions are legal only if 64-bit floating point operations are enabled.

Table 13.19 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								

13.3 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 13.13 and Table 13.18 above.

Table 13.20 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
0..15	00..0F	—	—	Used to encode Coprocessor 1 interface instructions (MFC1, CTC1, etc.). Not used for format encoding.			
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	2×32	Floating Point

Table 13.20 Floating Point Unit Instruction Format Encodings (Continued)

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**
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.			

13.4 MIPS32™ Instruction Set for the 74K™ Core

This section describes the MIPS32 instructions for the 74K cores. Table 13.21 lists the instructions in alphabetical order. Following the table, the instructions that have implementation-dependent behavior in the 74K core are described individually. The descriptions of other instructions that exist in the*MIPS32® Architecture Reference Manual* are not duplicated here.

Table 13.21 74K™ Core Instruction Set

Instruction	Description	Function
`ADD`	Integer Add	`Rd = Rs + Rt`
`ADDI`	Integer Add Immediate	`Rt = Rs + Immed`
`ADDIU`	Unsigned Integer Add Immediate	Rt = Rs $+_U$ Immed
`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 (Assembler idiom for: BEQ r0, r0, offset)	`PC += (int)offset`

`BAL`	Branch and Link
	(Assembler idiom for: BGEZAL r0, offset)
		`GPR[31] = PC + 8`
		`PC += (int)offset`

`BC2F`	Branch On Cp2 False	`if (cc[i] == 0) then`
		`   PC += (int)offset`

`BC2FL`	Branch On Cp2 False Likely	`if (cc[i] == 0)then`
		`  PC += (int)offset`
		`else`
		`  Ignore Next Instruction`

`BC2T`	Branch On Cp2True	`if(cc[i] == 1) then`
		`  PC += (int)offset`

`BC2TL`	Branch On Cp2 True Likely	`if (cc[i] == 1) then`
		`  PC += (int)offset`
		`else`
		`  Ignore Next Instruction`

Table 13.21 74K™ Core Instruction Set (Continued)

Instruction	Description	Function

`BEQ`	Branch On Equal	`if Rs == Rt`
		`  PC += (int)offset`

`BEQL`	Branch On Equal Likely	`if Rs == Rt`
		`  PC += (int)offset`
		`else`
		`  Ignore Next Instruction`

`BGEZ`	Branch on Greater Than or Equal To Zero	`if !Rs[31]`
		`  PC += (int)offset`

`BGEZAL`	Branch on Greater Than or Equal To Zero And Link	`GPR[31] = PC + 8`
		`if !Rs[31]`
		`  PC += (int)offset`

`BGEZALL`	Branch on Greater Than or Equal To Zero And Link Likely	`GPR[31] = PC + 8`
		`if !Rs[31]`
		`  PC += (int)offset`
		`else`
		`  Ignore Next Instruction`

`BGEZL`	Branch on Greater Than or Equal To Zero Likely	`if !Rs[31]`
		`  PC += (int)offset`
		`else`
		`  Ignore Next Instruction`

`BGTZ`	Branch on Greater Than Zero	`if !Rs[31] && Rs != 0`
		`  PC += (int)offset`

`BGTZL`	Branch on Greater Than Zero Likely	`if !Rs[31] && Rs != 0`
		`  PC += (int)offset`
		`else`
		`  Ignore Next Instruction`

`BLEZ`	Branch on Less Than or Equal to Zero	`if Rs[31] || Rs == 0`
		`  PC += (int)offset`

`BLEZL`	Branch on Less Than or Equal to Zero Likely	`if Rs[31] || Rs == 0`
		`  PC += (int)offset`
		`else`
		`  Ignore Next Instruction`

`BLTZ`	Branch on Less Than Zero	`if Rs[31]`
		`  PC += (int)offset`

`BLTZAL`	Branch on Less Than Zero And Link	`GPR[31] = PC + 8`
		`if Rs[31]`
		`  PC += (int)offset`

`BLTZALL`	Branch on Less Than Zero And Link Likely	`GPR[31] = PC + 8`
		`if Rs[31]`
		`  PC += (int)offset`
		`else`
		`  Ignore Next Instruction`

`BLTZL`	Branch on Less Than Zero Likely	`if Rs[31]`
		`  PC += (int)offset`
		`else`
		`  Ignore Next Instruction`

`BNE`	Branch on Not Equal	`if Rs != Rt`
		`  PC += (int)offset`

Table 13.21 74K™ Core Instruction Set (Continued)

Instruction	Description	Function

`BNEL`	Branch on Not Equal Likely	`if Rs != Rt`
		`  PC += (int)offset`
		`else`
		`  Ignore Next Instruction`

`BREAK`	Breakpoint	Break Exception
`CACHE`	Cache Operation	See the description of the CACHE instruction on page 317.
`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
`CTC2`	Move Control Word to Cp2	`Cp2 Control[Fs] = Rt`

`DERET`	Return from Debug Exception	`PC = DEPC`
		`Exit Debug Mode`

`DI`	Atomically Disable Interrupts	Rt = Status; $Status_{IE}$ = 0

`DIV`	Divide	`LO = (int)Rs / (int)Rt`
		`HI = (int)Rs % (int)Rt`

`DIVU`	Unsigned Divide	`LO = (uns)Rs / (uns)Rt`
		`HI = (uns)Rs % (uns)Rt`

`EHB`	Execution Hazard Barrier	Stop instruction execution until execution hazards are cleared
`EI`	Atomically Enable Interrupts	Rt = Status; $Status_{IE}$ = 1

`ERET`	Return from Exception	`if SR[2]`
		`  PC = ErrorEPC`
		`else`
		`  PC = EPC`
		`  SR[1] = 0`
		`SR[2] = 0`
		`LL = 0`

`EXT`	Extract Bit Field	`Rt = ExtractField(Rs, pos, size)`
`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`
		`PC = PC[31:28] || offset<<2`

`JALR`	Jump and Link Register	`Rd = PC + 8`
		`PC = Rs`

Table 13.21 74K™ Core Instruction Set (Continued)

Instruction	Description	Function
`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`
		`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]`
`LDC2`	Load Doubleword to Cp2	`Ft = memory[base+offset]`
`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]`
		`LL = 1`
		See also the description of the LL instruction on page 325.

`LUI`	Load Upper Immediate	`Rt = immediate << 16`
`LW`	Load Word	`Rt = Mem[Rs+offset]`
`LWC2`	Load Word to Cp2	`Ft = memory[base+offset]`
`LWPC`	Load Word, PC relative	`Rt = Mem[PC+offset]`
`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`
`MADDU`	Multiply-Add Unsigned	`HI | LO += (uns)Rs * (uns)Rt`
`MFC0`	Move From Coprocessor 0	`Rt = CPR[0, Rd, sel]`
`MFC2`	Move From Cp2 Register	$Rt = Fs_{31..0}$
`MFHC2`	Move From High Half of Cp2 Register	$Rt = Fs_{63..32}$
`MFHI`	Move From HI	`Rd = HI`
`MFLO`	Move From LO	`Rd = LO`

`MOVN`	GPR Conditional Move on Not Zero	`if Rt ≠ 0 then`
		`  Rd = Rs`

`MOVZ`	GPR Conditional Move on Zero	`if Rt = 0 then`
		`  Rd = Rs`

Table 13.21 74K™ Core Instruction Set (Continued)

Instruction	Description	Function
MSUB	Multiply-Subtract	HI | LO -= (int)Rs * (int)Rt
MSUBU	Multiply-Subtract Unsigned	HI | LO -= (uns)Rs * (uns)Rt
MTC0	Move To Coprocessor 0	CPR[0, n, Sel] = Rt
MTC2	Move to Cp2 register	Fs = Rt
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
MUL	Multiply with register write	HI | LO =Unpredictable Rd = $((int)Rs * (int)Rt)_{31..0}$
MULT	Integer Multiply	HI | LO = (int)Rs * (int)Rd
MULTU	Unsigned Multiply	HI | LO = (uns)Rs * (uns)Rd
NOP	No Operation (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. See also the description of the PREF instruction on page 327.
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]
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 = $Rt_{Rs-1..0}$ || $Rt_{31..Rs}$
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
		   mem[base+offset] = Rt
		Rt = LL
		See also the description of the SC instruction on page 331.

SDBBP	Software Debug Break Point	Trap to SW Debug Handler
SDC2	Store Doubleword from Cp2	memory[base+offset] = Ft

Table 13.21 74K™ Core Instruction Set (Continued)

Instruction	Description	Function
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`
		`  Rd = 1`
		`else`
		`  Rd = 0`

SLTI	Set on Less Than Immediate	`if (int)Rs < (int)Immed`
		`  Rt = 1`
		`else`
		`  Rt = 0`

SLTIU	Set on Less Than Immediate Unsigned	`if (uns)Rs < (uns)Immed`
		`  Rt = 1`
		`else`
		`  Rt = 0`

SLTU	Set on Less Than Unsigned	`if (uns)Rs < (uns)Immed`
		`  Rd = 1`
		`else`
		`  Rd = 0`

SRA	Shift Right Arithmetic	`Rd = (int)Rt >> sa`
SRAV	Shift Right Arithmetic Variable	`Rd = (int)Rt >> Rs[4:0]`
SRL	Shift Right Logical	`Rd = (uns)Rt >> sa`
SRLV	Shift Right Logical Variable	`Rd = (uns)Rt >> Rs[4:0]`
SSNOP	Superscalar Inhibit No Operation	`NOP`
SUB	Integer Subtract	`Rt = (int)Rs - (int)Rd`
SUBU	Unsigned Subtract	`Rt = (uns)Rs - (uns)Rd`
SW	Store Word	`Mem[base+offset] = Rt`
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
SYNC	Synchronize	See the description of the SYNC instruction on page 333.
SYNCI	Synchronize Caches to Make Instruction Writes Effective	For D-cache writeback and I-cache invalidate on specified address
SYSCALL	System Call	`SystemCallException`

Table 13.21 74K™ Core Instruction Set (Continued)

Instruction	Description	Function

`TEQ`	Trap if Equal	`if Rs == Rt`
		`  TrapException`

`TEQI`	Trap if Equal Immediate	`if Rs == (int)Immed`
		`  TrapException`

`TGE`	Trap if Greater Than or Equal	`if (int)Rs >= (int)Rt`
		`  TrapException`

`TGEI`	Trap if Greater Than or Equal Immediate	`if (int)Rs >= (int)Immed`
		`  TrapException`

`TGEIU`	Trap if Greater Than or Equal Immediate Unsigned	`if (uns)Rs >= (uns)Immed`
		`  TrapException`

`TGEU`	Trap if Greater Than or Equal Unsigned	`if (uns)Rs >= (uns)Rt`
		`  TrapException`

`TLBWI`	Write Indexed TLB Entry	See the description of the TLBWI instruction on page 337.
`TLBWR`	Write Random TLB Entry	See the description of the TLBWR instruction on page 341.
`TLBP`	Probe TLB for Matching Entry	See Software Users Manual
`TLBR`	Read Index for TLB Entry	See the description of the TLBR instruction on page 335.

`TLT`	Trap if Less Than	`if (int)Rs < (int)Rt`
		`  TrapException`

`TLTI`	Trap if Less Than Immediate	`if (int)Rs < (int)Immed`
		`  TrapException`

`TLTIU`	Trap if Less Than Immediate Unsigned	`if (uns)Rs < (uns)Immed`
		`  TrapException`

`TLTU`	Trap if Less Than Unsigned	`if (uns)Rs < (uns)Rt`
		`  TrapException`

`TNE`	Trap if Not Equal	`if Rs != Rt`
		`  TrapException`

`TNEI`	Trap if Not Equal Immediate	`if Rs != (int)Immed`
		`  TrapException`

`WAIT`	Wait for Interrupts	Stall until interrupt occurs. See the description of the WAIT instruction on page 339.
`WRPGPR`	Write to GPR in Previous Shadow Set	$\texttt{SGPR[SRSCtl}_{\texttt{PSS}}\texttt{, Rd] = Rt}$
`WSBH`	Word Swap Bytes Within HalfWords	$\texttt{Rd = Rt}_{23..16}$ || $\texttt{Rt}_{31..24}$ || $\texttt{Rt}_{7..0}$ || $\texttt{Rt}_{15..8}$
`XOR`	Exclusive OR	`Rd = Rs ^ Rt`
`XORI`	Exclusive OR Immediate	`Rt = Rs ^ (uns)Immed`
`ZEB`	Zero extend byte (MIPS16 only)	`Rt = (ubyte) Rs`

Table 13.21 74K™ Core Instruction Set (Continued)

Instruction	Description	Function
`ZEH`	Zero extend half (MIPS16 only)	`Rt = (uhalf) Rs`

31 26	25 21	20 16	15 0
CACHE 101111	base	op	offset
6	5	5	16

Format: `CACHE op, offset(base)` **MIPS32**

Purpose: Perform Cache Operation

To perform the cache operation specified by op.

Description:

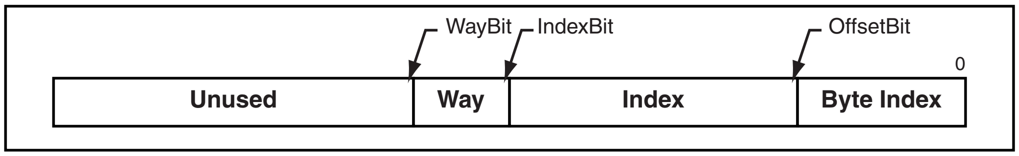
The 16-bit offset is sign-extended and added to the contents of the base register to form an effective address. The effective address is used in one of the following ways based on the operation to be performed and the type of cache as described in the following table.

Table 13.22 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.
		
		Assuming that the total cache size in bytes is CS, the associativity is A, and the number of bytes per tag is BPT, the following calculations give the fields of the address which specify the way and the index:
		
		`OffsetBit ← Log2(BPT)`
		`IndexBit ← Log2(CS / A)`
		`WayBit ← IndexBit + Ceiling(Log2(A))`
		Way ← $Addr_{WayBit-1..IndexBit}$
		Index ← $Addr_{IndexBit-1..OffsetBit}$

Figure 13.1 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 must 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 13.23 Encoding of Bits[17:16] of CACHE Instruction

Code	Name	Cache	Cop0 Registers Used
2#00	I	Primary Instruction	*ITagLo*, *ITagHi*, *IDataLo*, *IDataHi*, *ErrCtl*
2#01	D	Primary Data	*DTagLo*, *DTagHi*, *DDataLo*, *ErrCtl*
2#10	T	Tertiary - Not supported	
2#11	S	Secondary	*L2TagLo*

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 13.23 lists which registers are used by operations to each 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 *DDataLo* (primary D), *L23DataLo* and *L23DataHi* (secondary), or *IDataLo* and *IDataHi* (primary I) registers. All other cache instructions are line-based and the word and byte indexes will not affect their operation

Table 13.24 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST, DYT, SPR] Cleared

Code	Caches	Name	Effective Address Operand Type	Operation	
2#000	I	Index Invalidate	Index	Set the state of the cache line at the specified index to invalid. This encoding may be used by software to invalidate the entire instruction cache by stepping through all valid indices.	Yes
	D, S	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. 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 13.24 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST, DYT, SPR] Cleared (Continued)

Code	Caches	Name	Effective Address Operand Type	Operation	
2#001	I	Index Load Tag	Index	• Read the tag for the cache line at the specified index into the *ITagLo* and *ITagHi* registers. • Read the data corresponding to the dwordindex into the *IDataLo* and *IDataHi* registers. • If parity is implemented, read the parity bits corresponding to the data into $ErrCtl_{PI}$	Yes
2#001	D	Index Load Tag	Index	• Read the tag for the cache line at the specified index into the *DTagLo* Coprocessor 0 register. • Read the data corresponding to the word index into the *DDataLo* register. • 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 *L23TagLo* Coprocessor 0 register. • Read the data corresponding to the dword index into the *L23DataLo* and *L23DataHi* registers.	Yes
2#010	I	Index Store Tag	Index	• Write the tag for the cache block at the specified index from the *ITagLo* and *ITagHi* registers. • If parity is implemented, the parity written into the cache is generated by the hardware if $ErrCtl_{PO}$ = 0, or it is obtained from *ITagLo and ITagHi* if $ErrCtl_{PO}$ = 1.	Yes
2#010	D,S	Index Store Tag	Index	Write the tag for the cache line at the specified index from the associated *TagLo* Coprocessor 0 register. By default, the tag parity value will be automatically calculated. For test purposes, the parity/ECC bits from the associated *TagLo* register will be used if $ErrCtl_{PO}$ is set. 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	Reserved	Unspecified	Executed as a no-op	No

Table 13.24 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST, DYT, SPR] Cleared (Continued)

Code	Caches	Name	Effective Address Operand Type	Operation	
2#011	S	Index Store Data	Index	Write the *L23DataHi* and L23*DataLo* Coprocessor 0 register contents at the way and dword index specified. The ECC bits are always generated by the hardware (if present)	Yes
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. The cache line is refetched even if it is already in the cache. In that case, the existing copy in the cache is invalidated	Yes
	D, S	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. 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	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. 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 *associated TagLo* register. 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 13.25 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST] Set, ErrCtl[DYT, SPR] Cleared

Code	Caches	Name	Effective Address Operand Type	Operation	
2#001	I	Index Load WS	Index	Read the WS RAM at the specified index into the *ItagLo* Coprocessor 0 register.	Yes
2#001	D,S	Index Load WS	Index	Read the WS RAM at the specified index into the *associated TagLo* Coprocessor 0 register.	Yes
2#010	I	Index Store WS	Index	Update the WS RAM at the specified index from the *ITagLo* Coprocessor 0 register.	Yes
2#010	D	Index Store WS	Index	Update the WS RAM at the specified index from the *DTagLo* Coprocessor 0 register.	Yes
2#010	S	Index Store WS	Index	Update the WS RAM at the specified index from the *L23TagLo* Coprocessor 0 register. If $ErrCtl_{PO}$ is set, the dirty parity values in the *L23TagLo* 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 *IDataLo* and *IDataHi* Coprocessor 0 register contents at the way and dword index specified. If $ErrCtl_{PO}$ is set, $ErrCtl_{PI}$ is used for the parity value. Otherwise, the parity value is calculated for the write data. In addition, the precode value for the write data is also updated in the tag if $ErrCtl_{PCD}$ is not set. If $ErrCt_{IPCO}$ is set, *ITagHi* is used for the precode value and its corresponding parity bit. Otherwise, the precode value and its corresponding parity bit are calculated based on the write data.	Yes
2#011	D	Index Store Data	Index	Write the *DDataLo* Coprocessor 0 register contents at the way and word index specified. 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 *L23DataLo* Coprocessor 0 register contents to the *ECC* bits at the way and dword index specified.	Yes
All Oth-ers	All			Other codes should not be used while $ErrCtl_{WST}$ is set.	

Table 13.26 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[SPR] Set, ErrCtl[DYT, WST] Cleared

Code	Caches	Name	Effective Address Operand Type	Operation	
2#001	D	Index Load Tag	Index	Read the SPRAM tag at the specified index into the *TagLo1* Coprocessor 0 register.	Yes
2#010	D	Index Store Tag	Index	Update the SPRAM tag at the specified index from the *TagLo* Coprocessor 0 register.	Yes
2#011	D	Index Store Data	Index	Write the *DDataLo* Coprocessor 0 register contents into the SPRAM at the word index specified. The data parity is always calculated in this case.	Yes
All Others	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.	

Note: *ErrCtl*$_{SPR}$ is a don’t care for cache operations to I-cache.

Table 13.27 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[DYT] Set, ErrCtl[SPR, WST] Cleared

Code	Caches	Name	Effective Address Operand Type	Operation	
2#001	D	Index Load Tag	Index	Read the dirty RAM at the specified index into the *TagLo1* Coprocessor 0 register.	Yes
2#010	D	Index Store Tag	Index	Update the dirty RAM at the specified index from the *TagLo1* Coprocessor 0 register.	Yes
All Others	D			Other codes should not be used while *ErrCtl*$_{DYT}$ is set.	

Note: *ErrCtl*$_{DYT}$ is a don’t care for cache operations to I-cache.

Restrictions:

The operation of this instruction is **UNDEFINED** for any operation/cache combination that is not implemented.

The operation of this instruction is **UNDEFINED** if the 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

Programming Notes:

For cache operations that require an index, it is implementation dependent whether the effective address or the translated physical address is used as the cache index. Therefore, the index value should always be converted to a kseg0 address by ORing the index with 0x80000000 before being used by the cache instruction. For example, the following code sequence performs a data cache Index Store Tag operation using the index passed in GPR a0:

```
   li     a1, 0x80000000          /* Base of kseg0 segment */
   or     a0, a0, a1              /* Convert index to kseg0 address */
   cache  DCIndexStTag, 0(a1)     /* Perform the index store tag operation */
```

31–26	25–21	20–16	15–0
LL 110000	base	rt	offset
6	5	5	16

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 is **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 non-zero, an Address Error exception occurs.

Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if $vAddr_{1..0} \neq 0^2$ then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
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.

31 26	25 21	20 16	15 0
PREF 110011	base	hint	offset
6	5	5	16

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 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.

Table 13.28 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). Action: Fetch data as if for a load.
1	store	Use: Prefetched data is expected to be stored or modified. Action: Fetch data as if for a store.
2-3	Reserved	Reserved - treated as a NOP.
4	load_streamed	Use: Prefetched data is expected to be read (not modified) but not reused extensively; it “streams” through cache. Action: Fetch data as if for a load. 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, data will be placed in the way pointed to by the LRU.

Table 13.28 Values of *hint* Field for PREF Instruction

5	store_streamed	Use: Prefetched data is expected to be stored or modified but not reused extensively; it "streams" through cache. Action: Fetch data as if for a store. 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, data will be placed in the way pointed to by the LRU.
6	load_retained	Use: Prefetched data is expected to be read (not modified) and reused extensively; it should be "retained" in the cache. Action: Fetch data as if for a load. 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. 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")	Action: Schedule a writeback of any dirty data. The cache line is marked as invalid upon completion of the writeback. If cache line is locked, no action is taken. If a line is clean, it will be marked invalid and there will be no writeback scheduled.
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. 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. Programming Note: Because the cache line is filled with zero data 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	Fast Prepare For Store	Use: Prepare the cache for writing an entire line. No data is filled into the line. Action: If reference hits in the cache, no action is taken. If reference misses in the cache, a line is selected for replacement, any valid and dirty victim is written back to memory, the line is validated by writing the tag of the line while leaving the data as is. Programming Note: If the prefetch is not followed by real writes, it is possible that prevailing data and data parity may indicate a parity error.

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.

31 … 26	25 … 21	20 … 16	15 … 0
SC 111000	base	rt	offset
6	5	5	16

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 vAddr_{1..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 || 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.

31 26	25 21 20 16 15 11	10 6	5 0
SPECIAL 000000	0 00 0000 0000 0000 0	stype	SYNC 001111
6	15	5	6

Format: `SYNC (stype = 0 implied)` **MIPS32**

Purpose: Synchronize Shared Memory

To order loads and stores.

Description:

Simple Description:

- SYNC affects only *uncached* and *cached coherent* loads and stores. The loads and stores that occur before the SYNC must be completed before the loads and stores after the SYNC are allowed to start.

- Loads are completed when the destination register is written. Stores are completed when the stored value is visible to every other processor in the system.

- SYNC is required, potentially in conjunction with SSNOP (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.

Restrictions:

The effect of SYNC on the global order of loads and stores for memory access types other than *uncached* and *cached coherent* is **UNPREDICTABLE**.

Operation:

```
    SyncOperation(stype)
```

Exceptions:

None

31 … 26	25	24 … 6	5 … 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBR 000001
6	1	19	6

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 ||
            $TLB[i]_{PFN1}$ ||
            $TLB[i]_{C1}$ || $TLB[i]_{D1}$ || $TLB[i]_{V1}$ || $TLB[i]_G$
EntryLo0 ← 0 ||
            $TLB[i]_{PFN0}$ ||
            $TLB[i]_{C0}$ || $TLB[i]_{D0}$ || $TLB[i]_{V0}$ || $TLB[i]_G$
```

Exceptions:

Coprocessor Unusable

Machine Check

31 26	25	24 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBWI 000010
6	1	19	6

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. If multiple TLB matches are detected on a TLBWI, a Machine Check exception is signaled. 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:

```
$i \gets Index$
$TLB[i]_{Mask} \gets PageMask_{Mask}$
$TLB[i]_{VPN2} \gets EntryHi_{VPN2}$
$TLB[i]_{ASID} \gets EntryHi_{ASID}$
$TLB[i]_{G} \gets EntryLo1_{G}$ and $EntryLo0_{G}$
$TLB[i]_{PFN1} \gets EntryLo1_{PFN}$
$TLB[i]_{C1} \gets EntryLo1_{C}$
$TLB[i]_{D1} \gets EntryLo1_{D}$
$TLB[i]_{V1} \gets EntryLo1_{V}$
$TLB[i]_{PFN0} \gets EntryLo0_{PFN}$
$TLB[i]_{C0} \gets EntryLo0_{C}$
$TLB[i]_{D0} \gets EntryLo0_{D}$
$TLB[i]_{V0} \gets EntryLo0_{V}$
```

Exceptions:

Coprocessor Unusable

Machine Check

31 26	25	24 6	5 0
COP0 010000	CO 1	Implementation-Dependent Code	WAIT 100000
6	1	19	6

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 when an interrupt is signalled, irrespective of whether the interrupt is enabled or not. (*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 interrupt, that interrupt is taken between the WAIT instruction and the following instruction (*EPC* for the interrupt points to 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

31 26	25	24 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBWR 000110
6	1	19	6

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. If multiple TLB matches are detected on a TLBWR, a Machine Check exception is signaled. 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
$TLB[i]_{Mask} \leftarrow PageMask_{Mask}$
$TLB[i]_{VPN2} \leftarrow EntryHi_{VPN2}$
$TLB[i]_{ASID} \leftarrow EntryHi_{ASID}$
$TLB[i]_{G} \leftarrow EntryLo1_{G}$ and $EntryLo0_{G}$
$TLB[i]_{PFN1} \leftarrow EntryLo1_{PFN}$
$TLB[i]_{C1} \leftarrow EntryLo1_{C}$
$TLB[i]_{D1} \leftarrow EntryLo1_{D}$
$TLB[i]_{V1} \leftarrow EntryLo1_{V}$
$TLB[i]_{PFN0} \leftarrow EntryLo0_{PFN}$
$TLB[i]_{C0} \leftarrow EntryLo0_{C}$
$TLB[i]_{D0} \leftarrow EntryLo0_{D}$
$TLB[i]_{V0} \leftarrow EntryLo0_{V}$
```

Exceptions:

Coprocessor Unusable

Machine Check

Chapter 14

MIPS16e™ Application-Specific Extension to the MIPS32® Instruction Set

This chapter describes the MIPS16e™ ASE as implemented in the 74K core. Refer to *Volume IV-a* of the *MIPS32 Architecture Reference Manual* for a general description of the MIPS16e ASE and detailed descriptions of the instructions.

This chapter covers the following topics:

- Section 14.1 “Instruction Bit Encoding”
- Section 14.2 “Instruction Listing”

14.1 Instruction Bit Encoding

Table 14.2 through Table 14.9 describe the encoding used for the MIPS16e ASE. Table 14.1 describes the meaning of the symbols used in the tables.

Table 14.1 Symbols Used in the Instruction Encoding Tables

*	Operation or field codes marked with this symbol are reserved for future use. Executing such an instruction causes a Reserved Instruction Exception.
δ	(Also *italic* field name.) Operation or field codes marked with this symbol denote 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 causes 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 14.2 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 14.3 MIPS16e JAL(X) Encoding of the x Field

x	bit 26	
	0	1
	JAL	JALX

Table 14.4 MIPS16e SHIFT Encoding of the f Field

f	bits 1..0			
	0	1	2	3
	00	01	10	11
	SLL	β	SRL	SRA

Table 14.5 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 14.6 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 14.7 MIPS16e RRR Encoding of the f Field

f	bits 1..0			
	0	1	2	3
	00	01	10	11
	β	ADDU	β	SUBU

Table 14.8 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 14.9 MIPS16e I8 Encoding of the s Field when funct=SVRS

s	bit 7	
	0	1
	RESTORE	SAVE

Table 14.10 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 14.11 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	*

14.2 Instruction Listing

The MIPS16e instructions are listed by instruction type in Table 14.12 through Table 14.19.

Table 14.12 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 14.13 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 14.14 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 14.15 MIPS16e Arithmetic Two or Three Operand Register Instructions

Mnemonic	Instruction	Extensible Instruction
ADDU	Add Unsigned	No
AND	AND	No
CMP	Compare	No
MOVE	Move	No
NEG	Negate	No

Table 14.15 MIPS16e Arithmetic Two or Three Operand Register Instructions

Mnemonic	Instruction	Extensible Instruction
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 14.16 MIPS16e Special Instructions

Mnemonic	Instruction	Extensible Instruction
BREAK	Breakpoint	No
SDBBP	Software Debug Breakpoint	No
EXTEND	Extend	No

Table 14.17 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 14.18 MIPS16e Jump and Branch Instructions

Mnemonic	Instruction	Extensible Instruction
B	Branch Unconditional	Yes
BEQZ	Branch on Equal to Zero	Yes
BNEZ	Branch on Not Equal to Zero	Yes

Table 14.18 MIPS16e Jump and Branch Instructions

Mnemonic	Instruction	Extensible Instruction
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 14.19 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

References

This appendix lists other documents available from MIPS Technologies, Inc. that are referenced elsewhere in this document. These documents may be included in the `$MIPS_PROJECT/doc` area of a typical 74K soft or hard core release, or in some cases may be available on the MIPS web site http://www.mips.com.

1. MIPS32® Architecture For Programmers, Volume I: Introduction to the MIPS32® Architecture, MIPS document: MD0082

2. MIPS32® Architecture For Programmers, Volume II: The MIPS32® Instruction Set, MIPS document: MD0082

3. MIPS32® Architecture For Programmers, Volume IV-e, MIPS document: MD00374

4. Programming the MIPS32® 74K™ Processor Core Family, MIPS document: MD00541

5. Programming the MIPS 74K™ Family Cores for DSP, MIPS document: MD00544.

6. MIPS® EJTAG Specification, MIPS document: MD00047

7. MIPS® PDtrace™ Interface and Trace Control Block Specification, MIPS document: MD00439

Appendix B

Revision History

MIPS documents include change bars (vertical bars in the page margin) that mark significant changes to the document since its last release. Change bars are removed for changes which are more than one revision old.

Please note that changes to figures are denoted by change bars in the figure's title only, not on the entire figure. Also, certain parts of this document may refer to Architecture specifications, and the change bars within these sections indicate alterations since the previous version of the relevant Architecture document.

Date	Revision	Description
January 31, 2007	01.00	Initial external release
May 25, 2007	01.01	Updates to Pipeline description, CP0 registers, FPU pipeline, CACHE and WAIT instructions
Dec 14, 2007	01.02	Updated to reflect • Reduction in effective pipeline length by 2 stages (removal of AP, EF and GR stages) • Instruction buffer size increase to 12 entries • Addition of Instruction cache Prefetching and associated controls in *Config7[PREF]* • Addition of *UserLocal* Coprocessor 0 register and associated controls in *Config3[ULRI], HWREnal* • Addition of *Config7[WII]* to enable unblocking of **`wait`** even if *Status[IE]* is cleared • Inclusion of *L23DataLo and L23DataHi* and renaming of *STagLo* to *L23TagLo* • Corrections to descriptions of *ITagLo* and *DTagLo* • Enhanced description of *PerfCtl* register • Removed *Config6* register • Revised description of *CacheErr* register
November 14, 2008	01.03	• Updated Fig.7.37 to mark bit[10] as unused • Add section on PDtrace, including new registers • Modify document to reflect support for Instruction Scratchpad RAM (ISPRAM) and optional I-Cache size of 0KB