MIPS32® proAptiv™ Multiprocessing System Software User’s Manual

Document Number: MD00878
Revision 01.22
May 14, 2013

MIPS and MIPS’ affiliates do not assume any liability arising out of the application or use of this information, or of any error or omission in such information. Any warranties, whether express, statutory, implied or otherwise, including but not limited to the implied warranties of merchantability or fitness for a particular purpose, are excluded. Except as expressly provided in any written license agreement from MIPS or an authorized third party, the furnishing of this document does not give recipient any license to any intellectual property rights, including any patent rights, that cover the information in this document.

The information contained in this document shall not be exported, reexported, transferred, or released, directly or indirectly, in violation of the law of any country or international law, regulation, treaty, Executive Order, statute, amendments or supplements thereto. Should a conflict arise regarding the export, reexport, transfer, or release of the information contained in this document, the laws of the United States of America shall be the governing law.

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

Table of Contents

List of Figures

List of Tables

Chapter 1

Architectural Overview

The proAptiv Multiprocessing System is a high performance multi-core microprocessor with best in class power efficiency for use in system-on-chip (SoC) applications. The proAptiv Multiprocessing System combines a deep pipeline with multi-issue out of order execution to deliver outstanding computational throughput. The proAptiv Multiprocessing System is fully configurable/synthesizable and can contain one to six MIPS32® proAptiv cores, system level coherence manager with L2 cache, optional coherent I/O port, and optional floating point unit.

The proAptiv Multiprocessing System is available in the following configurations. All of these configurations include a second generation Coherence Manager with integrated L2 cache (CM2).

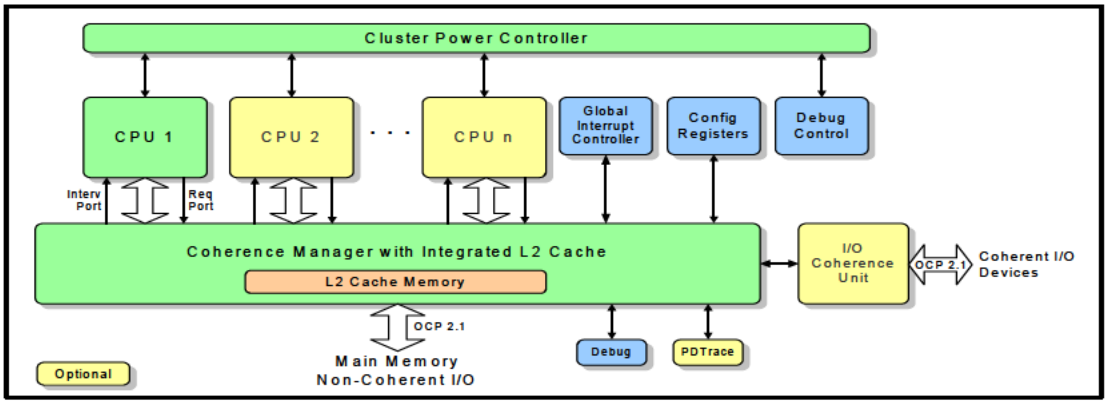
- Single core
- Dual core
- Three or more cores

The proAptiv Multiprocessing System contains the following logic blocks.

- proAptiv Cores (1, 2, 3, 4, or 6)
- Coherence Manager (2nd generation) with integrated L2 cache (CM2)
- Optional enhanced 2nd generation Floating Point Unit per core (FPU2)
- Cluster Power Controller (CPC)
- Global Interrupt Controller (GIC)
- I/O Coherence Unit (IOCU)
- Global Configuration Registers (GCR)
- Multiprocessing System Debug Unit
- Optional PDTrace in-system trace debugger

Figure 1.1 shows a block diagram of the proAptiv Multiprocessing System.

Figure 1.1 proAptiv Multiprocessing System Block Diagram



In the proAptiv Multiprocessing System, multi-CPU coherence is handled in hardware by the Coherence Manager. The optional I/O Coherence Unit (IOCU) supports hardware I/O coherence by bridging a non-coherent OCP I/O interconnect to the Coherence Manager (CM2) and handling ordering requirements. The Global Interrupt Controller (GIC) handles the distribution of interrupts between and among the CPUs. Under software controlled power management, the Cluster Power Controller (CPC) can gate off the clocks and/or voltage supply to idle cores.

1.1 Features

The following subsections describe the features of the proAptiv Multiprocessing System. The features are divided into system level and core level.

1.1.1 System Level Features

- 1 - 6 coherent MIPS32 proAptiv CPU cores
- Second generation system-wide Coherence Manager (CM2) providing L2 cache, I/O and interrupt coherence across all CPU cores
- Integrated 8-way set associative L2 cache controller supporting 256 KB to 8 MB cache sizes with variable wait state control for 1:1 clock and optimal SRAM speed
- L1 data cache supporting the MESI coherence states and cache-to-cache data transfers
- Cluster Power Controller (CPC) to shut down idle CPU cores
- Up to two hardware I/O coherence ports (IOCU) per system (optional)
- Speculative memory reads and out-of-order data return to reduce latency
- Separate clock ratios on memory and IOCU OCP ports
- Clock ratio of 1:1 between Core and L2 cache
- SOC system interface supports OCP version 2.1 protocol with 32-bit address and 64-bit or 256-bit data paths
- Power Control
  - Minimum frequency: 0 MHz
  - Software controlled power-down mode (triggered by WAIT instruction)
  - Software-controlled clock divider
  - Cluster-level dynamic clocking

  - Cluster Power Controller (CPC) controlling shut down of idle CPU cores

- EJTAG Debug 5.0 port supporting multi-CPU debug

- MIPS PDtrace debug version 6.16 (optional)
  - PC, data address and data value tracing w/ trace compression
  - Includes features for correlation with CM trace
  - Support for on-chip and off-chip trace memory
  - Support for system-level trace

- Full scan design achieves test coverage in excess of 99% with optional memory BIST for internal SRAM arrays

1.1.2 CPU Core Level Features

- Efficient pipeline with integer, floating point and optional CorExtend execution units shared amongst issue pipes.

- MIPS32 Release3 Instruction Set and Privileged Resource Architecture.

- Optional 2nd generation IEEE-754 compliant Floating Point Unit (FP2)

- Enhanced virtual addressing (EVA) mode allows for up to 3.5 GB of user or kernel virtual address space

- Instruction Fetch Unit (IFU) with 4 instructions fetched per cycle

- Quad integer Out-of-Order issue with dedicated integer completion buffers that hold execution results until instructions are graduated in program order

- Dual floating-point issue with dedicated completion buffers that hold execution results until instructions are graduated in program order

- Programmable Memory Management Unit with large first-level ITLB/DTLB backed by fast on-core second-level variable page size TLB (VTLB) and fixed page size TLB (FTLB):
  - 16-entry Instruction TLB (ITLB) with page sizes of 4 KB or 16 KB per entry
  - 32 dual-entry Data TLB (DTLB) with page sizes of 4 KB or 16 KB per entry
  - 64 dual-entry VTLB with page sizes up to 256 MB per entry
  - 512 dual-entry 4-way set associative FTLB with page sizes of 4 KB or 16 KB per entry (optional)
  - VTLB and FTLB can be accessed simultaneously on lookups

- L1 Instruction and Data Caches
  - Can be configured as 32 or 64 KB per cache
  - L1 MESI coherent cache states
  - 32-byte cache line size
  - Virtually indexed, physically tagged
  - Parity support on L1 data cache
  - Parity support on L1 instruction cache

- Data and Instruction Scratchpad RAM can be configured from 4 KB to 1 MB (optional).

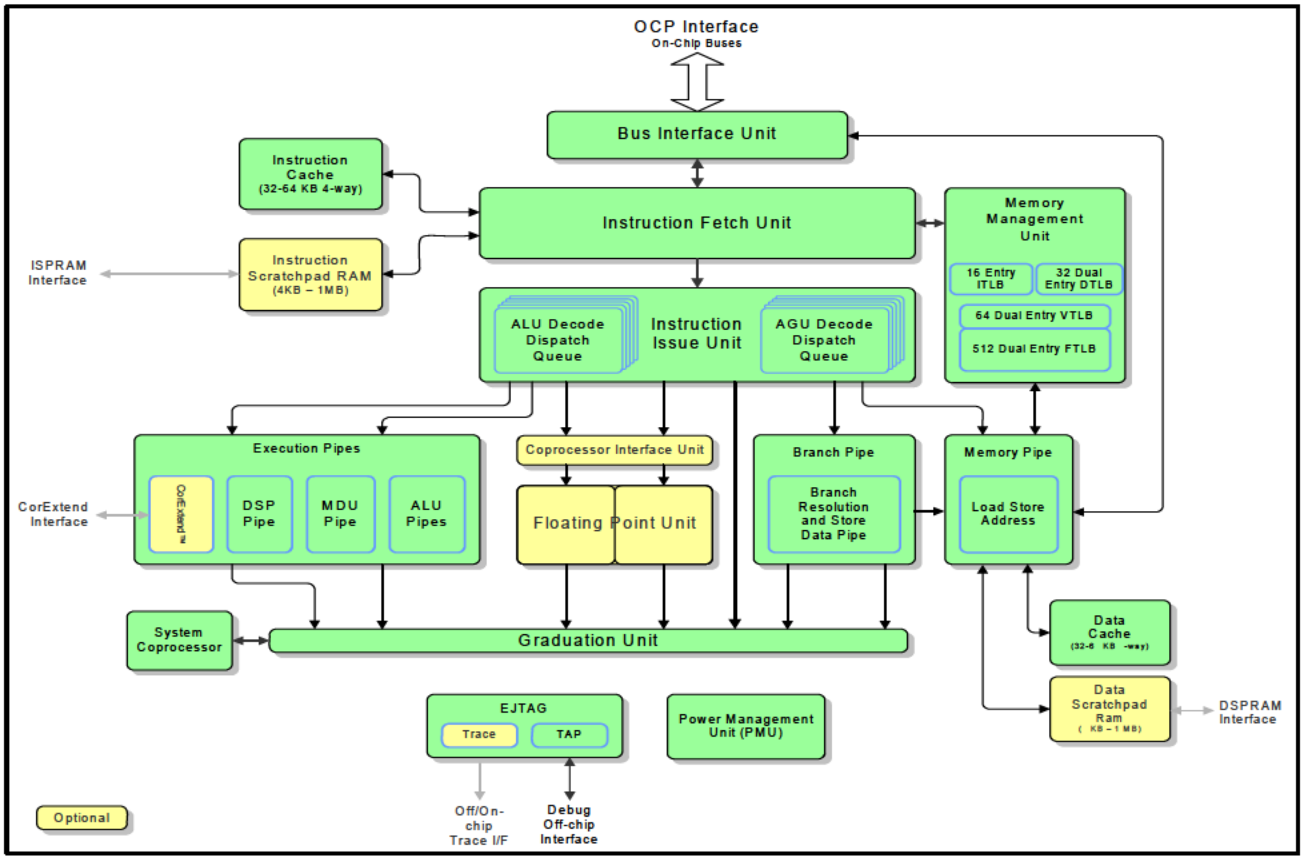
- Optional MIPS DSP ASE:
  - 3 additional pairs of accumulator registers
  - Fractional data types, Saturating arithmetic
  - SIMD instructions operate on 2x16b or 4x8b simultaneously

- Write merging for uncached accelerated (UCA) operations
- Integrated integer Multiply/Divide Unit (MDU)
- CorExtend® MIPS32® compatible User Defined Instruction Set Extension allows user to define and add instructions to the core at build time
- Core Power Reduction Features
  - Power reduction by turning off core clock during outstanding bus requests
  - Power reduction by implementing intelligent way selection in the L1 instruction cache
  - Power Reduction by enabling 32-bit accesses of the L1 data cache RAMs

1.2 proAptiv CPU Core

Figure 1.2 shows a block diagram of a single proAptiv core. The following subsections describe the logic blocks in this diagram.

Figure 1.2 proAptiv Core Block Diagram



For more information on the proAptiv core in a multiprocessing environment, refer to Section 1.3 "Multiprocessing System"

1.2.1 MIPS Release 3 Architecture

The proAptiv core implements the MIPS32™ Release 3 Architecture in a superscalar, out-of-order execution pipeline. In addition, the proAptiv core also supports the MIPS16e™ ASE for code compression, and the DSP ASE Revision 2 for accelerating integer SIMD codes.

1.2.2 Instruction Fetch Unit

The Instruction Fetch Unit (IFU) is responsible for fetching instructions from the instruction cache, instruction scratchpad or memory, supplying them to the Instruction Issue Unit (IIU). The IFU can fetch up to four MIPS32 instructions at a time from the 4-way associative instruction cache. Instructions can also be fetched immediately from refill buffers in the event of an instruction cache miss.

The IFU employs sophisticated branch prediction and instruction supply strategies. The main predictor consists of large RAM-based global branch history tables (BHT) that are indexed by different combinations of instruction PC and global history. A proprietary scheme is used to combine information from the three arrays to make a branch direction prediction.

Branch target prediction is provided by a hierarchy of multiple arrays. The fully-associative Level 0 and Level 1 Branch Target Buffer (BTB) are used tor fast target re-steers on predicted taken branches, including returns. A large 4-way associative Level 2 BTB backs up the Level 0 and Level 1 BTB's and also predicts indirect branches, even those with multiple target addresses.

The IFU also has a hardware-based return prediction stack to predict subroutine return addresses.The main predictor contains a BTAC (Branch Target Address Calculator) that can correct target mispredicts from lower-level predictors without paying a full branch resolution penalty. The IFU supports fully out-of-order branch resolution.

The IFU has a 16-entry micro-Instruction TLB (ITLB) used to translate the virtual address into a physical address. This translated physical address is used to compare against tags in the instruction cache to determine a hit. Refer to Section 1.2.5 "Memory Management Unit (MMU)" for more information.

A 24-entry instruction buffer decouples the instruction fetch from the execution. Up to 4 instructions can be written into this buffer, and a maximum of 2 instructions can be read from this buffer. To maximize performance, some 'bonding' (or concatenation) of instructions is done at this stage while other types of instruction 'bonding' are performed downstream.

The IFU can also be configured to allow for hardware prefetching of cache lines on a miss. When an instruction cache miss is detected, the IFU can prefetch the next 0, 1, or 2 lines (in addition to the missed line) to reduce average miss latency. This mechanism provides excellent performance without incurring the area, power and latency costs of overly complicated branch or instruction prefetch strategies.

The Global History register is internal to the IFU block and supports a novel history computation scheme that factors different information into the history for different kinds of control transfer instructions. These novel hashing schemes enable significantly lower mispredict rates than other competing processors, directly translating to real world performance in many different applications.

The proAptiv level 1 (L1) instruction cache incorporates 'next fetch way' hit prediction logic. This allows the IFU to power on only those cache tag and data arrays that will provide the final instruction bytes and contributes to low power consumption.

1.2.3 Instruction Issue Unit (IIU)

The Instruction Issue Unit (IIU) unit is responsible for receiving instructions from the IFU and dispatching them to the out-of-order instruction scheduling windows and global instruction tracking window at a rate of 4 instructions per cycle.

The IIU tracks dynamic data flow dependencies between operations and issues them to the various pipes as efficiently as possible. Two schedulers, called the ALU DDQ and the AGU DDQ, service the various integer pipes.

The schedulers employ multiple dependency wake-up and pick schemes to enable age-based scheduling at high frequency. Having only two schedulers, rather than a low-frequency centralized scheduler or a large number of distributed reservation stations, is key to providing superior performance and power characteristics.

The IIU helps to ‘bond’ load and store operations whereby two 32-bit loads or stores to adjacent locations are ‘bonded’ or concatenated into one 64-bit memory access. This allows a factor of two improvement in certain memory intensive codes. The IIU also enables instruction ‘cracking’, whereby certain operations, like cacheable stores, are split into multiple micro-ops such as store-address and store-data operations.

Instructions are first renamed using a rename map, replacing the architectural register names with microarchitectural names from a global rename pool. The IIU also keeps track of the progress of each instruction through the pipeline, updating the availability of operands in the ‘rename map’ and in all dependent instructions. Renamed instructions are steered to the most appropriate schedulers, taking opcode and other information into account.

The IIU also keeps track of global pipeline flushes, adjusting the rename map and other control structures to deal with interrupts, exceptions and other unexpected changes of control.

1.2.4 proAptiv L1 Caches

The proAptiv core contains L1 instruction and data caches as described in the following subsections.

1.2.4.1 Level 1 Instruction Cache

The Level-1 (L1) instruction cache is configurable at 32 or 64 KB in size and is organized as 4-way set associative. Up to four instruction cache misses can be outstanding. The instruction cache is virtually indexed and physically tagged to make the data access independent of virtual to physical address translation. Instruction cache tag and data access are staggered across 2 cycles, with up to 4 instructions fetched per cycle.

An instruction 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.

The proAptiv core supports instruction-cache locking. Cache locking allows critical code segments to be locked into the cache on a “per-line” basis, enabling the system programmer to maximize the performance of the system cache.

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

1.2.4.2 Level 1 Data Cache

The Level 1 (L1) data cache is configurable at 32 or 64 in size. It is also organized as 4-way set associative. Data cache misses are non-blocking and up to nine misses may be outstanding. The data cache is virtually indexed and physically tagged to make the data access independent of virtual-to-physical address translation. To achieve the highest possible frequencies using commercially available SRAM generators, cache access and hit determination is spread across three pipeline stages, dedicating an entire cycle for the SRAM access.

A data cache tag entry holds 21 bits of physical address, a valid bit, a lock bit, and an optional parity bit. The data entry holds 64 bits of data per way, with optional parity per byte. There are 4 data entries for each tag entry. The tag and data entries exist for each way of the cache.

The proAptiv core supports a data-cache locking mechanism identical to that used in the instruction cache. Critical data segments are locked into the cache on a “per-line” basis. The locked contents can be updated on a store hit, but are not selected for replacement on a cache miss. Locked lines do not participate in the coherence scheme so processes which lock lines into a particular cache should be locked to that processor and prevented from migrating.

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

1.2.4.3 Level 1 Cache Memory Configuration

As described above, the proAptiv CPU incorporates on-chip L1 instruction and data caches that are typically implemented from readily available single-port synchronous SRAMs and accessed in two cycles: one cycle for the actual SRAM read and another cycle for the tag comparison, hit determination, and way selection. The instruction and data caches each have their own 64-bit data paths and can be accessed simultaneously. Table 1.1 lists the proAptiv CPU instruction and data cache attributes.

Table 1.1 proAptiv™ CPU L1 Instruction and Data Cache Attributes

Parameter	Instruction	Data
Size[1]	32 or 64 KB	32 or 64 KB
Organization	4-way set associative	4-way set associative
Line Size	32 Bytes	32 Bytes
Read Unit	64 bits	64 bits
Write Policies	N/A	coherent and non-coherent write-back with write allocate
Miss restart after transfer of	miss word	miss word
Cache Locking	per line	per line
Error Detection Mechanism	Parity	Parity

[1] For Linux based applications, MIPS recommends a cache size of 64 KB, with a minimum size of 32 KB.

1.2.5 Memory Management Unit (MMU)

The proAptiv core contains a Memory Management Unit (MMU) that is primarily responsible for converting virtual addresses to physical addresses and providing attribute information for different segments of memory. The proAptiv MMU contains the following Translation Lookaside Buffer (TLB) types:

- 16-entry Instruction TLB (ITLB) with 4 KB or 16 KB per entry
- 32 dual-entry Data TLB (DTLB) with up to 4 KB or 16 KB per entry
- 64 dual-entry Variable Page Size Translation Lookaside Buffer (VTLB) with up to 256 MB per entry
- 512 dual-entry 4-way set associative Fixed Page Size Translation Lookaside Buffer (FTLB) with up to 16 KB per entry

1.2.5.1 Instruction TLB (ITLB)

The ITLB is a 16-entry high speed TLB dedicated to performing translations for the instruction stream. The ITLB maps only 4 KB or 16 KB pages. Larger pages are split into smaller pages of one of these two sizes and installed in the ITLB.

The ITLB is managed by hardware and is transparent to software. The larger VTLB and FTLB structures are used as a backup structure for the ITLB. If a fetch address cannot be translated by the ITLB, the VTLB/FTLB attempts to translate it in the following clock cycle or when available. If successful, the translation information is copied into the ITLB for future use.

1.2.5.2 Data TLB (DTLB)

The DTLB is a 32 dual-entry high speed TLB dedicated to performing translations for the data stream. The DTLB maps only 4 KB or 16 KB pages. Larger pages are split into one of these configured sizes and installed in the DTLB.

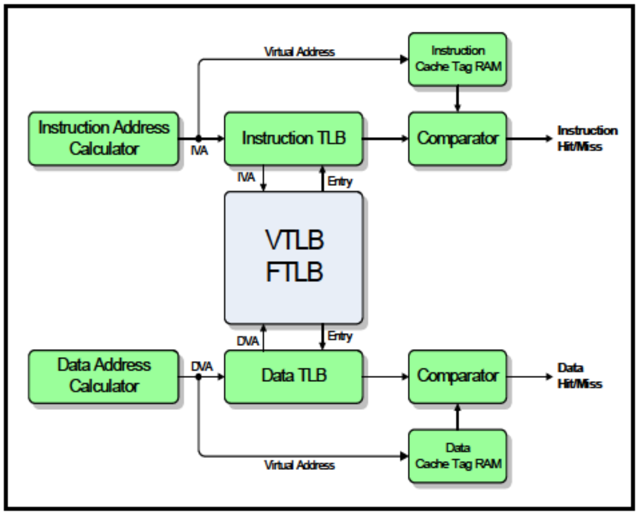
The DTLB is managed by hardware and is transparent to software. The larger VTLB and FTLB structures are used as a backup structure for the DTLB. If a fetch address cannot be translated by the DTLB, the VTLB/FTLB attempts to translate it in the following clock cycle or when available. If successful, the translation information is copied into the DTLB for future use.

1.2.5.3 Variable Page Size TLB (VTLB)

The VTLB is a fully associative variable translation lookaside buffer with 64 dual entries that can map variable size pages from 4KB to 256MB. When an instruction address is calculated, the virtual address is first compared to the contents of the ITLB and DTLB. If the address is not found in either the ITLB or DTLB, the VTLB/FTLB is accessed. If the entry is found in the VTLB, that entry is then written into the ITLB or DTLB. If the address is not found in the VTLB, a software TLB exception is taken. For data accesses, the virtual address is looked up in the VTLB only, and a miss causes a TLB exception.

Figure 1.3 shows how the ITLB, DTLB, and VTLB/FTLB are implemented in the proAptiv core.

Figure 1.3 Address Translation



1.2.5.4 Fixed Page Size TLB (FTLB)

The FTLB is 512 dual entries organized as 128 sets and 4-ways. Each set of each way contains dual data RAM entries and one tag RAM entry. If the tag RAM contents match the requested address, either the low or high RAM location of the dual data RAM is accessed depending on the state of the most-significant-bit (MSB) of the offset portion of the virtual address (VPN2). Each RAM location can only map a fixed page size, which is configurable to 4KB or 16KB. The FTLB resides at the top of the VTLB range as shown in Figure 1.4.

Figure 1.4 proAptiv VTLB and FTLB

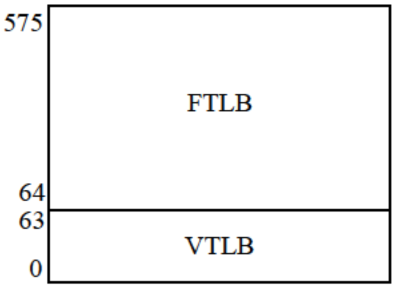
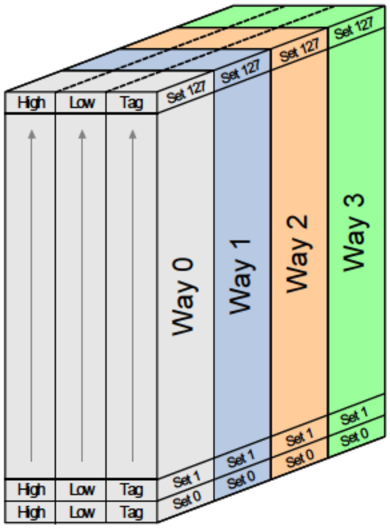


Figure 1.5 shows a block diagram of the 512-entry FTLB.

Figure 1.5 FTLB Organization



1.2.5.5 Enhanced Virtual Address

The proAptiv core contains a programmable memory segmentation scheme called Enhanced Virtual Address (EVA), which allows for more efficient use of 32-bit address space. Traditional MIPS virtual memory support divides up the virtual address space into fixed size segments, each with fixed attributes and access privileges. Such a scheme limits the amount of physical memory available to 0.5GB, the size of kernel segment 0 (*kseg0)*.

In EVA, the size of virtual address space segments can be programmed, as can their attributes and privilege access. With this ability to overlap access modes, *kseg0* can now be extended up to 3.0 GB, leaving at least one 1.0 GB segment for mapped kernel accesses[1]. This extended *kseg0* is called *xkseg0*. This space overlaps with *useg*, because segments in *xkseg0* are programmed to support mapped user accesses and unmapped kernel accesses. Consequently, user space is equal to the size of *xkseg0*, which can also be up to 3.5GB.

This concept is shown in Figure 1.6.

[1] If necessary, *xkseg0* can be extended to 3.5GB, allowing 0.5GB for Kernel mapped virtual address space (now *kseg2*).

Figure 1.6 Example Enhanced Virtual Address (EVA) Memory Map

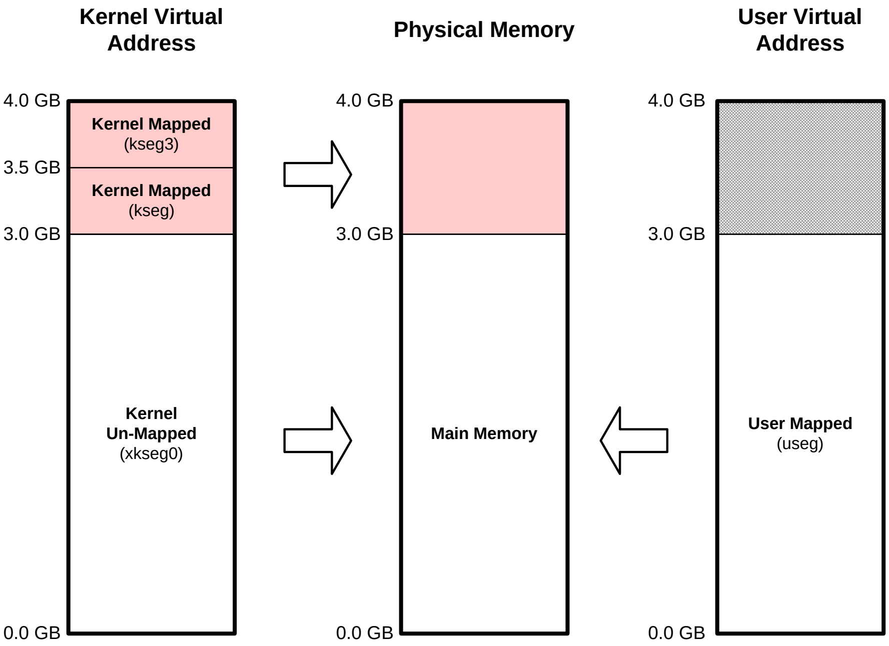


Figure 1.6 shows an example of how the traditional MIPS kernel virtual address space can be remapped using programmable memory segmentation to facilitate an extended virtual address space. As a result of defining the larger kernel segment as xkseg0, the kernel has unmapped access to the lower 3GB of the virtual address space. This allows for a total of 3GB of DRAM to be supported in the system.

To allow for efficient kernel access to user space, new load and store instructions have been defined which allow kernel mapped access to *useg*. For more information, refer to the MMU chapter.

Note that the attributes of xkseg0 are the same as the previous kseg0 space in that it is a kernel unmapped, uncached region.

1.2.6 Execution Pipelines

The proAptiv core contains the following execution pipelines;

- Arithmetic Logic Pipeline
- DSP Pipeline
- Multiply-Divide Pipeline
- Memory Pipeline
- Branch Pipeline
- Two FPU Pipelines (optional)

Each of these execution units is described in the following subsections. Instruction intended for arithmetic logic pipeline are driven by the out-of-order ALU Decode and Dispatch queue inside the Instruction Issue Unit (IIU) as shown in Figure 1.2. The other four pipelines are driven by the out-of-order Address Generation unit (AGU) Decode and Dispatch queue also located in the IIU.

1.2.6.1 Arithmetic Logic Pipeline

The arithmetic unit pipeline consists of one execution unit, called the ALU (Arithmetic Logic Unit), which performs integer instructions such as adds, shifts and bitwise logical operations with a single cycle latency.

If the IIU decodes a single cycle instruction, it is usually sent to the ALU dispatch queue that feeds the arithmetic unit pipeline. This pipeline also contributes to performing ‘bonded’ loads. Refer to Section 1.2.3 “Instruction Issue Unit (IIU)” for a definition of instruction ‘bonding’.

1.2.6.2 Digital Signal Processing Pipeline

The DSP pipeline executes a subset of the DSP instructions, including shifts. It can also execute certain arithmetic operations, large shifts and special operations such as counting leading zeroes or ones. Most operations in this unit execute with a two cycle latency.

1.2.6.3 Multiply/Divide Pipeline

The multiply/divide pipeline executes integer multiplies, integer divides, integer multiply-accumulates and some DSP instructions. The multiply/divide pipeline incorporates a new very high-speed integer divider.

The MDU consists of a 32 × 32 multiplier, result/accumulation registers (HI and LO), a divide state machine, and all necessary multiplexers and control logic.

The MDU supports execution of one multiply or multiply-accumulate operation every clock cycle whereas divides can be executed as fast as one every six cycles.

1.2.6.4 Memory Pipeline

The memory pipeline primarily contains the LSU (Load Store Unit), which is responsible for interfacing with the AGU dispatch queue and processing load/store instructions to read/write data from data caches and downstream memory.

This unit is capable of handling loads and stores issued out-of-order. The LSU also supports the ‘cracking’ of store instructions by allowing a store’s data and address to reach it in any order. This ability to receive loads and stores in almost any order enables very high performance, compared to competing out-of-order machines that do not allow such concurrency. Such instruction-level parallelism allows maximum utilization of the memory pipe resources with minimal area and power.

The LSU can execute loads and stores at twice the rate of regular operations by concatenating data from two 32-bit memory locations to form a single 64-bit entity. This ‘bonding’ of instructions allows the LSU to provide almost all the benefits of dual memory access pipes without incurring the area and power costs of multiple tag, data and TLB structures.

The Memory Pipe receives instructions from the Instruction Issue Unit (IIU) and interfaces to the L1 data cache and data scratchpad RAM (DSPRAM). Loads are non-blocking in the proAptiv core. Loads that miss in the data cache are allowed to proceed with their destination register marked unavailable. Consumers of this destination register are held back and replayed as needed once the cache miss has been serviced by the downstream memory subsystem, which includes the high performance L2 cache.

Graduated load misses and store hits and misses are sent in order to the Load/Store Graduation Buffer (LSGB). The LSGB has corresponding data and address buffers to hold all relevant attributes.

An 8-entry Fill Store Buffer (FSB) tracks outstanding fill or copy-back requests. It fills the data cache at the rate of 128-bits per cycle when an incoming line is completely received. Each FSB entry can hold an entire cache line. The Load Data Queue (LDQ) keeps track of outstanding load misses and forwards the critical data to the main pipe as soon as it becomes available.

Hardware anti-aliasing allows using the core with operating systems that do not support software page coloring. The fully-associative DTLB operates a clock earlier in the LSU pipeline, making use of fast add-and-compare logic to enable virtual address to physical address translations that do not require the area and power expense of virtual tagging. All of this is done completely transparent to software.

1.2.6.5 Branch Pipeline

The Branch pipeline performs the following functions:

- Executes Branch and Jump instructions
- Performs Branch resolution
- Performs Jump resolution
- Sends the redirect to the Instruction Fetch Unit (IFU)
- Performs a write-back to the Link registers

1.2.6.6 Floating Point Pipelines

The optional Floating Point Unit (FPU) contains two pipelines; one for arithmetic operations and one for data transfer operations. The arithmetic pipeline executes operations such as multiply, divide, and square root.

The data transfer pipeline executes floating point loads, stores, move operations, and register-to-register transfers between the FPU and the integer unit.

For more information, refer to Section 1.2.13 “Floating Point Unit (FP2)”.

1.2.6.7 Graduation Unit (GRU)

The Graduation Unit (GRU) is responsible for committing execution results and releasing 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. Asynchronous interrupts are funneled into the GRU, which prioritizes those events with existing conditions and takes the appropriate interrupt.

The GRU reads the next set of completed instructions from the global instruction window every cycle and then reads the corresponding completion buffers and associated information. After processing the exception conditions, the GRU performs the following functions:

- Destination register(s) are updated and the completion buffers are released.
- Graduation information is sent to the IIU so it can update the rename maps to reflect the state of execution results (i.e., GPRs, Accumulators, etc.).
- Resolved branch information is sent to the IFU so that branch history tables can be updated and if needed, a pipeline redirect can be initiated. If sequential control flow is aborted for any reason, the GRU signals all core units to flush and recover microarchitectural state. After recovery is complete, it allows the IIU to resume dispatching instructions.

1.2.7 Instruction and Data Scratch Pad RAM

The proAptiv core allows blocks of scratchpad RAM to be attached to the load/store and/or instruction units. These allow low-latency access to a fixed block of memory. The size of both the instruction scratch pad RAM (ISPRAM) and data scratch pad RAM (DSPRAM) can be configured from a range of 4 KB to 1 MB. These RAM's are used for the temporary storage of information and can be modified by the user at any time.

1.2.8 Bus Interface (BIU)

The Bus Interface Unit (BIU) controls the programmable 64-bit or 256-bit interface to the CM2. The interface implements the Open Core Protocol (OCP). This implementation features 128-bit read and write data buses to efficiently transfer data to and from the L1 caches.

1.2.8.1 Write Buffer

The BIU contains a merging write buffer. The purpose of this buffer is to store and combine write transactions before issuing them to the external interface. The write buffer is organized as eight, 32-byte buffers. Each buffer can contain data from a single 32-byte aligned block of memory.

When using the write-through cache policy or performing uncached accelerated writes, the write buffer significantly reduces the number of write transactions on the external interface and reduces the amount of stalling in the core caused by the issuance of multiple writes in a short period of time.

The write buffer also holds eviction data for write-back lines. The load-store unit extracts dirty data from the cache and sends it to the BIU. In the BIU, the dirty data is gathered in the write buffer and sent out as a bursted write.

For uncached accelerated writes, the write buffer can gather multiple writes together and then perform a bursted write in order to increase the efficiency of the bus. Uncached accelerated gathering is supported for any size less than a doubleword.

Gathering of uncached accelerated stores can start on any arbitrary address and can be combined in any order within a cache line. Uncached accelerated stores that do not meet the conditions required to start gathering are treated like regular uncached stores.

1.2.8.2 SimpleBE Mode

To aid in attaching the proAptiv core to structures that cannot easily handle arbitrary byte-enable patterns, there is a mode that generates only "simple" byte enables. In this mode, only byte enables representing naturally aligned byte, halfword, word, and doubleword transactions will be generated.

In SimpleBE mode, the *SI_SimpleBE* input pin only controls the byte enables generated by the proAptiv core(s). It has no effect on byte enables produced by the IOCU. To achieve the effect of setting *SI_SimpleBE* to 'one' in systems with an IOCU, the I/O sub-system must only issue requests to the IOCU with naturally aligned byte enables.

When the *SI_SimpleBE* input signal to the proAptiv core is asserted, hardware sets bit 21 of the *Config* register (*Config.SB*) to indicate the device is in simple byte enable mode.

1.2.9 System Control Coprocessor (CP0)

In the MIPS architecture, CP0 is responsible for the virtual-to-physical address translation and cache protocols, the exception control system, the processor's diagnostic capability, the operating modes (kernel, user, supervisor, and debug), and whether interrupts are enabled or disabled. Configuration information, such as cache size and associativity, and the presence of features like MIPS16e or a floating point unit, are also available by accessing the CP0 registers.

CP0 also contains the logic used for identifying and managing exceptions. Exceptions can be caused by a variety of sources, including boundary cases in data, external events, or program errors.

For more information, refer to the CP0 chapter.

1.2.10 Interrupt Handling

The proAptiv core supports six hardware interrupts, two software interrupts, a timer interrupt, and a performance counter interrupt. These interrupts can be used in any of three interrupt modes, as defined by Release 3 of the MIPS32 Architecture:

- Interrupt compatibility mode, which acts identically to that in an implementation of Release 1 of the Architecture.
- Vectored Interrupt (VI) mode, which adds the ability to prioritize and vector interrupts to a handler dedicated to that interrupt. The presence of this mode is denoted by the *VInt* bit in the *Config3* register. This mode is architecturally optional. As it is always present on the proAptiv core, the *VInt* bit will always read 1.
- External Interrupt Controller (EIC) mode, which provides support for an external interrupt controller that handles prioritization and vectoring of interrupts. This mode is optional in the Release 3 architecture. The presence of this mode is denoted by the *VEIC* bit in the *Config3* register.

1.2.11 Modes of Operation

The proAptiv core supports four modes of operation:

- User mode, most often used for application programs.
- Supervisor mode provides an intermediate privilege level with access to the *ksseg* (kernel supervisor segment) address space.
- Kernel mode, typically used for handling exceptions and operating system kernel functions, including CP0 management and I/O device accesses.
- Debug mode is used during system bring-up and software development. Refer to Section 1.2.15 “EJTAG Debug Support” for more information on debug mode.

1.2.12 Coprocessor Interface Unit (CIU)

The CIU provides an interface between the main integer core and the Floating Point Unit (FPU2). The CIU contains a number of queues used to pass data to and from the coprocessors.

Coprocessor1 Load/Store instructions are forwarded to the FPU2. Even though some Coprocessor instructions do not go through the main integer pipeline, they are assigned an instruction identifier. This identifier is tracked in the Graduation Unit to generate a synchronization signal that is used to indicate to the CP1 coprocessor that the instruction has been cleared of all speculation and exception conditions in the integer pipe. Only coprocessor instructions that have reached such a state are allowed to commit results in the Coprocessor.

Coprocessor-based conditional branches are handled in the graduation unit, with condition-code information passed through the CIU.

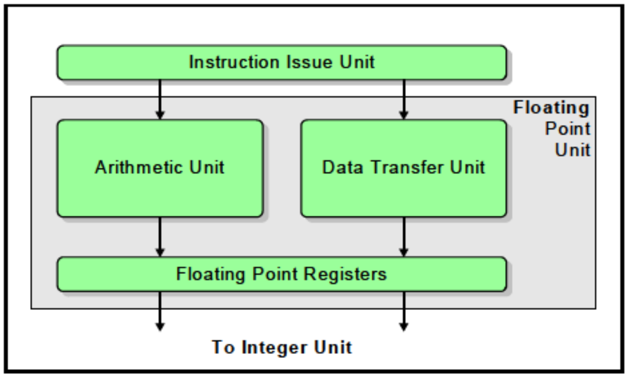
1.2.13 Floating Point Unit (FP2)

The proAptiv core features an optional IEEE 754 compliant 2nd generation Floating Point Unit (FPU2)[2]. The FP2 contains thirty-two, 64-bit floating point registers used for floating point operations. The FP2 is fully synthesizable and operates at the same clock speed as the CPU.

The FP2 supports both single- and double-precision instructions. It connects to the main processor through the CP1 coprocessor interface. The FPU can read up to 2 instructions from this queue and issue into its execution pipes.

Figure 1.7 shows a simplified block diagram of the FP2 floating point unit.

Figure 1.7 Floating Point Unit Block Diagram



1.2.13.1 FPU Performance

The performance of the FPU is optimized for double-precision formats. Most instructions have a one cycle throughput. The FPU implements the MIPS64 multiply-add (MADD) and multiply-sub (MSUB) instructions with intermediate rounding after the multiply function. The result is guaranteed to be the same as executing a MUL and an ADD instruction separately, but the instruction latency, instruction fetch, dispatch bandwidth, and the total number of register accesses required are greatly improved.

IEEE denormalized input operands and results are supported by hardware for many instructions. IEEE denormalized output results are not supported by hardware in general, but a fast flush-to-zero mode is provided to optimize performance. The fast flush-to-zero mode is enabled through the *FCSR* register, and use of this non-standard mode is recommended for best performance when denormalized results are generated. This situation occurs most often in GPU driver code or multimedia CODECS handling real-time data streams.

The FPU has two separate pipelines for floating point instruction execution—one for load/store instructions and another for all other compute instructions. These pipelines operate in parallel with the integer core pipeline and do not stall when the integer pipeline stalls. This allows long-running FPU operations, such as divide or square root, to be partially masked by system stalls and/or other integer unit instructions.

Arithmetic instructions are always dispatched and graduated in order, but loads and stores can complete out-of-order. The integer core performs the data access for load/store operations and transfer data to and from the FPU using the CIU. Load data may arrive in the FPU out-of-order relative to program order. The exception model is ‘precise’ at all times.

[2] Requires separate MIPS license.

The FPU implements a bypass mechanism that allows the result of an operation to be forwarded directly to the instruction that needs it without having to write the result to the FPU register and then read it back.

For more information, refer to the FPU chapter.

1.2.14 proAptiv Core Power Management

The proAptiv core offers several power management features, supporting low-power design, such as active power management and power-down modes of operation. The proAptiv core is a static design that supports slowing or halting the clocks to reduce system power consumption during idle periods.

For more information, refer to the Power Management chapter.

1.2.15 EJTAG Debug Support

The proAptiv core includes an Enhanced JTAG (EJTAG) block for use in software debugging of application and kernel code. For this purpose, in addition to standard user/supervisor/kernel modes of operation, the proAptiv core provides a Debug mode.

Debug mode is entered when a debug exception occurs (resulting from a hardware breakpoint, single-step exception, etc.) and continues until a debug exception return (DERET) instruction is executed. During this time, the processor executes the debug exception handler routine.

The EJTAG interface operates through the Test Access Port (TAP), a serial communication port used for transferring test data in and out of the proAptiv core. In addition to the standard JTAG instructions, special instructions defined in the EJTAG specification define which registers are selected and how they are used.

There are several types of simple hardware breakpoints defined in the EJTAG specification. These breakpoints stop the normal operation of the CPU and force the system into debug mode.

During synthesis, the proAptiv core can be configured to support the following breakpoint options:

- Zero instruction, zero data breakpoints
- Two instruction, one data breakpoints
- Four instruction, two data breakpoints

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

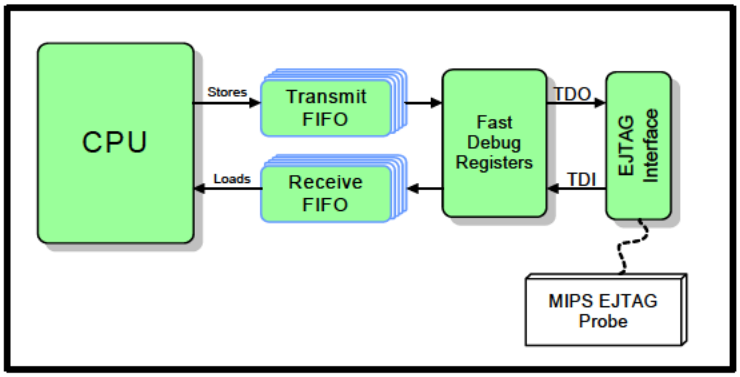
Data breakpoints occur on load and/or store transactions. Breakpoints are set on virtual address and address space identifier (ASID) values, similar to the Instruction breakpoint. Data breakpoints can also be set based on the value of the load/store operation. Finally, masks can be applied to the virtual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

In debug mode, EJTAG can request that a ‘soft’ reset be masked. This request is signalled via the *EJ_SRstE* pin. When this pin is deasserted, the system can choose to block some sources of soft reset. Hard resets, such as power-on reset or a reset switch, should not be blocked by this signal. This reset pin has no effect inside the core.

1.2.16 Fast Debug Channel

The proAptiv core includes the EJTAG Fast Debug Channel (FDC) as a mechanism for efficient bi-directional data transfer between the CPU and the debug probe. Data is transferred serially via the TAP interface. A pair of memory-mapped FIFOs buffer the data, isolating software running on the CPU from the actual data transfer. Software can configure the FDC block to generate an interrupt based on the FIFO occupancy or can poll the status.

Figure 1.8 Fast Debug Channel



1.2.17 PDTrace

The proAptiv core includes optional PDTrace support for real-time tracing of instruction addresses, data addresses, data values, performance counters, and processor pipeline inefficiencies. The trace information is collected in an on-chip or off-chip memory, for post-capture processing by trace regeneration software. Software-only control of trace is possible in addition to probe-based control.

An optional on-chip trace memory may be configured in size from 256B to 8 MB; it is accessed either through load instructions or the existing EJTAG TAP interface, which requires no additional chip pins. Off-chip trace memory is accessed through a special trace probe and can be configured to use 4, 8, 16, or 64 data pins plus a clock.

1.2.18 MIPS16e™ Application Specific Extension

The proAptiv core includes support for the MIPS16e ASE. This ASE improves code density through the use of 16-bit encodings of many MIPS32 instructions plus some MIPS16e-specific instructions. PC relative loads allow quick access to constants. Save/Restore macro instructions provide for single instruction stack frame setup/teardown for efficient subroutine entry/exit.

1.2.19 CorExtend® Unit

The CorExtend unit is a custom block that allows the user to connect to the proAptiv core pipeline with access to all programmer-visible general purpose registers and accumulator state.

MIPS provides a template to define the operand format and latency for the new instruction(s) to be added. Each instruction may select up to 2 source GPRs and/or 1 Accumulator from a set of 32 GPRs and 4 accumulators. The instruction may have a destination of either a GPR, an accumulator, or a private state.

1.3 Multiprocessing System

The proAptiv Multiprocessing System consists of the logic modules shown in Figure 1.1. Each of these blocks is described throughout this section.

1.3.1 Cluster Power Controller (CPC)

Individual CPUs within the cluster can have their clock and/or power gated off when they are not in use. This gating is managed by the Cluster Power Controller (CPC). The CPC handles the power shutdown and ramp-up of all CPUs in the cluster. Any proAptiv CPU that supports power-gating features is managed by the CPC.

The CPC also organizes power-cycling of the CM2 dependent on the individual core status and shutdown policy. Reset and root-level clock gating of individual CPUs are considered part of this sequencing.

1.3.1.1 Cluster Power Controller Reset Control

The reset input of the system resets the Cluster Power Controller (CPC). Reset sideband signals are required to qualify a reset as system cold, or warm start. Register setting determine the course of action:

- Remain in powered-down
- Go into clock-off mode
- Power-up and start execution

This prevents random power up of power domains before the CPC is properly initialized. In case of a system cold start, after reset is released, the CPC powers up the proAptiv CPUs as directed in the CPC cold start configuration. If at least one CPU has been chosen to be powered up on system cold start, the CM2 is also powered up.

When supply rail conditions of power gated coress have reached a nominal level, the CPC will enable clocks and schedule reset sequences for those coress and the coherence manager.

At a warm start reset, the CPC brings all power domains into their cold start configuration. However, to ensure power integrity for all domains, the CPC ensures that domain isolation is raised before power is gated off. Domains that were previously powered and are configured to power up at cold start remain powered and go through a reset sequence.

Within a warm start reset, sideband signals are also used to qualify if coherence manager status registers and GIC watch dog timers are to be reset or remain unchanged. The CPC, after power up of any CPU, provides a test logic reset sequence per domain to initialize TAP and PDTrace logic.

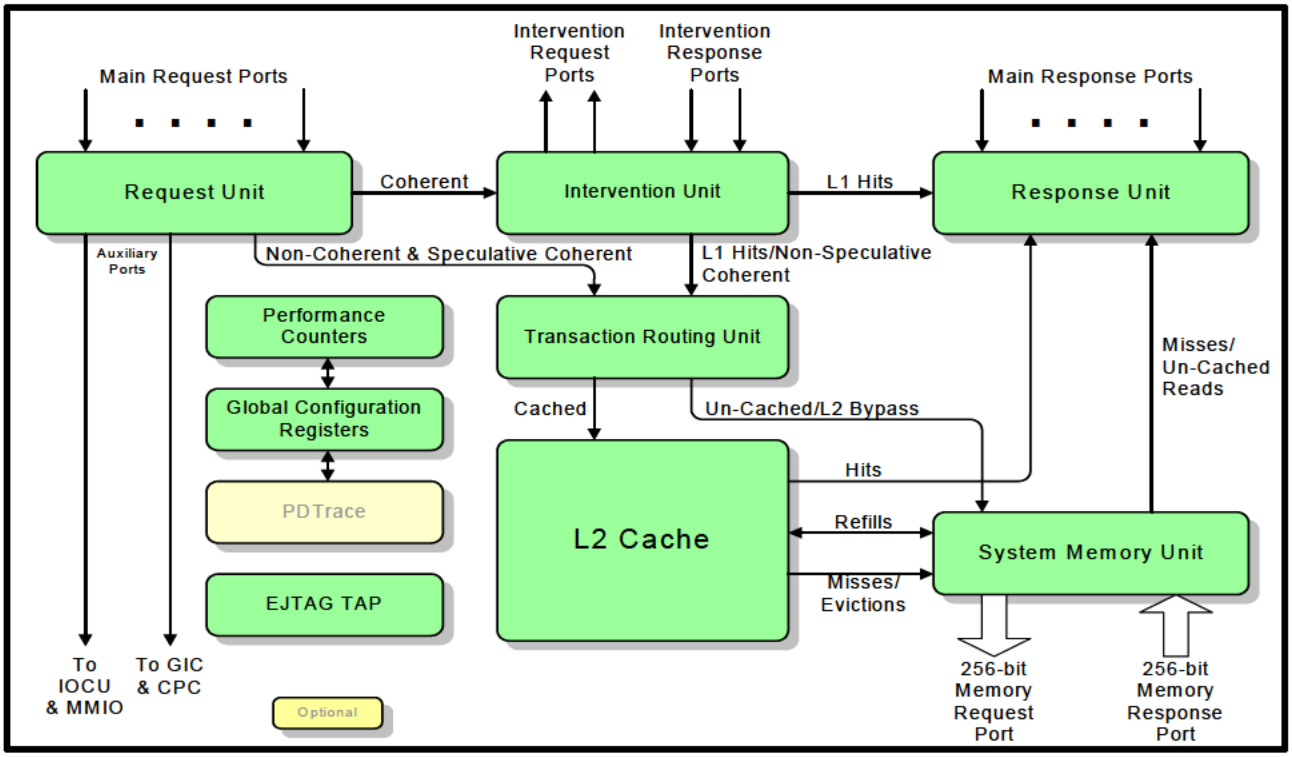
Note that unused CPUs are not held in reset until released by writing into the configuration registers. Rather, unused CPUs remain powered down and are held isolated towards the rest of the cluster. If power gating is not selected for a given implementation, unused CPUs are powered but receive no clock and remain isolated until activated by the CPC.

In addition to controlling the deassertion of the CPC reset signal, there are memory-mapped registers that can set the value for each CPU’s *SI_ExceptionBase* pins. This allows different boot vectors to be specified for each of the cores so they can execute unique code if required. Each of the cores will have a unique CPU number, so it is also possible to use the same boot vector and branch based on that.

1.3.2 Coherence Manager (CM2)

The Coherence Manager with integrated L2 cache (CM2) is responsible for establishing the global ordering of requests and for collecting the intervention responses and sending the correct data back to the requester. A high-level view of the request/response flow through the CM2 is shown in Figure 1.9. Each of the blocks is described in more detail in the following subsections.

Figure 1.9 Coherence Manager with Integrated L2 Cache (CM2) Block Diagram



1.3.2.1 Request Unit (RQU)

The Request Unit (RQU) receives OCP bus transactions from multiple CPU cores and/or I/O ports, serializes the transactions and routes them to the Intervention Unit (IVU), Transaction Routing Unit (TRU), or an auxiliary port used to access a configuration registers or memory-mapped IO. The routing is based on the transaction type, the transaction address, and the CM2's programmable address map.

1.3.2.2 Intervention Unit (IVU)

The Intervention Unit (IVU) interrogates the L1 caches by placing requests on the intervention OCP interfaces. Each processor responds with the state of the corresponding cache line. For most transactions, if a CPU core has the line in the MODIFIED or EXCLUSIVE states, it provides the data with its response. If the original request was a read, the IVU routes the data to the original requestor via the Response Unit (RSU). For the MESI protocol, intervention data may also be routed to the L2/Memory via the TRU (implicit writeback).

The IVU gathers the responses from each of the agents and manages the following actions:

- Speculative reads are resolved (confirmed or cancelled).

- Memory reads that are required because they were not speculative are issued to the Memory Interface Unit (MIU).
- Modified data returned from the CPU is sent to the MIU to be written back to memory.
- Data returned from the CPU is forwarded to the Response Unit (RSU) to be sent to the requester.
- The MESI state in which the line is installed by the requesting CPU is determined (the “install state”). If there are no other CPUs with the data, a Shared request is upgraded to Exclusive.

Each device updates its cache state for the intervention and responds when the state transition has completed. The previous state of the line is indicated in the response. If a read type intervention hits on a line that the CPU has in a Modified or Exclusive state, the CPU returns the cache line with its response. A cacheless device, such as the IOCU, does not require an intervention port. Note that the IVU is not included in non-coherent configurations, such as a single core without an IOCU.

1.3.2.3 System Memory Unit (SMU)

The System Memory Unit (SMU) provides the interface to the memory OCP port. For an L2 refill, the SMU reads the data from an internal buffer and issues the refill request to the L2 pipeline.

Note that the external interface may operate at a lower frequency than the Coherence Manager (CM2), and the external block may not be able to accept as many requests as multiple CPUs can generate, so some buffering of requests may be required.

1.3.2.4 Response Unit (RSU)

The RSU takes responses from the SMU, L2, IVU, or auxiliary port and places them on the appropriate OCP interface. Data from the L2 or SMU is buffered inside a buffer associated with each RSU port, which is an enhancement over the previous generation Coherence Manager.

When a coherent read receives an intervention hit in the MODIFIED or EXCLUSIVE state, the Intervention Unit (IVU) provides the data to the RSU. The RSU then returns the data to the requesting core.

1.3.2.5 Transaction Routing Unit

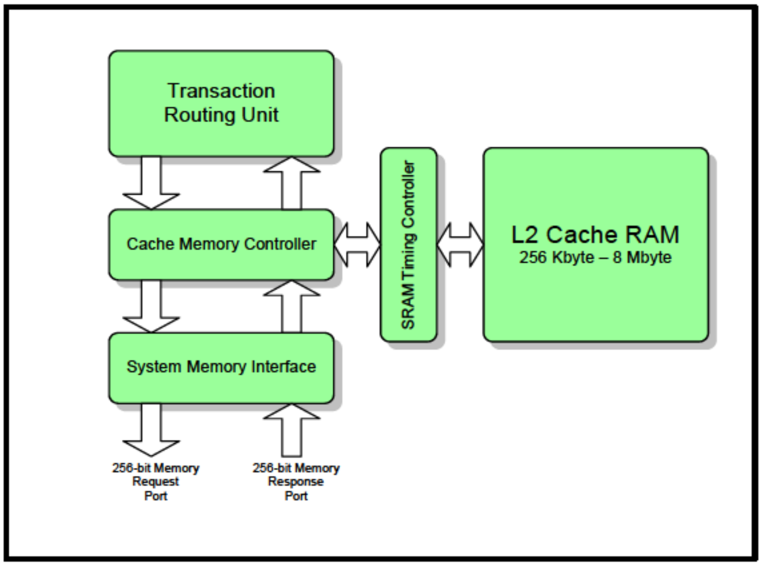
The Transaction Routing Unit (TRU) arbitrates between requests from the RQU and IVU, and routes requests to either the L2 or the SMU. The TRU also contains the request and intervention data buffers which are written directly from the RQU and IVU, respectively. The TRU reads the appropriate write buffer when it processes the corresponding write request.

1.3.2.6 Level 2 Cache

The unified L2 cache holds both instruction and data references and contains a 7-stage pipeline to achieve high frequencies with low power while using commercially available SRAM generators.

Cache read misses are non-blocking; that is, the L2 can continue to process cache accesses while up to 15 misses are outstanding. The cache is physically indexed and physical tagged. Figure 1.10 shows a block diagram of the L2 cache.

Figure 1.10 L2 Cache Block Diagram



L2 Cache Features

- Supports write-back operation.
- Pseudo-LRU replacement algorithm
- Programmable wait state generator to accommodate a wide variety of SRAMs.
- Operates at same clock frequency as CPU.
- Cache line locking support
- Optional ECC support for resilience to soft errors
- Single bit error correction and 2 bit error detection support for Tag and Data arrays
- Single bit detection only for WS array
- Bypass mode
- Fully static design: minimum frequency is 0MHz
- Sleep mode
- Support for extensive use of fine-grained clock gating
- Optional memory BIST for internal SRAM arrays, with support for integrated (March C+, IFA-13) or custom BIST controller

L2 Cache Configuration

The L2 cache in the CM2 can be configured as follows:

- 256 KBytes to 8 MBytes
- 32-byte or 64-byte line size
- 8 ways
- 1024 to 16384 sets per way (in powers of two)

L2 Pipeline Tasks

The L2 pipeline manages the flow of data to and from the L2 cache. The L2 pipeline performs the following tasks:

- Accesses the tags and data RAMs located in the memory block (MEM).
- Returns data to the RSU for cache hits.
- Issues L2 miss requests.
- Issues L2 write and eviction requests.
- Returns L2 write data to the SMU. The SMU issues refill requests to the L2 for installation of data for L2 allocations

1.3.2.7 CM2 Configuration Registers

The Registers block (GCR) contains the control and status registers for the CM2. It also contains the Trace Funnel, EJTAG TAP state machine, and other multi-core features.

The configuration registers in the CM2 allow software to configure and control various aspects of the operation of the CM2. Some of the control options include:

- *Address map*: the base address for the GCR and GIC address ranges can be specified. An additional four address ranges can be defined as well. These control whether non-coherent requests go to memory or to memory-mapped I/O. A default can also be selected for addresses that do not fall within any range.
- *Error reporting and control*: Logs information about errors detected by the CM2 and controls how errors are handled (ignored, interrupt, etc.).
- *Control Options*: Various features of the CM2 can be disabled or configured. Examples of this are disabling speculative reads and preventing read/shared requests from being upgraded to exclusive.

1.3.2.8 PDTrace Unit

The CM2 PDTrace Unit (PDT) is an optional unit used to collect, pack and send out CM2 debug information.

1.3.2.9 Performance Counter Unit

The CM Performance Counter Unit (PERF) implements the performance counter logic.

1.3.2.10 Coherence Manager Performance

The CM2 has a number of high performance features:

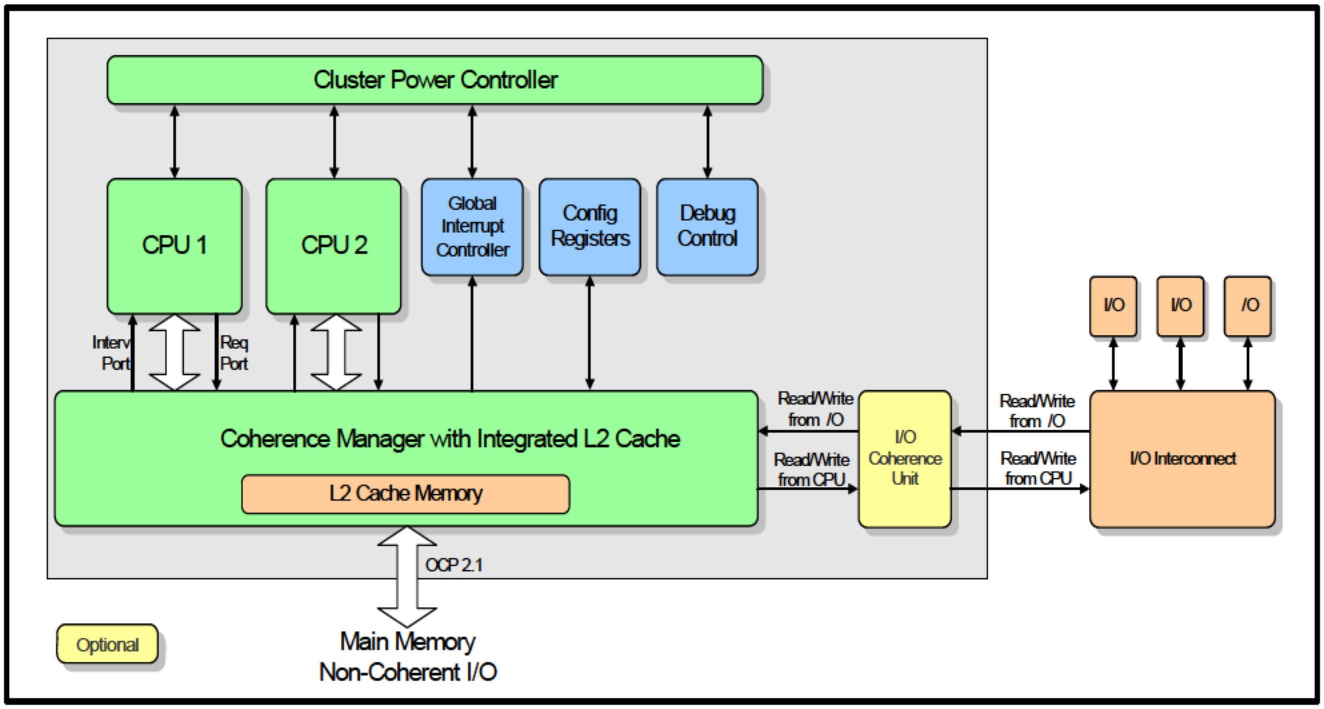
- 256-bit wide internal data paths throughout the CM2
- 256-bit wide system OCP interface
- Cache to Cache transfers: If a read request hits in another L1 cache in the EXCLUSIVE or MODIFIED state, it will return the data to the CM and it will be forwarded to the requesting CPU, thus reducing latency on the miss.
- Speculative Reads: Coherent read requests are forwarded to the memory interface before they are looked up in the other caches. This is speculating that the cache line will not be found in another CPU’s L1 cache. If another cache was able to provide the data, the memory request is not needed, and the CM2 cancels the speculative request—dropping the request if it has not been issued, or dropping the memory response if it has.

1.3.3 I/O Coherence Unit (IOCU)

Optional support for hardware I/O coherence is provided by the I/O Coherence Unit (IOCU), which maintains I/O coherence of the caches in all coherent CPUs in the cluster.

The IOCU acts as an interface block between the Coherence Manager (CM2) and I/O devices. Reads and writes from I/O devices may access the L1 and L2 caches by passing through the IOCU and the CM2. Each request from an I/O device may be marked as coherent, non-coherent cached, or uncached. Coherent requests access the L1 and L2 caches. Non-coherent cached requests access only the L2 cache. Uncached requests bypass both the L1 and L2 caches and are routed to main memory. An example system topology is shown in Figure 1.11.

Figure 1.11 Role of the IOCU in a Two-Core Multiprocessing System



The IOCU also provides a legacy (without coherent extensions) OCP slave interface to the I/O interconnect for I/O devices to read and write system memory. The reference design also includes an OCP Master port to the I/O interconnect that allows the CPUs to access registers and memory on the I/O devices.

The reference IOCU design provides several features for easier integration:

- A user-defined mapping unit can define cache attributes for each request—coherent or not, cacheable (in L2) or not, and L2 allocation policy.
- Supports incremental bursts up to 16 beats (128 bits) on I/O side. These requests are split into cache-line- sized requests on the CM2 side.
- Ensures proper ordering of responses for the split requests and tagged requests.

In addition, the IOCU contains the following features used to enforce transaction ordering.

- Set-aside buffer: This buffer can delay read responses from the I/O device until previous writes have completed.

- Writes are issued to the CM2 in the order they were received.
- The CM2 provides an acknowledge (ACK) signal to the IOCU when writes are "visible" (guaranteed that a subsequent CPU read will receive that data).
  - Non-coherent write is acknowledged after serialization
  - Coherent write is acknowledged after intervention complete on all CPUs
- The IOCU can be configured to treat incoming writes as non-posted and provide a write ACK when they become visible.

When I/O devices access the same memory that is accessed by the processor cores, care must be taken to account for the caches. When an I/O device is reading memory, dirty data in the caches means that main memory may not contain the latest data, and a read directly from memory can receive stale data. When writing main memory, data is the caches becomes stale—the cores can read the stale value and potentially write it back to memory, overwriting the more recent I/O data.

Taking care of these problems can be handled by hardware, software, or a combination of both.

1.3.3.1 Software I/O Coherence

For cases where system redesign to accommodate hardware I/O coherence is not feasible, the CPUs and Coherence Manager provide support for an efficient software-managed I/O coherence. This support is through the globalization of hit-type CACHE instructions.

When a coherent address is used for the CACHE operations, the CPU makes a corresponding coherent request. The CM2 sends interventions for the request to all of the CPUs, allowing all of the L1 caches to be maintained together. The basic software coherence routines developed for single CPU systems can be reused with minimal modifications.

In software managed I/O coherence, software running on the CPU performs any cache operations that are required in accordance with I/O memory accesses. This may include pushing dirty data out of the caches before an I/O read and invalidating stale data after an I/O write. The software I/O coherence code can run on one of the cores and ensure that the appropriate action is taken in all of the caches in the Cluster.

Previous uniprocessor cores from MIPS Technologies have not included support for hardware I/O coherence, and systems based on those cores have relied on software coherence. Generally, the same coherence routines will work on the multi-CPU system.

1.3.3.2 Hardware I/O Coherence

For hardware I/O coherence, the coherence features on the CPU are used to ensure that I/O requests are handled properly. Requests from I/O devices go to the I/O Coherence Unit (IOCU) and then to a request port of the Coherence Manager. Requests that are marked coherent will generate interventions to the cores. I/O read requests can obtain any dirty data directly from a data cache that has it in the M state. I/O write requests will invalidate the line in any data caches that have copies of it. Coherent requests access the in-line L2 cache from the CM, so they will automatically be coherent with the L2 cache.

Note that I/O interventions do not affect the instruction cache. The instruction cache cannot contain dirty data, so I/O reads are not a problem. However, if the I/O device is writing addresses that may reside in the instruction cache, software coherence must be used to invalidate the stale cache data.

In addition, even if hardware I/O coherence is present, there may be a need for software to explicitly maintain coherence. Examples of this are for systems configured without I/O coherence, devices that are not connected to the coherent port, or devices that directly access memory non-coherently. Initially, with the non-coherent I-Cache, this will also be needed to maintain I-Cache coherence with I/O traffic and data operations.

1.3.4 Global Interrupt Controller

The Global Interrupt Controller (GIC) handles the distribution of interrupts between and among the CPUs in the cluster. This block has the following features:

- Software interface through relocatable memory-mapped address range.
- Configurable number of system interrupts - from 8 to 256 in multiples of 8.
- Support for different interrupt types:
  - Level-sensitive: active high or low.
  - Edge-sensitive: positive, negative, or double-edge-sensitive.
- Ability to mask and control routing of interrupts to a particular CPU.
- Support for NMI routing.
- Standardized mechanism for sending inter-processor interrupts.

1.3.5 Global Configuration Registers (GCR)

The Global Configuration Registers (GCR) are a set of memory-mapped registers that are used to configure and control various aspects of the Coherence Manager and the coherence scheme.

1.3.5.1 Reset Control

The reset input of the system resets the Cluster Power Controller (CPC). Reset sideband signals are required to qualify a reset as system cold, or warm start. Register setting determine the course of action:

- Remain in powered-down
- Go into clock-off mode
- Power-up and start execution

This prevents random power up of power domains before the CPC is properly initialized. In case of a system cold start, after reset is released, the CPC powers up the proAptiv CPUs as directed in the CPC cold start configuration. If at least one CPU has been chosen to be powered up on system cold start, the CM2 is also powered up.

When supply rail conditions of power gated CPUs have reached a nominal level, the CPC will enable clocks and schedule reset sequences for those CPUs and the coherence manager.

At a warm start reset, the CPC brings all power domains into their cold start configuration. However, to ensure power integrity for all domains, the CPC ensures that domain isolation is raised before power is gated off. Domains that were previously powered and are configured to power up at cold start remain powered and go through a reset sequence.

Within a warm start reset, sideband signals are also used to qualify if coherence manager status registers and GIC watch dog timers are to be reset or remain unchanged. The CPC, after power up of any CPU, provides a test logic reset sequence per domain to initialize TAP and PDTrace logic.

Note that unused CPUs are not held in reset until released by writing into the configuration registers. Rather, unused CPUs remain powered down and are held isolated towards the rest of the cluster. If power gating is not selected for a given implementation, unused CPUs are powered but receive no clock and remain isolated until activated by the CPC.

In addition to controlling the deassertion of the CPC reset signal, there are memory-mapped registers that can set the value for each CPU's *SI_ExceptionBase* pins. This allows different boot vectors to be specified for each of the cores so they can execute unique code if required. Each of the cores will have a unique CPU number, so it is also possible to use the same boot vector and branch based on that.

1.3.5.2 Inter-CPU Debug Breaks

The CPS includes registers that enable cooperative debugging across all CPUs. Each core features an *EJ_DebugM* output that indicates it has entered debug mode (possibly through a debug breakpoint). Registers are defined that allow CPUs to be placed into debug groups such that whenever one CPU within the group enters debug mode, a debug interrupt is sent to all CPUs within the group, causing them to also enter debug mode and stop executing non-debug mode instructions.

1.3.5.3 CM2 Control Registers

Control registers in the CM2 allow software to configure and control various aspects of the operation of the CM2. Some of the control options include:

- *Address map*: the base address for the GCR and GIC address ranges can be specified. An additional four address ranges can be defined as well. These control whether non-coherent requests go to memory or to memory-mapped I/O. A default can also be selected for addresses that do not fall within any range.
- *Error reporting and control*: Logs information about errors detected by the CM2 and controls how errors are handled (ignored, interrupt, etc.).
- *Control Options*: Various features of the CM2 can be disabled or configured. Examples of this are disabling speculative reads and preventing ReadShared requests from being upgraded to Exclusive.

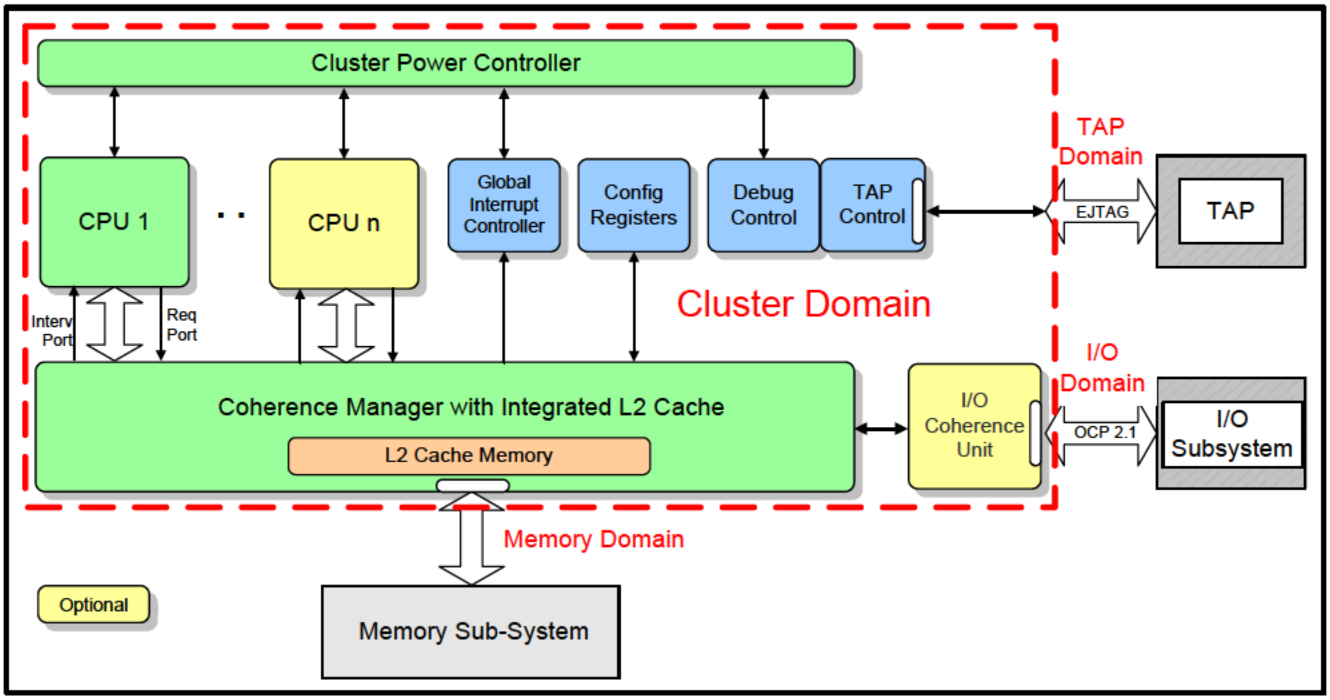
1.3.6 Clocking Options

The proAptiv core has the following clock domains:

- Cluster domain — This is the main clock domain, and includes all proAptiv cores (including optional FP2) and the CM2 (including Coherence Manager, Global Interrupt Controller, Cluster Power Controller, trace funnel, IOCU, and L2 cache).
- System Domain - The OCP port connecting to the SOC and the rest of the memory subsystem may operate at a ratio of the cluster domain. Supported ratios are 1:1, 1:1.5, 1:2, 1:2.5, 1:3, 1:3.5, 1:4, 1:5, and 1:10.
- TAP domain - This is a low-speed clock domain for the EJTAG TAP controller, controlled by the *EJ_TCK* pin. It is asynchronous to *SI_ClkIn*.
- IO Domain - This is the OCP port connecting the IOCU to the I/O Subsystem. This clock may operate at a ratio of the CM2 domain. Supported ratios are the same as the system domain.

Figure 1.12 shows a diagram with the four clock domains.

Figure 1.12 proAptiv CPS Clocking Domains



1.3.7 Design For Test (DFT) Features

The proAptiv core provides the following tests for determining the integrity of the core. For more information, refer to the Test and Debug chapter.

1.3.7.1 Internal Scan

The proAptiv core supports full mux-based scan for maximum test coverage, with a configurable number of scan chains. ATPG test coverage can exceed 99%, depending on standard cell libraries and configuration options.

1.3.7.2 Memory BIST

The proAptiv core provides an integrated memory BIST solution for testing of all internal SRAMs. These BIST controllers can be configured to utilize the March C+ or IFA-13 algorithms.

Memory BIST can also be inserted with a CAD tool or other user-specified method. Wrapper modules and signal buses of configurable width are provided within the core to facilitate this approach.

1.4 Build-Time Configuration Options

The proAptiv Multiprocessing System allows a number of features to be customized based on the intended application. Refer to the proAptiv data sheet for more information on the configuration options.

Chapter 2

CP0 Registers

The proAptiv System Control Coprocessor (CP0) provides the register interface to the proAptiv 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.* A separate *select number* is used to differentiate additional registers within the *register number*. For example, as shown in the table below, there are eight configuration registers with register number 16. If the *select number* is omitted, it is zero.

This chapter contains the following sections:

- Section 2.1 “CP0 Register Summary”
- Section 2.2 “CP0 Register Formats”
- Section 2.3 “CP0 Register Descriptions”

2.1 CP0 Register Summary

The following two subsections show the CP0 register set grouped by function and grouped by number.

2.1.1 CP0 Registers Grouped by Function

The CP0 registers set are divided into the register groups shown in Table 2.1. Note that assembly programmers modifying certain CP0 registers or register fields must clear any execution or instruction hazards created by the modification. Refer to Section 6.3.1, "Hazard Barrier Instructions" in Chapter 6 for more information on hazard barrier instructions.

The following table provides a functional listing of the CP0 registers. Click on a Name column entry to provide a link to the desired register.

Table 2.1 proAptiv CP0 Registers Grouped by Function

Category	Register Name	Register Number	Register Select	Location in Document
CPU Configuration and Status	Config	16	0	Section 2.3.1.1 on page 82
	Config1	16	1	Section 2.3.1.2 on page 84
	Config2	16	2	Section 2.3.1.3 on page 87
	Config3	16	3	Section 2.3.1.4 on page 89
	Config4	16	4	Section 2.3.1.5 on page 90
	Config5	16	5	Section 2.3.1.6 on page 92
	Config6	16	6	Section 2.3.1.7 on page 93
	Config7	16	7	Section 2.3.1.8 on page 96
	PRId	15	0	Section 2.3.1.9 on page 100
	EBase	15	1	Section 2.3.1.10 on page 100
	Status	12	0	Section 2.3.1.11 on page 102
	IntCtl	12	1	Section 2.3.1.12 on page 106
TLB Management	Index	0	0	Section 2.3.2.1 on page 108
	Random	1	0	Section 2.3.2.2 on page 109
	EntryLo0	2	0	Section 2.3.2.3 on page 110
	EntryLo1	3	0	
	EntryHi	10	0	Section 2.3.2.4 on page 112
	Context	4	0	Section 2.3.2.5 on page 113
	ContextConfig	4	1	Section 2.3.2.6 on page 114
	PageMask	5	0	Section 2.3.2.7 on page 115
	PageGrain	5	1	Section 2.3.2.8 on page 116
	Wired	6	0	Section 2.3.2.9 on page 117
	BadVAddr	8	0	Section 2.3.2.10 on page 118
Memory Segmentation	SegCtl0	5	2	Section 2.3.3.1 on page 119
	SegCtl1	5	3	Section 2.3.3.2 on page 120
	SegCtl2	5	4	Section 2.3.3.3 on page 121
Exception Control	Cause	13	0	Section 2.3.4.1 on page 124
	EPC	14	0	Section 2.3.4.2 on page 127
	ErrorEPC	30	0	Section 2.3.4.3 on page 128
Timer	Count	9	0	Section 2.3.5.1 on page 129
	Compare	11	0	Section 2.3.5.2 on page 129

Table 2.1 proAptiv CP0 Registers Grouped by Function *(continued)*

Category	Register Name	Register Number	Register Select	Location in Document
Cache Management	ITagLo	28	0	Section 2.3.6.1 on page 130
	ITagHi	29	0	Section 2.3.6.2 on page 132
	IDataLo	28	1	Section 2.3.6.3 on page 133
	IDataHi	29	1	Section 2.3.6.4 on page 133
	DTagLo	28	2	Section 2.3.6.5 on page 134
	DDataLo	28	3	Section 2.3.6.6 on page 140
	L23TagLo	28	4	Section 2.3.6.7 on page 141
	L23DataLo	28	5	Section 2.3.6.8 on page 141
	L23DataHi	29	5	Section 2.3.6.9 on page 142
	ErrCtl	26	0	Section 2.3.6.10 on page 142
	CacheErr	27	0	Section 2.3.6.11 on page 144
Shadow Registers	SRSCtl	12	2	Section 2.3.7.1 on page 146
Performance Monitoring	PerfCtl0	25	0	Section 2.3.8.1 on page 149
	PerfCtl1	25	2	
	PerfCtl2	25	4	
	PerfCtl3	25	6	
	PerfCnt0	25	1	Section 2.3.8.2 on page 158
	PerfCnt1	25	3	
	PerfCnt2	25	5	
	PerfCnt3	25	7	
Debug	Debug	23	0	Section 2.3.9.1 on page 158
	DEPC	24	0	Section 2.3.9.2 on page 161
	DESAVE	31	0	Section 2.3.9.3 on page 162
	WatchLo0	18	0	Section 2.3.9.4 on page 162
	WatchLo1	18	1	
	WatchLo2	18	2	
	WatchLo3	18	3	
	WatchHi0	19	0	Section 2.3.9.5 on page 163
	WatchHi1	19	1	
	WatchHi2	19	2	
	WatchHi3	19	3	

Table 2.1 proAptiv CP0 Registers Grouped by Function *(continued)*

Category	Register Name	Register Number	Register Select	Location in Document
PDTrace	TraceControl	23	1	Section 2.3.10.1 on page 164
	TraceControl2	23	2	Section 2.3.10.2 on page 167
	TraceControl3	24	2	Section 2.3.10.3 on page 169
	UserTraceData1	23	3	Section 2.3.10.4 on page 170
	UserTraceData2	24	3	Section 2.3.10.5 on page 170
	TraceIPBC	23	4	Section 2.3.10.6 on page 170
	TraceDBPC	23	5	Section 2.3.10.7 on page 171
User Mode Support	HWREna	7	0	Section 2.3.11.1 on page 172
	UserLocal	4	2	Section 2.3.11.2 on page 174
Kernel Mode Support	KScratch0	31	2	Section 2.3.12.1 on page 175
	KScratch1	31	3	Section 2.3.12.2 on page 175
	KScratch2	31	4	Section 2.3.12.3 on page 175
Memory Mapped	CDMMBase	15	2	Section 2.3.13.1 on page 176
	CMGCRBase	15	3	Section 2.3.13.2 on page 177

2.1.2 CP0 Registers Grouped by Number

The following table provides a numerical listing of the proAptiv CP0 registers. Click on a Name column entry to provide a link to the desired register.

Table 2.2 CP0 Registers Grouped by Number

Register			Function	Location
Num	Sel	Name		
0	0	Index	Index into the TLB array	Section 2.3.2.1
1	0	Random	Randomly generated index into the TLB array.	Section 2.3.2.2
2	0	EntryLo0	Low-order portion of the TLB entry for even-numbered virtual pages.	Section 2.3.2.3
3	0	EntryLo1	Low-order portion of the TLB entry for odd-numbered virtual pages.	
4	0	Context	Pointer to page table entry in memory.	Section 2.3.2.5
4	1	ContextConfig	Defines the bits of the Context register into which the high order bits of the virtual address causing a TLB exception will be written, and how many bits of that virtual address will be extracted.	Section 2.3.2.6
4	2	UserLocal	User information that can be written by privileged software and read via RDHWR register 29	Section 2.3.11.2
5	0	PageMask	PageMask controls the variable page sizes in TLB entries.	Section 2.3.2.7
5	1	PageGrain	PageGrain controls the granularity of the page sizes in TLB entries.	Section 2.3.2.7
5	2	SegCtl0	Segmentation control register 0. Used for enhanced virtual addressing (EVA).	Section 2.3.3.1
5	3	SegCtl1	Segmentation control register 1. Used for enhanced virtual addressing (EVA).	Section 2.3.3.2
5	4	SegCtl2	Segmentation control register 2. Used for enhanced virtual addressing (EVA).	Section 2.3.3.3
6	0	Wired	Controls the number of fixed ("wired") TLB entries. This register is reserved if the TLB is not implemented.	Section 2.3.2.9
7	0	HWREna	Enables access via the RDHWR instruction to selected hardware registers in non-privileged mode.	Section 2.3.11.1
8	0	BadVAddr	Reports the address for the most recent address-related exception.	Section 2.3.2.10
9	0	Count	Processor cycle count.	Section 2.3.5.1
10	0	EntryHi	High-order portion of the TLB entry. This register is reserved if the TLB is not implemented.	Section 2.3.2.4
11	0	Compare	Timer interrupt control.	Section 2.3.5.2
12	0	Status	Processor status and control.	Section 2.3.1.11
12	1	IntCtl	Setup for interrupt vector and interrupt priority features.	Section 2.3.1.12
12	2	SRSCtl	Shadow register set control.	Section 2.3.7.1
13	0	Cause	Cause of last exception.	Section 2.3.4.1
14	0	EPC	Program counter at last exception.	Section 2.3.4.2
15	0	PRId	Processor identification and revision.	Section 2.3.1.9
15	1	EBase	Exception base address.	Section 2.3.1.10
15	2	CDMMBase	Common Device Memory Map Base Address.	Section 2.3.13.1

Table 2.2 CP0 Registers Grouped by Number *(continued)*

Register				
Num	Sel	Name	Function	Location
15	3	CMGCRBase	Defines the 36-bit physical base address for the memory-mapped Coherency Manager Global Configuration Register (CMGCR) space.	Section 2.3.13.1
16	0	Config	Configuration register.	Section 2.3.1.1
16	1	Config1	Configuration for MMU, caches etc.	Section 2.3.1.2
16	2	Config2	Configuration for MMU, caches etc.	Section 2.3.1.3
16	3	Config3	Interrupt and ASE capabilities	Section 2.3.1.4
16	4	Config4	Indicates presence of Config5 register	Section 2.3.1.5
16	5	Config5	Provides information on EVA and cache error exception vector.	Section 2.3.1.6
16	5	Config6	Provides information about the presence of optional extensions to the base MIPS32 architecture.	Section 2.3.1.7
16	7	Config7	proAptiv Multiprocessing System family-specific configuration register.	Section 2.3.1.8
18	0	WatchLo0	Watchpoint address associated with instruction watchpoint 0.	Section 2.3.9.4
18	1	WatchLo1	Watchpoint address associated with instruction watchpoint 1.	
18	2	WatchLo2	Watchpoint address associated with data watchpoints 0.	
18	3	WatchLo3	Watchpoint address associated with data watchpoints 1.	
19	0	WatchHi0	Watchpoint ASID and Mask associated with instruction watchpoint 0.	Section 2.3.9.5
19	1	WatchHi1	Watchpoint ASID and Mask associated with instruction watchpoint 1.	
19	2	WatchHi2	Watchpoint ASID and Mask associated with data watchpoint 0.	
19	3	WatchHi3	Watchpoint ASID and Mask associated with data watchpoint 1.	
23	0	Debug	EJTAG Debug register.	Section 2.3.9.1
23	1	TraceControl	EJTAG Trace Control register	Section 2.3.10.1
23	2	TraceControl2	EJTAG Trace Control2 register	Section 2.3.10.2
23	3	UserTraceData1	EJTAG User Trace Data1 register	Section 2.3.10.4
23	4	TraceIBPC	EJTAG Trace Instruction breakpoint control register	Section 2.3.10.6
23	5	TraceDBPC	EJTAG Trace Debug breakpoint control register	Section 2.3.10.7
24	0	DEPC	Restart address from last EJTAG debug exception.	Section 2.3.9.2
24	2	TraceControl3	EJTAG Trace Control3 register	Section 2.3.10.3
24	3	UserTraceData2	EJTAG User Trace Data2 register	Section 2.3.10.5
25	0	PerfCtl0	Performance counter 0 control.	Section 2.3.8.1
25	1	PerfCnt0	Performance counter 0 count.	Section 2.3.8.2
25	2	PerfCtl1	Performance counter 1 control.	Section 2.3.8.1
25	3	PerfCnt1	Performance counter 1 count.	Section 2.3.8.2
25	4	PerfCtl2	Performance counter 2 control.	Section 2.3.8.1
25	5	PerfCnt2	Performance counter 2 count.	Section 2.3.8.2
25	6	PerfCtl3	Performance counter 3 control.	Section 2.3.8.1
25	7	PerfCnt3	Performance counter 3 count.	Section 2.3.8.2

Table 2.2 CP0 Registers Grouped by Number *(continued)*

Register			Function	Location
Num	Sel	Name		
26	0	ErrCtl	Software test enable of way-select and Data RAM arrays for I-Cache and D-Cache.	Section 2.3.6.10
27	0	CacheErr	Records information about cache parity errors	Section 2.3.6.11
28	0	ITagLo	Cache tag read/write interface for I-cache.	Section 2.3.6.1
28	1	IDataLo	Low-order data read/write interface for I-cache.	Section 2.3.6.3
28	2	DTagLo	Cache tag read/write interface for D-cache.	Section 2.3.6.5
28	3	DDataLo	Low-order data read/write interface for D-cache.	Section 2.3.6.6
28	4	L23TagLo	Cache tag read/write interface for L2-cache.	Section 2.3.6.7
28	5	L23DataLo	Low-order data read/write interface for L2-cache.	Section 2.3.6.8
28	0	ITagHi	Cache tag read/write interface for I-cache, upper 32 bits.	Section 2.3.6.1
29	1	IDataHi	High-order data read/write interface for I-cache.	Section 2.3.6.4
29	5	L23DataHi	High-order data read/write interface for L2-cache.	Section 2.3.6.9
30	0	ErrorEPC	Program counter at last error.	Section 2.3.4.3
31	0	DESAVE	Debug handler scratchpad register.	Section 2.3.9.3
31	2	KScratch0	Kernel scratch pad register 0.	Section 2.3.12.1
31	3	KScratch1	Kernel scratch pad register 1.	Section 2.3.12.2
31	4	KScratch2	Kernel scratch pad register 2.	Section 2.3.12.3

2.2 CP0 Register Formats

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

2.2.1 CP0 Register Field Types

For each register described below, field descriptions include the read/write properties of the field, and the reset state of the field.

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 2.3.

Table 2.3 CP0 Register Field R/W Access Types

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.

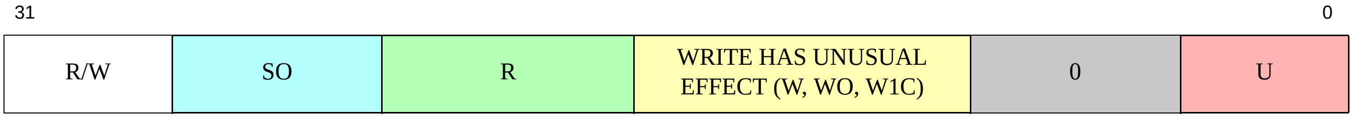
Table 2.3 CP0 Register Field R/W Access Types*(continued)*

Notation	Hardware Interpretation	Software Interpretation
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 2.1. A field with two access types (for example, R/W0) is uncolored,

Figure 2.1 Register Format Color Coding of Access Field Types



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.

2.2.2 Value Notations

The following conventions are used for numeric values in this document:

- Decimal values are written as standard base 10 numbers.
- Hexadecimal values are prefaced with “0x”.

- Binary numbers are appended with the prefix *x*’b*y*. The number preceding the ‘b (*x*) indicates the number of binary bits involved. The numbers after the ‘b (*y*) indicates the actual binary value. The number of bits of *y* should match the value stated in *x*.
    - For example, the following numbers are equivalent: 17 decimal = 0x11 = 5’b10001.
    - If the (*y*) value is the same for all bits, only one number need be shown. For example, 12’b0 would indicate there are 12 bits in the field and all of them are zero.

2.3 CP0 Register Descriptions

The following subsections describe the CP0 registers listed in Table 2.1 above.

2.3.1 CPU Configuration and Status Registers

This section contains the following CPU Configuration and Status registers.

- Section 2.3.1.1, "Device Configuration — Config (CP0 Register 16, Select 0)" on page 82
- Section 2.3.1.2, "Device Configuration 1 — Config1 (CP0 Register 16, Select 1)" on page 84
- Section 2.3.1.3, "Device Configuration 2 — Config2 (CP0 Register 16, Select 2)" on page 87
- Section 2.3.1.4, "Device Configuration 3 — Config3 (CP0 Register 16, Select 3)" on page 89
- Section 2.3.1.5, "Device Configuration 4 — Config4 (CP0 Register 16, Select 4)" on page 90
- Section 2.3.1.6, "Device Configuration 5 — Config5 (CP0 Register 16, Select 5)" on page 92
- Section 2.3.1.7, "Device Configuration 6 — Config6 (CP0 Register 16, Select 6)" on page 93
- Section 2.3.1.8, "Device Configuration 7 — Config7 (CP0 Register 16, Select 7)" on page 96
- Section 2.3.1.9, "Processor ID — PRId (CP0 Register 15, Select 0)" on page 100
- Section 2.3.1.10, "Exception Base Address — EBase (CP0 Register 15, Select 1)" on page 100
- Section 2.3.1.11, "Status (CP0 Register 12, Select 0)" on page 102
- Section 2.3.1.12, "Interrupt Control — IntCtl (CP0 Register 12, Select 1)" on page 106

2.3.1.1 Device Configuration — Config (CP0 Register 16, Select 0)

The main role of the *Config* register is to be a read-only repository of information about the proAptiv core resources, encoded so as to be useful to operating system initialization code.

Figure 2.2 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 2.4 Field Descriptions for Config Register

Name	Bit(s)	Description	Read/ Write	Reset State
M	31	This bit is hardwired to ‘1’ to indicate the presence of the *Config1* register.	R	1
K23	30:28	These fields are unused in the proAptiv core since the TLB structure is supported. They should be written as zero only.	R	0
KU	27:25		R	0

Table 2.4 Field Descriptions for Config Register*(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
ISP	24	Instruction Scratch Pad RAM present. 0: Iinstruction scratch pad RAM *(*ISPRAM*)* is not implemented. 1: Instruction scratch pad RAM *(*ISPRAM*)* is implemented.	R	Preset
DSP	23	Data Scratch Pad RAM present. 0: Data scratch pad RAM *(*DSPRAM*)* is not implemented. 1: Data scratch pad RAM *(*DSPRAM*)* is implemented. This bit should not be confused with the MIPS DSP ASE, whose presence is indicated by $Config3_{DSPP}$.	R	Preset
UDI	22	User-Defined Instrucitons. 0: The proAptiv core does not contain user-defined "CorExtend" instructions. 1: The proAptiv core contains user-defined "CorExtend" instructions.	R	Preset
SB	21	Read-only "SimpleBE" bus mode indicator, which reflects the proAptiv Multiprocessing System input signal *SI_SimpleBE*. 0: No reserved byte enabled on the OCP interface. 1: Only simple byte enables allows on the OCP interface. If set by hardware, the proAptiv Multiprocessing System 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. This generates less requests, but may not be supported by all downstream devices.	R	Externally Set
0	20:19	Must be written as zero; returns zero on read.	R	0
MM	18	Write Merge.This bit indicates whether write-through merging is enabled in the 32-byte collapsing write buffer. 0: No merging allowed 1: Merging allowed Setting this bit allows writes resulting from separate store instructions in write-through mode to be merged into a single transaction at the interface. The state of this bit does not affect cache writebacks (which are always whole blocks together) or uncached writes (which are never merged). Note that write-through caching is not supported in the proAptiv core, so this bit has no meaning.	R/W	1
0	17	Must be written as zero; returns zero on read.	R	0

Table 2.4 Field Descriptions for Config Register*(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
BM	16	Burst Mode. 0: Sequential burst mode 1: SubBlock burst mode This bit reads 0 when the bus uses sequential burst ordering and reads 1 when it uses sub-block burst ordering. This bit is set by the input signal *SI_SBlock* signal to match the system controller. Note that the proAptiv core only supports sequential burst ordering. Hence this bit is always zero.	R	0
BE	15	Endian mode. 0: Little endian 1: Big endian This bit is written by hardware based on the state of the *SI_Endian* input pin.	R	Externally Set
AT	14:13	Architecture type implemented by the processor. This field is always 00 to indicate the MIPS32 architecture.	R	0
AR	12:10	Architecture release. 0x0 = Release 1 0x1 = Release 2 or Release 3 This bit always reads 1 to reflect Release 3 of the MIPS32 architecture.	R	1
MT	9:7	MMU type: 000: Reserved 001: VTLB Only 010 - 011: Reserved 100: VTLB + FTLB 101 - 111: Reserved	R	1
0	6:4	Must be written as zero; returns zero on read.	R	0
VI	3	Virtually indexed. This bit is set by hardware and is 0 to indicate that the L1 instruction cache is physically tagged.	R	0
K0	2:0	Kseg0 coherency attribute of the page. See Table 2.21 for the field encoding.	R/W	2

2.3.1.2 Device Configuration 1 — Config1 (CP0 Register 16, Select 1)

The Config1 register provides information such as the size of the VTLB and the L1 instruction and data cache parameters. It also contains a series of single bits that indicate the presence of selected logic units on the proAptiv core.

Figure 2.3 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 2.5 Field Descriptions for Config1 Register

Name	Bit(s)	Description	Read/ Write	Reset State
M	31	Continuation bit, set to 1 to indicate that the *Config2* register is implemented.	R	1
MMUSize	30:25	The size of the VTLB array (the array has MMUSize + 1 entries). Refer to the *Config4* register for more information.	R	0x3F
IS	24:22	L1 Instruction cache number of sets per way. This field indicates the number of sets per way in the L1 instruction cache. The number of sets is multiplied by the number of ways and the line size to derive the cache size. In this case, the number of sets defines the cache size since the line size and number of ways in the proAptiv core are fixed. This field is encoded as follows: 000 - 001: Reserved 010: 256 sets per way (equates to 32 KByte instruction cache) 011: 512 sets per way (equates to 64 KByte instruction cache) 100 - 111: Reserved Because the line size and associativity are fixed for the proAptiv instruction cache as defined in the IL and IA fields below, the IS field is used to determine the overall cache size as follows: If this field is set to 2, the instruction cache size would be: 256 sets/way x 32 bytes/line x 4 sets per way = 32 KBytes. If this field is set to 3, the instruction cache size would be: 512 sets/way x 32 bytes/line x 4 sets per way = 64 KBytes.	R	Preset
IL	21:19	L1 Instruction cache line size. In the proAptiv core, the instruction cache line size is fixed at 32 bytes. As such, this field is encoded as follows: 000 - 011: Reserved 100: 32 byte line size 101 - 111: Reserved A value of zero in this field indicates means no cache.	R	4
IA	18:16	L1 Instruction cache associativity. In the proAptiv core, the instruction cache associativity is fixed at 4 ways. As such, this field is encoded as follows: 000 - 010: Reserved 011: 4-ways 100 - 111: Reserved A default value of 3 indicates a 4-way set associative instruction cache. Refer to the IS field above to determine how to calculate the size of the L1 instruction cache.	R	3

Table 2.5 Field Descriptions for Config1 Register

Name	Bit(s)	Description	Read/ Write	Reset State
DS	15:13	L1 Data cache number of sets per way. This field indicates the number of sets per way in the L1 data cache and is encoded as follows: The number of sets is multiplied by the number of ways and the line size to derive the cache size. In this case, the number of sets defines the cache size since the line size and number of ways in the proAptiv core are fixed. This field is encoded as follows: 000 - 001: Reserved 010: 256 sets per way (equates to 32 KByte instruction cache) 011: 512 sets per way (equates to 64 KByte instruction cache) 100 - 111: Reserved Because the line size and associativity are fixed for the proAptiv data cache as defined in the DL and DA fields below, the DS field is used to determine the overall cache size as follows: If this field is set to 2, the data cache size would be: 256 sets/way x 32 bytes/line x 4 sets per way = 32 KBytes. If this field is set to 3, the data cache size would be: 512 sets/way x 32 bytes/line x 4 sets per way = 64 KBytes.	R	Preset
DL	12:10	L1 data cache line size. In the proAptiv core, the data cache line size is fixed at 32 bytes. As such, this field is encoded as follows: 000 - 011: Reserved 100: 32 byte line size 101 - 111: Reserved A value of zero in this field indicates means no cache.	R	4
DA	9:7	L1 data cache associativity. In the proAptiv core, the data cache associativity is fixed at 4 ways. As such, this field is encoded as follows: 000 - 010: Reserved 011: 4-ways 100 - 111: Reserved A default value of 3 indicates a 4-way set associative data cache.	R	3
C2	6	This bit is cleared to indicate that a coprocessor 2 does not exist in the system.	R	0
MD	5	MDMX Application Specific Extension (ASE). A logic ‘0’ indicates that the MDMX ASE is not implemented in the floating point unit (FPU) of the proAptiv core. Note that if the FPU is not implemented, this bit has no meaning.	R	0
PC	4	Performance counter enable. There is at least one performance counter implemented in the proAptiv core. Hence this bit is always a logic ‘1’. Refer to the *PerfCtl0-3* and *PerfCnt0-3* registers for more information.	R	1
WR	3	Watchpoint registers present. This bit always reads 1 because the proAptiv core always has watchpoint registers. Refer to the *WatchLo 0-3*/*WatchHi 0-3* registers in Section 2.3.9.4 “Watch Low 0 - 3 — WatchLo0-3 (CP0 Register 18, Select 0-3)”.	R	1

Table 2.5 Field Descriptions for Config1 Register

Name	Bit(s)	Description	Read/ Write	Reset State
CA	2	MIPS16e present. This bit always reads 1 to indicate the MIPS16e compressed-code instruction set is available.	R	1
EP	1	EJTAG unit present. This bit always reads 1 as the EJTAG debug unit is provided on the proAptiv core.	R	1
FP	0	Floating Point Unit present. This bit is set to indicate that a floating point unit is present. The floating point unit is optional on the proAptiv core. If no FPU is present, this bit will be zero. •	R	Preset

2.3.1.3 Device Configuration 2 — Config2 (CP0 Register 16, Select 2)

The *Config2* register provides information about the size and organization of L2 and L3 caches. The *Config2* register also has fields that indicate the presence of some extensions to the base MIPS32 architecture.

An L3 cache can be used with the proAptiv Multiprocessing System core. However, the core does not support passing of the L3 configuration information via the Config2 register. As such, the TU, TS, TL and TA bits of this register, which handle L3 operations, are not used and are all tied to 0. Information on L3 transfers may be available in an implementation specific register elsewhere in the system.

Figure 2.4 Config2 Register Format

31	30 28	27 24	23 20	19 16	15 13	12	11 8	7 4	3 0
M	TU	TS	TL	TA	SU	L2B	SS	SL	SA

Table 2.6 Field Descriptions for Config2 Register

Name	Bit(s)	Description	Read/ Write	Reset State
M	31	This bit is hardwired to ‘1’ to indicate the presence of the Config3 register.	R	1
TU	30:28	An L3 cache can be used with the proAptiv core. However, the core does not support passing of the L3 configuration data via the Config2 register. As such, the TU, TS, TL and TA bits of this register, which report L3 information, are not used and are all tied to 0. Details of the L3 configuration may be available in an implementation specific register elsewhere in the system.	R	0
TS	27:24		R	0
TL	23:20		R	0
TA	19:16		R	0
SU	15:13	This bit is reserved in the proAptiv core and is always 0.	R	0
L2B	12	L2 cache bypass. Setting this bit disables or bypasses the L2 cache. Setting this bit also forces *Config2*$_{SL}$ to 0. Based on this information, most operating system code will conclude that there is no L2 cache on the system. Setting this bit forces hardware to drivea series of internal handshake signals between the core to the CM2, placing the L2 cache into bypass mode. When this bit is set through a write operation, a subsequent read of this bit will not indicate a logic 1 until the L2 has asserted its internal handshakle signal, indicating that it has been bypassed.	R/W	0

Table 2.6 Field Descriptions for Config2 Register

Name	Bit(s)	Description	Read/ Write	Reset State
SS	11:8	L2 cache number of sets per way. This field indicates the number of sets per way in the L2 cache of the Coherent Processing System (CPS) and is written by hardware at reset based on the state of the *L2_Sets[3:0]* signals. At IP configuration time, the user selects the cache size and the line size. Hardware then takes this information and selects the appropriate number of sets. See the example formulas below for determining the number of sets based on cache and line size. This field is encoded as follows: 0x0 - 0x2: Reserved 0x3: 512 sets per way 0x4: 1024 sets per way 0x5: 2048 sets per way 0x6: 4096 sets per way 0x7: 8192 sets per way 0x8: 16384 sets per way 0x9: 32768 sets per way 0xA- 0xF: Reserved For example: If this field is set to 0x3, the SL field is set to 0x5, and the SA field is set to 0x4, the L2 cache size would be: 512 sets/way x 64 bytes/line x 8 ways = 256 KBytes Conversely, if this field is set to 0x9, the SL field is set to 0x4, and the SA field is set to 0x4, the L2 cache size would be: 32768 sets/way x 32 bytes/line x 8 ways = 8 MBytes Note that the setting for 32768 sets/way cannot be used with the 64-byte line size because the proAptiv core does not support a 16 MB L2 cache size.	R	Preset
SL	7:4	L2 data cache line size. In the proAptiv core, the L2 cache line size can be configured at 32 or 64 bytes. This field is written by hardware at reset based on the state of the *L2_LineSize[3:0]* signals. These signals are driven based on the customer's line size choice during IP configuration. As such, this field is encoded as follows: 0x0 - 0x1: Reserved 0x2: 32 byte line size 0x3: 64 byte line size 0x4 - 0xF: Reserved	R	Preset per GUI
SA	3:0	L2 cache associativity. In the proAptiv core, the L2 cache associativity is fixed at 8 ways. This field is written by hardware at reset based on the state of the *L2_Assoc[3:0]* signals. As such, this field is encoded as follows: 0x0 - 0x6: Reserved 0x7: 8 ways 0x8 - 0xF: Reserved	R	0x7

2.3.1.4 Device Configuration 3 — Config3 (CP0 Register 16, Select 3)

Config3 provides information about the presence of optional extensions to the base MIPS32 architecture in addition to those specified in *Config2*. All fields in the *Config3* register are read-only.

Figure 2.5 Config3 Register Format — Multi-Core

31	30	29	28 26	25	24 16
M	0	CMGCR	0	SC	0

15 14	13	12	11	10	9	8 7	6	5	4	3	2	1	0
ISA	ULRI	RXI	DSP2P	DSPP	CTXTC	0	VEIC	VInt	SP	CDMM	MT	SM	TL

Table 2.7 Field Descriptions for Config3 Register

Name	Bit(s)	Description	Read/ Write	Reset State
M	31	Configuration continuation bit. This bit is always one to indicate the presence of *Config4*.	R	1
0	30	Must be written as zeros; returns zeros on read	0	0
CMGCR	29	Reads 1 to indicate that the Coherence Manager has a Global Configuration Register Space and the CMGCRBase cop0 register is implemented.	R	1
0	28:26	Must be written as zeros; returns zero on read.	R	0
SC	25	Segment Control implemented. This bit indicates whether the Segment Control registers *SegCtl0*, *SegCtl1* and *SegCtl2* are present. This bit is always 1 in the proAptiv core.	R	1
0	24:16	Must be written as zero; returns zero on read.	R	0
ISA	15:14	Indicates the instruction set availability. This bit is always 0 to indicate MIPS32.	R	0
ULRI	13	Reads 1 to indicate that the *UserLocal* Register is implemented.	R	1
RXI	12	Reads 1 to indicate that the *RIE* and *XIE* fields exist in the *PageGrain* register.	R	1
DSP2P	11	Reads 1 to indicate that Revision 2 of the MIPS DSP ASE is implemented	R	1
DSPP	10	Reads 1 to indicate that the MIPS DSP ASE extension is implemented.	R	1
CTXTC	9	Reads 1 to indicate the *ContextConfig* register is implemented. The width of the *BadVPN2* field in the *Context* register depends on the contents of the *ContextConfig* register.	R	1
0	8:7	Must be written as zero; returns zero on read.	R	0
VEIC	6	Support for an external interrupt controller. This bit is set or cleared by hardware depending on whether the EIC option was selected at build time. 0: Support for EIC mode not supported. 1: Support of EIC mode supported. The value of this bit is set by the static input, *SI_EICPresent*. This allows external logic to communicate whether an external interrupt controller is attached to the processor or not	R	Externally Set

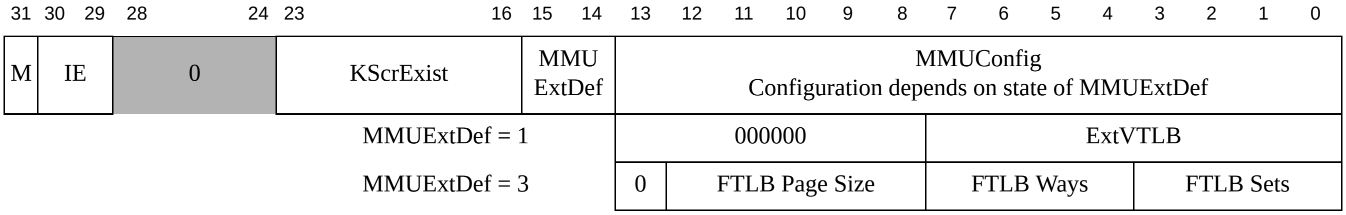
Table 2.7 Field Descriptions for Config3 Register *(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
VInt	5	Vectored interrupts implemented. This bit indicates whether vectored interrupts are implemented.. On the proAptiv core, this bit reads 1 to indicate the CPU can handle vectored interrupts.	R	1
SP	4	Reads 0 to indicate the CPU does not support 1 Kbyte TLB pages.	R	0
CDMM	3	Reads 1 to indicate the Common Device Memory Map (CDMM) feature is implemented, as well as the *CDMMBase* register is present.	R	1
MT	2	Reads 0 to indicate the proAptiv core does not include the MIPS MT module.	R	0
SM	1	Reads 0 to indicate the CPU does not include the instructions of the SmartMIPS ASE.	R	0
TL	0	Reads 1 to indicate PDTrace is supported.	R	1

2.3.1.5 Device Configuration 4 — Config4 (CP0 Register 16, Select 4)

The *Config4* register encodes additional capabilities such as the number of page-pair entries within the FTLB.

Figure 2.6 Config4 Register Format



All other MMUExtDef values reserved.

Table 2.8 Field Descriptions for Config4 Register

Name	Bit(s)	Description	Read/ Write	Reset State
M	31	Configuration continuation bit. This bit is one to indicate the presence of *Config5*.	R	1
IE	30:29	TLBINV instruction support. For this field, the proAptiv core only returns the following encoding. 10: TLBINV, TLBINVF instruction supported, EntryHi$_{EHINV}$ supported. TLBINV, TLBINVF instruction operate on one TLB entry.	R	2
0	28:24	Reserved. Must be written as zero. Ignored on reads.	R	0

Table 2.8 Field Descriptions for Config4 Register *(continued)*

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Description</th><th>Read/ Write</th><th>Reset State</th></tr>
<tr><td><i>KScrExist</i></td><td>23:16</td><td>Indicates how many scratch registers are available to kernel-mode software within CP0 Register 31. In the proAptiv architecture, three kernal scratch registers are included at register selects 2, 3, and 4.
Each bit represents a select for CP0 Register 31. Bit 16 represents Select 0, Bit 23 represents Select 7. If the bit is set, the associated scratch register is implemented and available for kernel-mode software. Therefore, this field contains a value of 0x1C (8’b00011100). This indicates that bits 18 - 20 are set, corresponding to selects 2, 3, and 4.
These registers are used by the kernel for temporary storage of information. Refer to Section 2.3.12, "Kernel Mode Support Registers" on page 174 for more information.</td><td>R</td><td>0x1C</td></tr>
<tr><td><i>MMUExtDef</i></td><td>15:14</td><td>MMU Extension Definition. This 2-bit field defines how Config4[12:0] is to be interpreted. Refer to Figure 2.6 for more information. This field is encoded as follows:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>00</td><td>Reserved</td></tr>
<tr><td>01</td><td>Config4[7:0] used as MMUSizeExt.</td></tr>
<tr><td>10</td><td>Reserved</td></tr>
<tr><td>11</td><td>Config4[3:0] indicates FTLB ways.
Config4[7:4] indicates FTLB sets.
Config4[12:8] indicates FTLB page size.</td></tr>
</table></td><td>R</td><td>1</td></tr>
<tr><td colspan="5"><b><i>Assignment of bits 13:0 when MMUExtDef = 2’b11</i></b></td></tr>
<tr><td>0</td><td>13</td><td>Reserved. Must be written as zero. Ignored on reads.</td><td>R</td><td>0</td></tr>
<tr><td><i>FTLB Page Size</i></td><td>12:8</td><td>Indicates the Page Size of the FTLB Array Entries. The FTLB must be flushed of any valid entries before this register field value is changed by software. The FTLB behavior is UNDEFINED if there are valid FTLB entries which were not all programmed using a common page size.
This field is encoded as follows:
<table>
<tr><th>Encoding</th><th>Page Size</th></tr>
<tr><td>00000</td><td>Reserved</td></tr>
<tr><td>00001</td><td>4 KB</td></tr>
<tr><td>00010</td><td>16 KB</td></tr>
<tr><td>00011 - 11111</td><td>Reserved</td></tr>
</table>
Note that the <i>MMUExtDef</i> field must contain have a value of 2’b11 for this field to have meaning. If the <i>MMUExtDef</i> field is 2’b01, this field is ignored. This bit is set not used during reset as the FLTB is disabled by default. If the FTLB is not enabled, this field is ignored.</td><td>R/W</td><td>0x01</td></tr>
</table>

Table 2.8 Field Descriptions for Config4 Register *(continued)*

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Description</th><th>Read/ Write</th><th>Reset State</th></tr>
<tr><td><i>FTLB Ways</i></td><td>7:4</td><td>Indicates the set associativity of the FTLB array, which is fixed at 4 in the proAptiv architecture. This field is encoded as follows:
<table>
<tr><th>Encoding</th><th>Associativity</th></tr>
<tr><td>0000 - 0001</td><td>Reserved</td></tr>
<tr><td>0010</td><td>4</td></tr>
<tr><td>0011 - 1111</td><td>Reserved</td></tr>
</table>
This field is not used during reset as the FLTB is disabled by default. If the FTLB is not enabled, this field is used as part the MMUSizeExt field. For more information, refer to Section 5.3.3.2 of the MMU chapter.</td><td>R</td><td>0x2</td></tr>
<tr><td><i>FTLB Sets</i></td><td>3:0</td><td>Indicates the number of sets per way within the FTLB array, which is fixed at 128 in the proAptiv architecture. This field is encoded as follows:
<table>
<tr><th>Encoding</th><th>Sets per Way</th></tr>
<tr><td>0000 - 0110</td><td>Reserved</td></tr>
<tr><td>0111</td><td>128</td></tr>
<tr><td>1000 - 1111</td><td>Reserved</td></tr>
</table>
This field is not used during reset as the FLTB is disabled by default. If the FTLB is not enabled, this field is used as part the MMUSizeExt field. For more information, refer to Section 5.3.3.2 of the MMU chapter.</td><td>R</td><td>0x7</td></tr>
<tr><td colspan="5"><b><i>Assignment of bits 13:0 when MMUExtDef = 2’b01</i></b></td></tr>
<tr><td>0</td><td>13:8</td><td>Reserved. Must be written as zero. Ignored on reads.</td><td>R</td><td>0</td></tr>
<tr><td><i>ExtVTLB</i></td><td>7:0</td><td>If the FTLB is either not present or disabled, bits 7:0 of the <i>Config4</i> register are used to extend the number of VTLB entries indicated by the 6-bit <i>Config4</i>$_{MMUSize}$ field. This extends the number of bits used to determine the number of VTLB entries from 6 to 14.
This field can be used to extend the number of VTLB entries in future generations of MIPS processors. Since there are a maximum of 64 VLTB entries in the proAptiv core, bits 7:0 of the <i>Config4</i> register always contain a value of 0 when only the VTLB is present.</td><td>R</td><td>0x00</td></tr>
</table>

2.3.1.6 Device Configuration 5 — Config5 (CP0 Register 16, Select 5)

The *Config5* register encodes additional capabilities for the address mode programming and cache error exceptions.

Figure 2.7 Config5 Register Format

31	30	29	28	27 ... 0
M	K	CV	EVA	0

Table 2.9 Field Descriptions for Config5 Register

Name	Bit(s)	Description	Read/ Write	Reset State
M	31	Configuration continuation bit. Even though the *Config6 and Config7* registers are used in the proAptiv Multiprocessing System, they are both defined as implementation-specific registers. As such, this bit is zero and is not used to indicate the presence of *Config6*.	R	0
K	30	This bit effects the cache coherency attributes, the boot exception vector overlay, and the location of the exception vector as follows: When this bit is cleared, the following events occur: 1. The *Config*$_{K0}$ field is used to set the cache coherency attributes for the kseg0 region (0x8000_0000 - 0x9FFF_FFFF). 2. Hardware creates two boot overlay segments, one for kseg0 and one for kseg1. 3. The exception vectors are forced to reside in kseg0/kseg1 by ignoring the state of bits 31:30 of the *EBase* register as well as the *SI_ExceptionBase[31:30]* pins and forcing them to a value of 2’b10. When this bit is set, the following events occur: 1: The *Config*$_{K0}$ field is ignored and the cache coherency attributes are derived from the C fields of the various segmentation control registers (*SegCtl0 - SegCtl2*). 2. Hardware creates one boot overlay segment that can reside anywhere in virtual address space. 3. The exception vectors are not forced to reside in kseg0/kseg1. Rather, bits 31:30 of the *EBase* register, as well as the *SI_ExceptionBase[31:30]* signals and used to place the exception vectors anywhere within virtual address space.	R/W	0
CV	29	Cache error exception vector control. Disables logic forcing use of kseg1 region in the event of a Cache Error exception when *Status*$_{BEV}$ = 0. When the CV bit is cleared, bits 31:30 of the *EBase* Register are fixed with the value 2’b10 to force the exception base address to be in the kseg0 or kseg1 unmapped virtual address segments. Bit 29 of exception base address will be forced to 1 on Cache Error exceptions so the exception handler will be executed from the uncached kseg1 segment. When the CV bit is set, the ExcBase field is expanded to include bits 31:30 to facilitate programmable memory segmentation.	R/W	0
EVA	28	This bit is always a logic one to indicate support for enhanced virtual address (EVA).	R	1
Reserved	27:0	Reserved. Must be written as zero. Ignored on reads.	R	0

2.3.1.7 Device Configuration 6 — Config6 (CP0 Register 16, Select 6)

Config6 provides information about the presence of optional extensions to the base MIPS32 architecture.

Figure 2.8 Config6 Register Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	FPDSR	DOPC	MNAN	DSFW	DWP	DL1B	DNPE	ODTG	ODDG	DLSB	DFIS	HITLB	HDTLB	EFTLB	

15	14	13	12	10	9	1	0
EFTLB	SPCD	0	PERFSEL[2:0]		0		JDRC

Table 2.10 Field Descriptions for Config6 Register

Name	Bit(s)	Description	Read/ Write	Reset State
0	31:30	Reserved. Write as zero. Ignored on reads.	R	0
FPDSR	30	Floating point disable square root. 0: Enable floating point divide and square root 1: Disable floating point divide and square root	R/W	0
DOPC	29	Opcode cache disable. Setting this bit indicates that the opcode cache is disabled. 0: Opcode cache is enabled. 1: Opcode cache is disabled.	R/W	0
MNAN	28	MIPS NaN compliance. 0: Default NaN compliance. 1: Legacy MIPS NaN compliance.	R/W	0
DSFW	27	Disable superforwarding. 0: Enable superforwarding. 1: Disable superforwarding.	R/W	0
DWP	26	Disable IFU way prediction. 0: Enable IFU way prediction. 1: Disable IFU way prediction.	R/W	0
DL1B	25	Disable L1 branch target buffer. 0: Enable L1 branch target buffer. 1: Disable L1 branch target buffer.	R/W	0
DNPE	24	Disable NOP elimination. 0: Enable NOP elimination. 1: Disable NOP elimination.	R/W	0
ODTG	23	Override data cache tag clock gater. 0: Enable data cache tag clock gating. 1: Override data cache tag clock gating. Enable the clock to data cache tag array always.	R/W	0
ODDG	22	Override data cache data clock gater. 0: Enable data cache data clock gating. 1: Override data cache data clock gating. Enable the clock to data cache data array always.	R/W	0
DLSB	21	Disable load/store bonding. 0: Enable load/store bonding. 1: Disable load/store bonding.	R/W	0

Table 2.10 Field Descriptions for Config6 Register*(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
DFIS	20	Disable ‘cracking’. 0: Enable cracking. 1: Disable cracking.	R/W	0
HITLB	19	Half size instruction TLB (ITLB). When this bit is set, the ITLB becomes half of its current size. 0: Full size ITLB. 1: Half size ITLB.	R/W	0
HDTLB	18	Half size data TLB (DTLB). When this bit is set, the DTLB becomes half of its current size. 0: Full size DTLB. 1: Half size DTLB.	R/W	0
FTLBP	17:16	FTLB probability. On a TLBWR instruction, if the *PageMask* register matches the FTLB page size, the write would be done to the FTLB. Otherwise it would go to the FTLB. However, for systems that use only a single page size, the FTLB would be used and most of the FTLB would be unused. This field allows some TLBWR instruction to go to the VTLB instead of the FTLB whenever the *PageMask* register matches the FTLB page size. If the contents of the *PageMask* register do not match the FTLB page size, the TLBWR instruction goes to the VTLB. 0: FTLB only. All TLBWR instructions go to the FTLB. 1: FTLB:VTLB = 15:1. For every 16 TLBWR instructions, 15 go to the FTLB and 1 goes to the VTLB. 2: FTLB:VTLB = 7:1. For every 8 TLBWR instructions, 7 go to the FTLB and 1 goes to the VTLB. 3: FTLB:VTLB = 3:1. For every 4 TLBWR instructions, 3 go to the FTLB and 1 goes to the VTLB.	R/W	0
FTLBEn	15	FTLB enable. Setting this bit indicates that the FTLB is enabled. 0: FTLB is disabled. 1: FTLB is enabled.	R/W	0
SPCD	14	Sleep state performance counter disable. When this bit is set, the performance counter proAptiv clocks are prevented from shutting down. The primary use of this bit is to keep performance counters alive when the proAptiv core is in sleep mode. 0: Performance counters are enabled in sleep mode. 1: Performance counters are disabled in sleep mode.	R/W	0
0	13	Reserved. Write as zero. Ignored on reads.	R	0

Table 2.10 Field Descriptions for Config6 Register*(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
IFUPerfCtl	12:10	IFU Performance Control. This field encodes IFU events that provide debug and performance information for the IFU pipeline and is encoded as follows: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>000</td><td>IDU is accepting instructions, but IFU is not providing any.</td></tr><tr><td>001</td><td>A control transfer instruction such as a branch or jump causes lost IDU bandwidth.</td></tr><tr><td>010</td><td>A stalled instruction such as an unpredicted jump must wait for an address and thus causes lost IDU bandwidth.</td></tr><tr><td>011</td><td>Cache prediction was correct.</td></tr><tr><td>100</td><td>Cache prediction was incorrect.</td></tr><tr><td>101</td><td>Cache did not predict due to invalid JR cache entry, or the instruction tag miscompared with tag in JR cache.</td></tr><tr><td>110</td><td>Unimplemented.</td></tr><tr><td>111</td><td>Condition branch was taken.</td></tr></table> Lost IDU bandwidth occurs when the IDU is accepting instructions, but instructions are not being provided by the IFU. The count of these events can be seen via Performance Counters 0 or 3, and the event number 11. In order to view the *IFU Perf Ctl* events, the Performance Counter Control needs to be programmed accordingly See Table 2.63, "Performance Counter Events and Codes" for general information on event number 11.	R/W	0
0	9:1	Reserved. Write as zero. Ignored on reads.	R	0
JRCD	0	Jump register cache prediction disable. Setting this bit disables the Jump Register (JR) target address prediction. 0: JR cache target address prediction is enabled. 1: JR cache target address prediction is not enabled.	R/W	0

2.3.1.8 Device Configuration 7 — Config7 (CP0 Register 16, Select 7)

This register controls machine-specific features of the proAptiv core. A few of them are for hardware interface adaptation, but most are for chip or system test only. They default to a "safe" value. Most software, including bootstrap software, can and should ignore these registers unless specifically advised to use them.

Figure 2.9 Config7 Register Format

31	30	29	28	27	26 25 24	23	22	21	20 19	18	17	16
WII	FPFS	IHB	0	SEHB	0	DGHR	SG	SUI	0	HCI	0	AR

15 13	12 11	10	9	8	7	6	5	4	3	2	1	0
0	PREF	IAR	IVAD	ES	0	CP1IO	0	ULB	BP	RPS	BHT	SL

Table 2.11 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 proAptiv core remains in the wait condition forever if entered with interrupts disabled. If set to 1, it allows operating system code to avoid complex race conditions.	R	1
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	Implicit hazard barrier. If *IHB* = 1, the following behavior will be true: • When the proAptiv 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 CPU 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, but the proAptiv core cannot guarantee such behavior with a small number of **`nops/ssnops`**. Per Release3, 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
0	28	Reserved. Write as zero. Ignored on reads.	R	0
SEHB	27	Slow EHB. An experimental mode to accelerate CP0 sequences using the **`ehb`** instruction. 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
0	26:24	Reserved for future use.	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 2.11 Field Descriptions for Config7 Register*(continued)*

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Description</th><th>Read/ Write</th><th>Reset State</th></tr>
<tr><td><i>SUI</i></td><td>21</td><td>Strict Uncached Instruction (SUI) policy control.
When this bit is set, hardware runs uncached instructions strictly in order and (as far as possible) unpipelined. This will cause a significant performance degradation as it will introduce a bubble equivalent to the depth of the pipeline between each instruction. 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.</td><td>R/W</td><td>0</td></tr>
<tr><td>0</td><td>20:19</td><td>Reserved. Write as zero. Ignored on reads.</td><td>R</td><td>0</td></tr>
<tr><td>HCI</td><td>18</td><td>Hardware Cache Initialization: Indicates that a cache does not require initialization by software. This bit will most likely only be set on simulation-only cache models and not on real hardware.</td><td>R</td><td>Based on HW present</td></tr>
<tr><td>0</td><td>17</td><td>Reserved. Write as zero. Ignored on reads.</td><td>R</td><td>0</td></tr>
<tr><td><i>AR</i></td><td>16</td><td>Alias removed. Hardware sets this bit to indicate that the L1 data cache is configured to avoid cache aliases. The data cache virtual aliasing hardware is always present in the proAptiv core.</td><td>R</td><td>Preset at build time</td></tr>
<tr><td>0</td><td>15:13</td><td>Reserved. Write as zero. Ignored on reads.</td><td>R</td><td>0</td></tr>
<tr><td><i>PREF</i></td><td>12:11</td><td>These two bits control the extent of prefetching of instructions into the instruction cache as indicated. This field is encoded as follows:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>00</td><td>Prefetch 0 cache lines on an I-cache miss in addition to fetching the missing cache line. i.e. Disable I-cache prefetching.</td></tr>
<tr><td>01</td><td>Prefetch 1 cache line (sequential next line) on an I-cache miss in addition to fetching the missing cache line.</td></tr>
<tr><td>10</td><td>Reserved.</td></tr>
<tr><td>11</td><td>Prefetch 2 cache lines (sequential next 2 lines) on an I-cache miss in addition to fetching the missing cache line.</td></tr>
</table></td><td>R/W</td><td>01</td></tr>
<tr><td><i>IAR</i></td><td>10</td><td>Instruction Alias Removed.
Indicates that the proAptiv core has hardware support to remove instruction cache aliasing. This hardware is only present when the proAptiv core is configured with a TLB and cache size of 32KB or larger. The virtual aliasing hardware can be disabled via the <i>IVAD</i> bit described below. The instruction cache virtual aliasing hardware is always present in the proAptiv core.</td><td>R</td><td>1</td></tr>
</table>

Table 2.11 Field Descriptions for Config7 Register*(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
IVAD	9	Instruction Virtual Aliasing disabled. The hardware required to resolve instruction cache virtual alising is always present in the proAptiv core as noted by the defualt state of the *IAR* bit shown above. However, software can toggle the *IVAD* bit to enable or disable the virtual aliasing hardware for the instruction cache. Setting this bit disables the hardware alias removal on the instruction cache. If this bit is cleared, the **`CACHE Hit Invalidate`** and **`SYNCI`** instructions look up all possible aliased locations and invalidate the given cache line in all of them. This bit is Read-only if *IAR* = 0.	R/W	0 (hardware aliasing enabled)
ES	8	Externalize **`sync`**. If this bit is set, and if the downstream device (toward memory) is capable of accepting SYNCs (indicated by the pin *SI_SyncTxEn*), the **`sync`** instruction causes a SYNC-specific transaction to go out on the external bus. If this bit is cleared or if *SI_SyncTxEn* is deasserted, no transaction will go out, but all SYNC handling internal to the CPU will nevertheless be performed. The **`sync`** instruction is signalled on the proAptiv'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 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. When this bit is read, the value returned depends on the state of the *SI_SyncTxEn* pin. If *SI_SyncTxEn* is 0, a value of 0 is returned. If *SI_SyncTxEn* is 1, the value returned is the last value that was written to this bit.	R	1
0	7	Reserved. Write as zero. Ignored on reads.	R	0
CP1IO	6	CP1 instruction order. By default, data sent from the proAptiv 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
0	5	Reserved. Write as zero. Ignored on reads.	R	0
ULB	4	Uncached load blocking. Set to 1 to make all uncached loads blocking (a program usually only blocks when it uses the data which is loaded).	R/W	0
BP	3	Branch prediction. When set, no branch prediction is done, and all branches and jumps stall as above.	R/W	0
RPS	2	Return prediction stack. When set, the return address branch predictor is disabled, so **`jr $31`** is treated just like any other jump register. An instruction fetch stalls after the branch delay slot, until the jump instruction reaches the Address Generation pipeline and can provide the right address.	R/W	0
BHT	1	Branch history table. 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
SL	0	Scheduled loads. When set, non-blocking loads are disabled. Normally the proAptiv core continues to after a load instruction, even if it misses in the D-cache, until the data is used. When this bit is set, the CPU stalls on any D-cache load miss.	R/W	0

2.3.1.9 Processor ID — PRId (CP0 Register 15, Select 0)

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

Figure 2.10 PRId Register Format

31 ... 24	23 ... 16	15 ... 8	7 ... 0
CoOpt	CoID	ProcType	Rev

Table 2.12 Field Descriptions for PRId Register

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Description</th><th>Read/ Write</th><th>Reset State</th></tr>
<tr><td><i>CoOpt</i></td><td>31:24</td><td>Company Option. Should be a number between 0 and 127— higher values are reserved by MIPS Technologies.</td><td>R</td><td>Preset</td></tr>
<tr><td><i>CoID</i></td><td>23:16</td><td>Company ID. Identifies the company that designed or manufactured the processor. In the proAptiv Multiprocessing System, this field contains a value of 1 to indicate MIPS Technologies, Inc.</td><td>R</td><td>1</td></tr>
<tr><td><i>ProcType</i></td><td>15:8</td><td>Processor ID. Identifies the type of processor. This field allows software to distinguish between the various types of processors from MIPS Technologies. The value of this field is 0xA3 for the proAptiv core.</td><td>R</td><td>A3</td></tr>
<tr><td><i>Rev</i></td><td>7:0</td><td>The revision number of the proAptiv design. This field allows software to distinguish between one revision and another of the same processor type.
This field is broken up into the following three subfields:
<table>
<tr><th>Bit(s)</th><th>Name</th><th>Meaning</th></tr>
<tr><td>7:5</td><td>Major Revision</td><td>This number is increased on major revisions of the proAptiv core.</td></tr>
<tr><td>4:2</td><td>Minor Revision</td><td>This number is increased on each incremental revision of the processor and reset on each new major revision.</td></tr>
<tr><td>1:0</td><td>Patch Level</td><td>If a patch is made to modify an older revision of the processor, this field will be incremented.</td></tr>
</table></td><td>R</td><td>Preset</td></tr>
</table>

2.3.1.10 Exception Base Address — EBase (CP0 Register 15, Select 1)

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

The *EBase* register provides the ability for software to identify the specific processor within a multi-processor system, and allows the exception vectors for each processor to be different. Bits 31:12 of the *EBase* register are concatenated with zeros to form the base of the exception vectors when *Status*$_{BEV}$ is 0. The exception vector base address comes from the fixed defaults when *Status*$_{BEV}$ is 1, or for any EJTAG Debug exception. The reset state of bits 31:12 of the *EBase* register initialize the exception base register to `0x8000.0000`, providing backward compatibility with Release 1 implementations.

The size of the ExcBase field depends on the state of the WG bit. At reset, the WG bit is cleared by default. In this case, the ExcBase field is comprised of bits 29:12. Bits 31:30 of the EBase Register are not writeable and are forced to a value of 2'b10 by hardware so that the exception handler will be executed from the *kseg0/kseg1* segments. This is shown in Figure 2.11.

When the WG bit is set, bits 31:30 of the ExcBase field become writeable and are used to relocate the ExcBase field to other segments after they have been setup using the *SegCtl0* through *SegCtl2* registers. This is shown in Figure 2.12. Note that if the WG bit is set by software (allowing bits 31:30 to become part of the ExcBase field) and then cleared, bits 31:30 can no longer be written by software and the state of these bits remains unchanged for any writes after WG was cleared. Therefore, it is the responsibility of software to write a value of 2'b10 to bits 31:30 of the EBase register prior to clearing the WG bit if it wants to ensure that future exceptions will be executed from the kseg0 or kseg1 segments.

Note that the WG bit is different from the CV bit in the SegCtl0 register located in Section 2.3.3.1, "Segmentation Control 0 — SegCtl0 (CP0 Register 5, Select 2)". Although their functions are similar, the CV bit applies only to cache error exceptions, whereas the WG bit applies to all exceptions.

If the value of the exception base register is to be changed, this must be done with *StatusBEV* equal to 1. The operation of the processor is **UNDEFINED** if the exception base field is written with a different value when *StatusBEV* is 0.

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

Figure 2.11 EBase Register Format — WG = 0

31	30	29 12	11	10	9 0
1	0	ExcBase	WG	0	CPUNum

Figure 2.12 EBase Register Format — WG = 1

31 12	11	10	9 0
ExcBase	WG	0	CPUNum

Table 2.13 Field Descriptions for EBase Register

Name	Bit(s)	Description	Read/ Write	Reset State
ExcBase	31:12	Exception Base Address. The size and behavior of this field depends on the state of the WG bit. When the WG bit is set, the ExcBase field includes bits 31:30 to facilitate programmable memory segmentation. This field specifies the base address of the exception vectors when $\text{Status}_{\text{BEV}}$ is zero. Bits 31:30 can be written only when WG is set. When WG is zero, these bits are unchanged on a write. When the WG bit is cleared, bits 31:30 of this field must be 2'b`10` to make sure the exception vector maps to kseg0 or kseg1, conventionally used for OS code. In a multi-core environment, setting *EBase* in any CPU to a unique value allows that CPU can have its own unique exception handlers. This field should be written only when $\textit{Status}_{\textit{BEV}}$ is set so that any exception will be handled through the ROM entry points.	R/W	0x8000.0
WG	11	Write gate. When the WG bit is set, the ExcBase field is expanded to include bits 31:30 of the *EBase* register to facilitate programmable memory segmentation controlled by the *SegCtl0* through *SegCtl2* registers. When the WG bit is cleared, bits 31:30 of the *EBase* register are not writeable and remain unchanged from the last time that WG was cleared.	R/W	Externally Set

Table 2.13 Field Descriptions for EBase Register

Name	Bit(s)	Description	Read/ Write	Reset State
0	10	Reserved. Write as zero. Ignored on reads.	R	0
CPUNum	9:0	This field contains an identifier that will be unique among the CPU’s in a multi-processor system. The value in this field is set by the *SI_CPUNum[9:0]* static input pins to the proAptiv core.	R	Externally Set

2.3.1.11 Status (CP0 Register 12, Select 0)

The *Status* register is a read/write register that contains the operating mode, interrupt enabling, and diagnostic states of the processor. Fields in this register and the CP0 Debug register combine to create operating modes for the processor. Selected bits are encoded as follows to place the processor into one of the operating modes. Refer to the MMU chapter for more information on the various operating modes. A brief summary is provided below.

Table 2.14 Operating Mode Encoding

Status$_{IE}$	Status$_{ERL}$	Status$_{EXL}$	Status$_{KSU}$	Debug$_{DM}$	Mode of Operation
1	0	0	x	0	Individual interrupts can be disabled/enabled using the *Status$_{IM7-0}$* mask bits.
x	0	0	0x2	0	User Mode. In user mode, the CPU has access only to the mapped kuseg address region.
x	0	0	0x1	0	Supervisor Mode. In supervisor mode, the CPU has access to the top half of the kseg2 region (sometimes known as kseg3), but no access to CP0 registers or most kernel memory.
x	1	1	0	0	Kernel Mode. 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 is unable to access some debug resources.
x	x	x	x	1	Debug Mode. In debug mode, the processor has full access to all resources that are available in Kernel Mode operation, in addition to those provided by EJTAG.

Figure 2.13 shows the format of the *Status* Register; Table 2.15 describes the *Status* register fields.

Figure 2.13 Status Register Format

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

Table 2.15 Field Descriptions for Status Register

Name	Bit(s)	Description	Read/ Write	Reset State
CU3	31	Coprocessor 3 Usable. Because the proAptiv core does not support a coprocessor 3, *Status*$_{CU3}$ is hardwired to zero.	R	0
CU2	30	Coprocessor 2 Usable. Controls access to coprocessor 2. 0: Access not allowed. 1: Access allowed. *CU2* is reserved for a customer's coprocessor. Currently the proAptiv Multiprocessing System family of cores does not support Coprocessor 2, so this bit is read-only and reads zero.	R	0
CU1	29	Coprocessor 1 Usable. Controls access to coprocessor 1. 0: Access not allowed. 1: Access allowed. *CU1* is most often used for a floating-point unit. When no coprocessor 1 is present, this bit is read-only and reads zero.	R/W	Undefined
CU0	28	Coprocessor 0 accessible in User Mode. This bit controls user mode access to coprocessor 0. 0: Access not allowed. 1: Access allowed. Coprocessor 0 is always usable when the processor is running in Kernel or Debug Mode, regardless of the state of the *CU0* bit. Setting *Status*$_{CU0}$ to 1 has the effect of allowing privileged instructions to execute in user mode, although this is not something a secure OS is likely to allow.	R/W	Undefined
RP	27	Reduced Power. Enable/disable reduced power mode. 0: Disable reduced power mode. 1: Enable reduced power mode. The state of the *RP* bit is visible on the core's external interface signal *SI_RP*.	R/W	0

Table 2.15 Field Descriptions for Status Register *(continued)*

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Description</th><th>Read/ Write</th><th>Reset State</th></tr>
<tr><td><i>FR</i></td><td>26</td><td>Floating Register. This bit is used to control the floating-point register mode for 64-bit floating point units:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Floating point registers can contain any 32-bit data type. 64-bit data types are stored in even-odd pairs of registers.</td></tr>
<tr><td>1</td><td>Floating point registers can contain any datatype</td></tr>
</table>
This bit must be ignored on writes and read as zero under the following conditions
• No floating point unit is implemented
• 64-bit floating point unit is not implemented
If the proAptiv core is equipped with an optional FPU, set this bit to 0 for MIPS I compatibility mode, which allows for 16 real FP registers, with 16 odd FP register numbers reserved for access to the high-order bits of double-precision values. .</td><td>R/W</td><td>0</td></tr>
<tr><td><i>RE</i></td><td>25</td><td>Reverse Endian. Enables Reverse endianness for instructions that execute in User mode. This bit is always 0 as this feature is not supported in the proAptiv Multiprocessing System.</td><td>R</td><td>0</td></tr>
<tr><td><i>MX</i></td><td>24</td><td>MIPS DSP Extension. Enables access to DSP ASE resources.
0: Access not allowed.
1: Access allowed.
An attempt to execute any DSP ASE instruction before when this bit is 0 will cause a <i>DSP State Disabled</i> exception. The state of this bit is reflected in <i>Config3_{DSPP}</i> .</td><td>R/W</td><td>0</td></tr>
<tr><td>0</td><td>23</td><td>Reserved. Write as zero. Ignored on reads.</td><td>R</td><td>0</td></tr>
<tr><td><i>BEV</i></td><td>22</td><td>Boot Exception Vector. Controls the location of exception vectors:
0: Normal. Refer to the EBase register for more information.
1: Bootstrap
When set to 1, all exception entry points are relocated to near the reset start address.</td><td>R/W</td><td>1</td></tr>
<tr><td><i>TS</i></td><td>21</td><td>TLB Shutdown. This bit is set by hardware if software attempts to create a duplicate TLB entry (which will also produce a "machine check" exception). It can be cleared to zero by software, but can never set to 1.
The name of the field originated as a "TLB Shutdown". Historically, MIPS CPUs stop translating addresses when they detected invalid TLB operations.</td><td>R/W0</td><td>0
(Set by hardware, Cleared by software)</td></tr>
<tr><td><i>SR</i></td><td>20</td><td>Soft Reset. The proAptiv core only supports a full external reset, so this bit is not used and always reads zero.</td><td>R</td><td>0</td></tr>
<tr><td><i>NMI</i></td><td>19</td><td>Indicates that the entry through the reset exception vector was due to an NMI.
0: Not NMI (reset)
1: NMI
Software can only write a 0 to this bit to clear it and cannot force a 0 to 1 transition.</td><td>R/W0</td><td>1 for NMI
0 otherwise</td></tr>
</table>

Table 2.15 Field Descriptions for Status Register *(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
0	18	Reserved. Write as zero. Ignored on reads.	R	0
CEE	17	CorExtend Enable. Enable/disable CorExtend User Defined Instructions (UDIs). 0: Disable CorExtend block 1: Enable CorExtend block The presence of the CorExtend extension is indicated in *Config*$_{UDI}$, which is set when the core is configured. This bit is reserved if CorExtend is not present.	R/W	Undefined
0	16	Reserved. Write as zero. Ignored on reads.	R	0
IM7-0	15:8	Interrupt Mask. Bitwise interrupt enables for the eight interrupt conditions. The state of these bits is visible in *Cause*$_{IP7-0}$, except in EIC mode. External Interrupt Controller (EIC) mode is activated when the *Config3*$_{VEIC}$ is set by hardware at reset based on the state of the *SI_EICPresent* signal. If this bit is set by hardware, software should set the *Cause*$_{IV}$ bit, then write a non-zero "vector spacing" in the VS bit of the *IntCtl* register. In EIC mode, *IM7-2* is used as a 6-bit *Status*$_{IPL}$ (Interrupt Priority Level) field. An interrupt is only triggered when the interrupt controller presents an interrupt code which is numerically higher than the current value of *Status*$_{IPL}$. *Status*$_{IM1-0}$ always acts as a bitwise mask for the two software interrupt bits programmable in *CauseI*$_{P1-0}$.	R/W	Undefined
0	7:5	Reserved. Write as zero. Ignored on reads.	R	0
KSU	4:3	These bits denote the processor’s operating mode. 2’b00: Kernel Mode 2’b01: Supervisor Mode 2’b10: User Mode. Note that the processor can also be in Kernel mode if *ERL* or *EXL* is set, regardless of the state of these bits.	R/W	Undefined
ERL	2	Error Level; Set by the processor when a Reset, NMI, or Cache Error exception is taken. 0: Normal level 1: Error level When *ERL* is set: • The processor is running in kernel mode • Interrupts are disabled • The ERET instruction will use the return address held in *ErrorEPC* instead of *EPC* • When ERL = 1 in the Status register, the segment kuseg (legacy) or xkseg0 (EVA) is treated as an 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.	R/W	1

Table 2.15 Field Descriptions for Status Register *(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
EXL	1	Exception Level; Set by the processor when any exception other than Reset, Cache Error, or NMI exception is taken. 0: Normal level 1: Exception level When EXL is set: • The processor is running in Kernel Mode. • Hardware and software interrupts are disabled. • TLB Refill exceptions use the general exception vector instead of the TLB Refill vector. When an exception occurs and *EXL* is set, a nested TLB Refill exception is sent to the general exception handler (rather than to its dedicated handler) and the values in *EPC*, $Cause_{BD}$ are not overwritten. The result is that, after returning from the second exception, the processor jumps back to the code that was executing before the first exception occurred.	R/W	Undefined
IE	0	Interrupt Enable. Acts as the master enable for software and hardware interrupts. 0: Interrupts are disabled 1: Interrupts are enabled This bit can be written using the `di`/`ei` instructions.	R/W	Undefined

2.3.1.12 Interrupt Control — IntCtl (CP0 Register 12, Select 1)

The *IntCtl* register controls the interrupt capabilities of the *proAptiv Multiprocessing System* core, including vectored interrupts and support for an external interrupt controller.

Figure 2.14 IntCtl Register Format

31 29	28 26	25 23	22 10	9 5	4 0
IPTI	IPPCI	IPFDCI	0	VS	0

Table 2.16 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. This field is encoded as shown in Table 2.17, "Encoding of IPTI, IPPCI, and IPFDCI Fields". 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.	R	Externally Set

Table 2.16 Field Descriptions for IntCtl Register

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Description</th><th>Read/
Write</th><th>Reset State</th></tr>
<tr><td><i>IPPCI</i></td><td>28:26</td><td>For <i>Interrupt Compatibility</i> and <i>Vectored Interrupt</i> 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. This field is encoded as shown in Table 2.17, "Encoding of IPTI, IPPCI, and IPFDCI Fields".
The value of this bit is set by the static input <i>SI_IPPCI[2:0]</i>. This allows external logic to communicate the specific <i>SI_Int</i> hardware interrupt pin to which the <i>SI_PCInt</i> 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.</td><td>R</td><td>Externally Set</td></tr>
<tr><td><i>IPFDCI</i></td><td>25:23</td><td>For <i>Interrupt Compatibility</i> and <i>Vectored Interrupt</i> modes, this field specifies the IP number to which the Fast Debug Channel Interrupt request is merged, and allows software to determine whether to consider $Cause_{FDCI}$ for a potential interrupt. This field is encoded as shown in Table 2.17, "Encoding of IPTI, IPPCI, and IPFDCI Fields".
The value of this bit is set by the static input, <i>SI_IPFDCI[2:0]</i>. This allows external logic to communicate the specific <i>SI_Int</i> hardware interrupt pin to which the <i>SI_FDCInt</i> 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.</td><td>R</td><td>Externally Set</td></tr>
<tr><td>0</td><td>22:10</td><td>Reserved. Write as zero. Ignored on reads.</td><td>R</td><td>0</td></tr>
<tr><td><i>VS</i></td><td>9:5</td><td>Vector Spacing. If vectored interrupts are implemented (as denoted by $Config3_{VInt}$ or $Config3_{VEIC}$), this field specifies the spacing between vectored interrupts.
<table>
<tr><th>VS Field Encoding</th><th>Spacing Between Vectors (hex)</th><th>Spacing Between Vectors (decimal)</th></tr>
<tr><td>0x00</td><td>0x000</td><td>0</td></tr>
<tr><td>0x01</td><td>0x020</td><td>32</td></tr>
<tr><td>0x02</td><td>0x040</td><td>64</td></tr>
<tr><td>0x04</td><td>0x080</td><td>128</td></tr>
<tr><td>0x08</td><td>0x100</td><td>256</td></tr>
<tr><td>0x10</td><td>0x200</td><td>512</td></tr>
</table>
All other values are reserved. The operation of the processor is <b>UNDEFINED</b> if a reserved value is written to this field.</td><td>R/W</td><td>0</td></tr>
<tr><td>0</td><td>4:0</td><td>Reserved. Write as zero. Ignored on reads.</td><td>R</td><td>0</td></tr>
</table>

Table 2.17 Encoding of IPTI, IPPCI, and IPFDCI Fields

Encoding	IP bit	Hardware Interrupt Source
0	0	Reserved
1	1	Reserved
2	2	HW0
3	3	HW1
4	4	HW2
5	5	HW3
6	6	HW4
7	7	HW5

2.3.2 TLB Management Registers

This section contains the following TLB management registers.

- Section 2.3.2.1, "Index (CP0 Register 0, Select 0)" on page 108
- Section 2.3.2.2, "Random (CP0 Register 1, Select 0)" on page 109
- Section 2.3.2.3, "EntryLo0 - EntryLo1 (CP0 Registers 2 and 3, Select 0)" on page 110
- Section 2.3.2.4, "EntryHi (CP0 Register 10, Select 0)" on page 112
- Section 2.3.2.5, "Context (CP0 Register 4, Select 0)" on page 113
- Section 2.3.2.6, "Context Configuration — ContextConfig (CP0 Register 4, Select 1)" on page 114
- Section 2.3.2.6, "Context Configuration — ContextConfig (CP0 Register 4, Select 1)" on page 114
- Section 2.3.2.7, "PageMask (CP0 Register 5, Select 0)" on page 115
- Section 2.3.2.8, "Page Granularity — PageGrain (CP0 Register 5, Select 1)" on page 116
- Section 2.3.2.9, "Wired (CP0 Register 6, Select 0)" on page 117
- Section 2.3.2.10, "Bad Virtual Address — BadVAddr (CP0 Register 8, Select 0)" on page 118

2.3.2.1 Index (CP0 Register 0, Select 0)

Index is used as the TLB index when reading or writing the TLB with **TLBR/TLBWI/TLBINV/TLBINVF** respectively. It is also set by a TLB probe (**TLBP**) instruction to return the location of an address match in the TLB.

During execution of a **TLBR** instruction, the Index field that was previously written by software or by a TLBP instruction is used to indicate the TLB entry to be read. Hardware then uses this information to perform the read operation.

During execution of a **TLBWI**, **TLBINV**, or **TLBINVF** instruction, the Index field that was previously written by software or by a TLBP instruction is used to indicate the TLB entry to be written or invalidated. Hardware then uses this information to perform the respective write or invalidate operation.

Prior to executing a **`TLBP`** instruction, the VPN to be searched should have been written to the VPN2 field in the *EntryHi* register. During the **`TLBP`** instruction, hardware searches the TLB array for a match to the VPN stored in the *EntryHi* register. If a match is found, hardware writes the index into the *Index* field of this register.

The *P* bit of this register is set by hardware to indicate that a match was not found. If this bit is not set, software can then read the corresponding index from this register.

The size of the index field depends on whether the device is configured with a fixed TLB (FTLB). In the proAptiv architecture, the VTLB is 64 dual entries, and the FTLB is 512 dual entries. If an FTLB is implemented and enabled, the Index field is 10 bits wide. If the FTLB is not implemented or not enabled, the Index field is 6 bits wide and is used to index the VTLB. This is shown in Figure 2.15 below. A hardware lookup can occur on the VTLB and the FTLB simultaneously.

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

Figure 2.15 Index Register Format

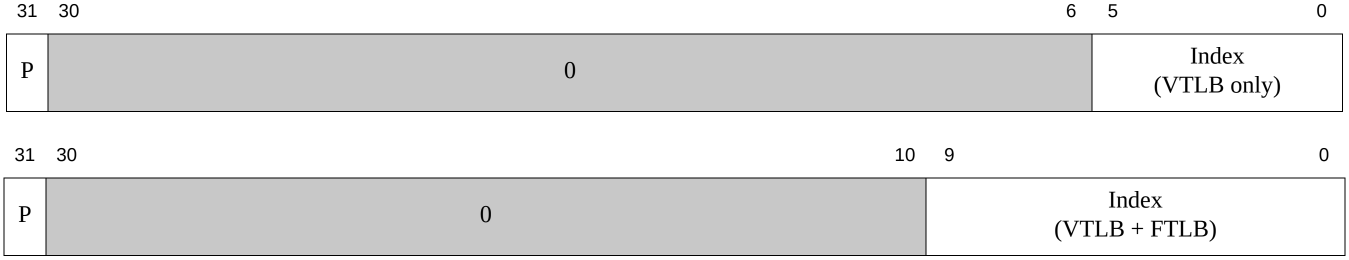


Table 2.18 Field Descriptions for Index Register

Name	Bit(s)	Description	Read/ Write	Reset State
P	31	Probe Failure. This bit is automatically set when a **`TLBP`** search of the TLB fails to find a matching entry.	R	Undefined
0	30:10 or 30:6	Must be written as zero; returns zero on reads.	0	0
Index	9:0 or 5:0	An index into the TLB used for **`TLBR, TLBWI, TLBINV`** and **`TLBINVF`** instructions. This field is set by the **`TLBP`** instruction when it finds a matching entry.	R/W	Undefined

2.3.2.2 Random (CP0 Register 1, Select 0)

The *Random* register is a read-only register whose value is used to index the VTLB during a **`TLBWR`** instruction. It provides a quick way of replacing a VTLB entry at random. As a result, it will not take values less than the value programmed in the *Wired* register. The *Random* register employs a pseudo-random least-recently-used (LRU) algorithm that ensures that no wired entries are selected, Only those LRU entries that are not in the *Wired* register are targeted for replacement. The contents of the *Random* register are modified after a VTLB write, or on a write to the *Wired* register.

The processor initializes the *Random* register to the reflect the maximum number of entries (63) on a Reset exception. Note that the Random register is used only for VTLB accesses. It is not used for FTLB accesses.

Figure 2.16 Random Register Format

31 6	5 0
0	Random

Table 2.19 Field Descriptions for Random Register

Name	Bit(s)	Description	Read/ Write	Reset State
0	31:6	Must be written as zero; returns zero on reads.	0	0
Random	5:0	This field cycles "randomly" through the potential indices of the VTLB, so its length varies with the VTLB size. It is a pseudo-least-recently-used VTLB index. Refer to the MMUSize field (bits [30:25]) in Section 2.3.1.2 "Device Configuration 1 — Config1 (CP0 Register 16, Select 1)" for more information on the VTLB size. This field is not used for FTLB accesses.	R	VTLB Entries — 1

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

The pair of *EntryLo* registers act as the interface between the TLB and the **TLBR, TLBWI,** and **TLBWR** instructions. These registers store the contents of a TLB entry. Each entry maps a pair of pages. The *EntryLo0* and *EntryLo1* register store even and odd numbered virtual pages respectively. These registers are read during a **TLBWR** or **TBLWI** instruction, and written by a **tlbr** instruction. They are not used for any other purpose.

Figure 2.17 EntryLo0 and EntryLo1 Register Format

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

Table 2.20 Field Descriptions for EntryLo0 and EntryLo1 Registers

Name	Bit(s)	Description	Read/ Write	Reset State
RI	31	Read Inhibit. If this bit is set in a TLB entry, any attempt (other than a MIPS16 PC-relative load) to read data on the virtual page causes either a TLB Invalid or a TLBRI exception, even if the V (Valid) bit is set. The RI bit is writable only if the RIE bit of the *PageGrain* register is set. For more information, refer to Section 2.3.2.8, "Page Granularity — PageGrain (CP0 Register 5, Select 1)". If the RIE bit of the *PageGrain* register is not set, the RI bit of *Entry 0* and *Entry 1* are set to zero on any write to the register, regardless of the value written.	R/W	Undefined
XI	30	Execute Inhibit. If this bit is set in a TLB entry, any attempt to fetch an instruction or to load MIPS16 PC-relative data from the virtual page causes a TLB Invalid or a TLBXI exception, even if the V (Valid) bit is set. The XI bit is writable only if the XIE bit of the *PageGrain* register is set. For more information, refer to Section 2.3.2.8, "Page Granularity — PageGrain (CP0 Register 5, Select 1)". If the XIE bit of the *PageGrain* register not set, the XI bit of TLB Entry 0 - 1 is set to zero on any write to the register, regardless of the value written.	R/W	Undefined

Table 2.20 Field Descriptions for EntryLo0 and EntryLo1 Registers

Name	Bit(s)	Description	Read/ Write	Reset State
U	29:26	The upper 4 bits of the PFN cannot be written by software and will return 0 on reads.	R/W	Undefined
PFN	25:6	The "Physical Frame Number" represents bits 31:12 of the physical address. The 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. The proAptiv Multiprocessing System core supports a 32-bit physical address bus.	R/W	Undefined
C	5:3	Coherency attribute of the page. See Table 2.21.	R/W	Undefined
D	2	The "Dirty" flag. Indicates that the page has been written, and/or is writable. If this bit is a one, stores to the page are permitted. If this bit is a zero, stores to the page cause a TLB Modified exception. Software can use this bit to track pages that have been written to. When a page is first mapped, this bit should be cleared. It is set on the first write that causes an exception.	R/W	Undefined
V	1	The “Valid” flag. Indicates that the TLB entry, and thus the virtual page mapping, are valid. If this bit is a set, accesses to the page are permitted. If this bit is a zero, accesses to the page cause a *TLB Invalid* exception. This bit can be used to make just one of a pair of pages valid.	R/W	Undefined
G	0	The “Global” bit. On a TLB write, the logical AND of the G bits in both the *Entry 0* and *Entry 1* registers become the G bit in the TLB entry. If the TLB entry G bit is a one, then the ASID comparisons are ignored during TLB matches. On a read from a TLB entry, the G bits of both *Entry 0* and *Entry 1* reflect the state of the TLB G bit.	R/W	Undefined

Table 2.21 Cache Coherency Attributes Encoding of the C Field

C[5:3] / K0[2:0][1]	Name	Cache Coherency Attribute
0	—	Reserved
1	—	Reserved
2	UC	Uncached, non-coherent
3	WB	Cacheable, noncoherent, write-back, write allocate
4	CWBE	Cacheable, coherent, write-back, write-allocate, read misses request Exclusive
5	CWB	Cacheable, coherent, write-back, write-allocate, read misses request Shared
6	—	Reserved
7	UCA	Uncached Accelerated, non-coherent

[1] State of the K0 field at bits 2:0 of the Config register. See Section 2.3.1.1 “Device Configuration — Config (CP0 Register 16, Select 0)”

2.3.2.4 EntryHi (CP0 Register 10, Select 0)

The *EntryHi* register contains the upper portion of the virtual address match information used for TLB read, write, and access operations. The remaining information is stored in the *EntryLo0* and *EntryLo1* registers described in Section 2.3.2.3 “EntryLo0 - EntryLo1 (CP0 Registers 2 and 3, Select 0)”.

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

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

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

The $EntryHi_{EHINV}$ field has been added to support explicit invalidation of TLB entries via the **TLBWI** instruction. When $EntryHi_{EHINV}$ = 1, the **TLBWI** instruction acts as a TLB invalidate operation, setting the hardware valid bit associated with a TLB entry to the invalid state. When $EntryHi_{EHINV}$ = 1, only the *Index* register is required to be valid. Behavior of the TLBWR instruction is unmodified by $EntryHi_{EHINV}$. The **TLBR** instruction copies the EHINV bit from the TLB Entry to $EntryHI_{EHINV}$. Note that execution of the **TLBP** instruction does not change this value.

Figure 2.18 EntryHi Register Format

31 13	12 11	10	9 8	7 0
VPN2	0	EHINV	0	ASID

Table 2.22 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**. This field consists of $VA_{31:13}$ of the virtual address (virtual page number / 2). It is also the virtual address to be written into the TLB on a **TLBWI** and **TLBWR,** and the destination of the virtual address on a **TLBR**. On a TLB-related exception, the VPN2 field is automatically set to the virtual address that was being translated when the exception occurred. This field is written by software before a **TLBP** or **TLBWI** and written by hardware in all other cases.	R/W	Undefined
0	12:11	Reserved. Write as zero. Ignored on reads.	R	0
EHINV	10	TLBWI invalidate enable. When this bit is set, the TLBWI instruction acts as a TLB invalidate operation, setting the hardware valid bit associated with the TLB entry to the invalid state. When this bit is set, the *PageMask* and *EntryLo0/EntryLo1* registers do not need to be valid. Only the *Index* register is required to be valid. This bit is ignored on a **TLBWR** instruction.	R/W	0
0	9:8	Reserved. Write as zero. Ignored on reads.	R	0

Table 2.22 Field Descriptions for EntryHi Register

Name	Bit(s)	Description	Read/ Write	Reset State
ASID	7:0	Address space identifier. This field 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 and help to map address translations for the current process. This field is written by hardware on a TLB read and by software to establish the current ASID value for TLB write and against which TLB references match each entry's TLB ASID field. This field supports up to 256 unique ASID values, consisting of a virtual tag that is in addition to the 32-bit address.	R/W	0

2.3.2.5 Context (CP0 Register 4, Select 0)

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

The *BadVPN2* field of the *Context* register is not defined after an address error exception, and this field may be modified by hardware during the address error exception sequence.

The pointer implemented by the *Context* register can point to any power-of-two-sized PTE structure within memory. This allows the TLB refill handler to use the pointer without additional shifting and masking steps. For example, if the low-order bit of the PTEBase field is 20, the page table entry (PTE) structure occurs on a 1M boundary. If the low-order bit is 21, PTE structure occurs on a 2M boundary, etc. Depending on the value in the *ContextConfig* register, it may point to an 8-byte pair of 32-bit PTEs within a single-level page table scheme, or to a first level page directory entry in a two-level lookup scheme.

A TLB exception (Refill, Invalid, Modified, Read Inhibit, Execute Inhibit) causes the virtual address to be written to a variable range of bits, defined as (X-1):Y of the *Context* register. This range corresponds to the contiguous range of set bits in the *ContextConfig* register. Bits 31:X, Y-1:0 are read/write to software and are unaffected by the exception.

For example, if X = 23 and Y = 4, i.e. bits 22:4 are set in *ContextConfig*, the behavior is identical to the standard MIPS32 *Context* register (bits 22:4 are filled with VA$_{31:13}$). Although the fields have been made variable in size and interpretation, the MIPS32 nomenclature is retained. Bits 31:X are referred to as the *PTEBase* field, and bits X-1:Y are referred to as *BadVPN2*.

The value of the *Context* register is **UNPREDICTABLE** following a modification of the contents of the *ContextConfig* register. After the *ContextConfig* register is modified, software should write the PTEBase field of the *Context* register. However, note that the contents of the BadVPN2 field will not be valid until the next TLB exception.

Figure 2.19 shows the format of the *Context* Register; Table 2.23 describes the *Context* register fields.

Figure 2.19 Context Register Format

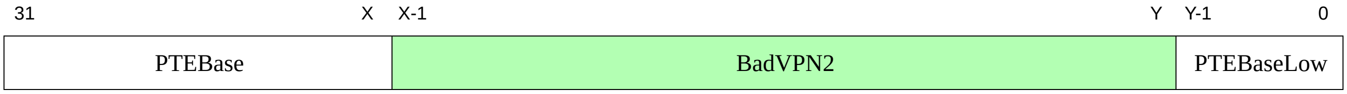


Table 2.23 Context Register Field Descriptions

Fields				
Name	**Bits**	**Description**	**Read / Write**	**Reset State**
PTEBase	Variable, 31:X where X in {31:0}	This field is for use by the operating system and is normally written with a value that allows the operating system to use the *Context* Register as a pointer to an array of data structures in memory corresponding to the address region that contains the virtual address which caused the exception. The size of the BadVPN2 field is determined by number of contiguous ‘ones’ in the VirtualIndex field of the *ContextConfig* register described below. If the VirtualIndex field is all ‘ones’, then the BadVPN2 field is comprised of bits 22:2. If the VirtualIndex field is all ‘zero’, then there is no BadVPN and the PTEBase and PTEBase low fields are merged together to form a single 32-bit PTEBase value.	R/W	Undefined
BadVPN2	Variable, (X-1):Y where X in {23:Y} and Y in {22:2}.	This field is written by hardware on a TLB exception. It contains bits $VA_{31:32-X+Y}$ of the virtual address that caused the exception.	R	Undefined
PTEBaseLow	Variable, (Y-1):0 where Y in {22:2}.	This field is for use by the operating system and is normally written with a value that allows the operating system to use the *Context* Register as a pointer to an array of data structures in memory corresponding to the address region that contains the virtual address which caused the exception.	R/W	Undefined

2.3.2.6 Context Configuration — ContextConfig (CP0 Register 4, Select 1)

The *ContextConfig* register defines the bits of the *Context* register into which the high order bits of the virtual address causing a TLB exception will be written, and how many bits of that virtual address will be extracted. Bits above the selected *BadVPN2* field of the *Context* register are read/write to software and serve as the *PTEBase* field. Bits below the selected *BadVPN2* field of the *Context* register serve as the *PTEBaseLow* field.

Software writes a set of contiguous ones to the *VirtualIndex* field of the *ContextConfig* register. Hardware then determines which bits of this register are high and low. The highest order bit that is a logic ‘1’ serves as the MSB of the *BadVPN2* field of the *Context* register. The lowest order bit that is a logic ‘1’ serves as the LSB of the *BadVPN2* field of the *Context* register. A value of all zero’s in the *VirtualIndex* field means that the full 32 bits of the *Context* register are R/W for software and are unaffected by TLB exceptions.

Figure 2.20 shows the formats of the *ContextConfig* register; Table 2.24 describes the *ContextConfig* register fields.

Figure 2.20 ContextConfig Register Forma t

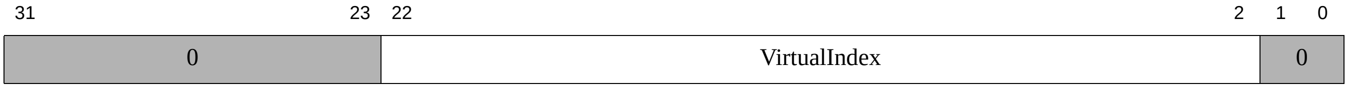


Table 2.24 ContextConfig Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:23	Ignored on write; returns zero on read.	R	0x00
VirtualIndex	22:2	A mask of 0 to 21 contiguous 1 bits in this field causes the corresponding bits of the *Context* register to be written with the high-order bits of the virtual address causing a TLB exception. Behavior of the processor is **UNDEFINED** if non-contiguous 1 bits are written into the register field. Note that it is the responsibility of software to ensure that this field is written with contiguous ones because if non-contiguous 1 bits are written, no exception will be taken.	R/W	0x1F_FFFC
0	1:0	Ignored on write; returns zero on read.	R	0

2.3.2.7 PageMask (CP0 Register 5, Select 0)

Every TLB entry has an independent virtual-address mask that 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.

Software can determine the maximum page size supported by writing all ones to the *PageMask* register, then reading the value back. If a pair of bits reads back as ones, the processor implements that page size. Note that the bits are read in pairs, so bits 14:13 are read first and can have only a value of 00 or 11. If they are both 11, bits 16:15 are read, and so on.

The operation of the processor is **UNDEFINED** if software loads the *Mask* field with a value other than one of those listed in Table 2.26, even if the hardware returns a different value on read. Hardware may depend on this requirement in implementing hardware structures.

Figure 2.21 PageMask Register Format

31 29	28 13	12 0
0	Mask	0

Table 2.25 Field Descriptions for PageMask Register

Name	Bit(s)	Description	Read/ Write	Reset State
0	31:29	Ignored on write; returns zero on read.	R	0
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 (i.e., with "1"s filling the *PageMask*$_{Mask}$ field from low bits upward, two at a time)	R/W	Undefined
0	12:0	Ignored on write; returns zero on read.	R	0

Table 2.26 PageMask Register Values

PageMask Register 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

Software may determine which page sizes are supported by writing all ones to the *PageMask* register, then reading the value back. If a pair of bits reads back as ones, the processor implements that page size. The operation of the processor is **UNDEFINED** if software loads the *Mask* field with a value other than one of those listed in Table 2.26, even if the hardware returns a different value on read. Hardware may depend on this requirement in implementing hardware structures.

2.3.2.8 Page Granularity — PageGrain (CP0 Register 5, Select 1)

The PageGrain register is a read/write register used for XI/RI TLB protection bits.The PageGrain register is present in Release 3 (and subsequent releases) of the architecture.

Figure 2.22 shows the format of the PageGrain register; Table 2.27 describes the PageGrain register fields.

Figure 2.22 PageGrain Register Format

31	30	29	28	27	26 13	12 8	7 0
RIE	XIE	0	ESP	IEC	0	ASE	0

Table 2.27 Field Descriptions for PageGrain Register

Name	Bit(s)	Description	Read/ Write	Reset State
RIE	31	Read inhibit enable. 0: RI bit of the *Entry0* and *Entry1* registers is disabled and not writeable by software. 1: RI bit of the *Entry0* and *Entry1* registers is enabled.	R/W	0
XIE	30	Execute inhibit enable. 0: XI bit of the *Entry0* and *Entry1* registers is disabled and not writeable by software. 1: XI bit of the *Entry0* and *Entry1* registers is enabled.	R/W	0
0	29	Reserved. Ignored on write; returns zero on read.	R	0
ESP	28	This bit is always 0 as 1K pages are not supported. This bit must be written with 0.	R	0

Table 2.27 Field Descriptions for PageGrain Register *(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
IEC	27	Enables unique exception codes for the Read-Inhibit and Execute-Inhibit exceptions. 0: Read-Inhibit and Execute-Inhibit exceptions both use the TLBL exception code. 1: Read-Inhibit exceptions use the TLBRI exception code. Execute-Inhibit exceptions use the TLBXI exception code.	R/W	0
0	26:13	Reserved. Ignored on write; returns zero on read.	R	0
ASE	12:8	Ignored on write; returns zero on read.	R	0
0	7:0	Reserved. Ignored on write; returns zero on read.	R	0

2.3.2.9 Wired (CP0 Register 6, Select 0)

The *Wired* register is a read/write register that specifies the boundary between the wired and random entries in the TLB as shown in Figure 2.28. Wired entries are fixed, non-replaceable entries that cannot be overwritten by a TLBWR instruction. Wired entries can be overwritten by a TLBWI instruction.

Note that wired entries in the TLB must be contiguous and start from 0. For example, if the Wired field of this register contains a value of 5, this indicates that entries 4, 3, 2, 1, and 0 of the VTLB are wired. Refer to Section 3.9, "Hardwiring VTLB Entries" for more information.

The *Wired* register is reset to zero by a Reset exception. Writing the *Wired* register may cause the *Random* register to change state.

The operation of the processor is undefined if a value greater than or equal to the number of VTLB entries is written to the *Wired* register. *Wired* can be set to a non-zero value to prevent the random replacement of up to 63 VTLB pages.

Figure 2.23 Wired Register Format

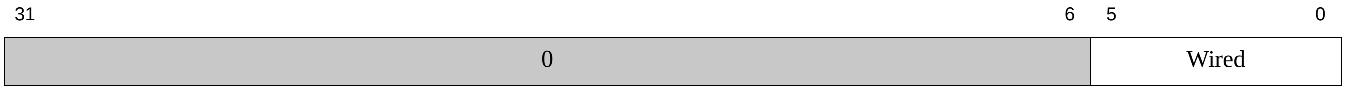


Table 2.28 Field Descriptions for Wired Register

Name	Bit(s)	Description	Read/ Write	Reset State
0	31:6	Ignored on write; returns zero on read.	R	0

Table 2.28 Field Descriptions for Wired Register

Name	Bit(s)	Description	Read/ Write	Reset State
Wired	5:0	Defines the number of wired dual entries in the TLB. Up to 63 of the 64 dual entries in the VTLB can be hard wired using this register. A value of 0 in this field indicates that no VTLB entries are hard wired. A value of 0x3F indicates that 63 of the 64 VTLB entries are hard wired.This field is encoded as follows: 0x00: 0 TLB entries are hardwired 0z01: 1 TLB entry is hardwired 0x02: 2 TLB entries are hardwired ...... 0x3F: 63 TLB entries are hardwired These entries become a good place for an OS to keep translations which must never cause a TLB translation-not-present exception.	R/W	0

2.3.2.10 Bad Virtual Address — BadVAddr (CP0 Register 8, Select 0)

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

- Address error (AdEL or AdES)
- TLB Refill
- TLB Invalid (TLBL, TLBS)
- TLB Read Inhibit (TLBRI)
- TLB Execute Inhibit (TLBXI)
- TLB Modified

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

There is more information about this register in the notes to the $Cause_{ExcCode}$ field.

Figure 2.24 BadVAddr Register Format

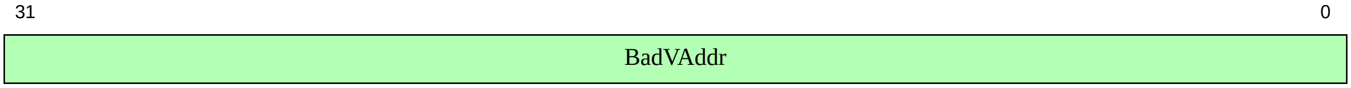


Table 2.29 BadVAddr Register Field Description

Fields		Description	Read / Write	Reset State
Name	Bits			
BadVAddr	31:0	Bad virtual address. This register stores the virtual address that causes one of the TLB exceptions listed above.	R	Undefined

2.3.3 Memory Segmentation Registers

This section contains the following memory segmentation registers.

- Section 2.3.3.1, "Segmentation Control 0 — SegCtl0 (CP0 Register 5, Select 2)" on page 119
- Section 2.3.3.2, "Segmentation Control 1 — SegCtl1 (CP0 Register 5, Select 3)" on page 120
- Section 2.3.3.3, "Segmentation Control 2 — SegCtl2 (CP0 Register 5, Select 4)" on page 121

Programmable segmentation allows for the virtual address space segments to be programmed with different access modes and attributes. Control of the 4GB of virtual address space is divided into six segments that are controlled using three CP0 registers; *SegCtl0* through *SegCtl2*. Each register has two 16-bit fields. Each field controls one of the six address segments as shown in Table 2.30. For more information, refer to Section 2.6 of the MMU chapter of this manual.

Table 2.30 Programmable Segmentation Register Interface

Register	CP0 Location	Memory Segment	Register Bits	Virtual Address Space Controlled	Virtual Address Range (Hex)
SegCtl2	Register 5 Select 4	CFG5	31:16	0.0 GB to 1.0 GB	0x0000_0000 - 0x3FFF_FFFF
		CFG4	15:0	1.0 GB to 2.0 GB	0x4000_0000 - 0x7FFF_FFFF
SegCtl1	Register 5 Select 3	CFG3	31:16	2.0 GB to 2.5 GB	0x8000_0000 - 0x9FFF_FFFF
		CFG2	15:0	2.5 GB to 3.0 GB	0xA000_0000 - 0xBFFF_FFFF
SegCtl0	Register 5 Select 2	CFG1	31:16	3.0 GB to 3.5 GB	0xC000_0000 - 0xDFFF_FFFF
		CFG0	15:0	3.5 GB to 4.0 GB	0xE000_0000 - 0xFFFF_FFFF

2.3.3.1 Segmentation Control 0 — SegCtl0 (CP0 Register 5, Select 2)

The *SegCtl0* register works in conjunction with the *SegCtl1* and *SegCtl2* registers to allow for configuration of the memory segmentation system. The address is split into the six segments defined in Table 2.30.

Figure 2.25 shows the format of the *SegCtl0* Register. Note that the *Config3*$_{SR}$ bit must be set to enable this register.

Figure 2.25 SegCtl0 Register Format (CP0 Register 5, Select 2)

31 25	24 23	22 20	19	18 16	15 9	8 7	6 4	3	2 0
CFG1_PA	0	CFG1_AM	CFG1_EU	CFG1_C	CFG 0_PA	0	CFG0_AM	CFG0_EU	CFG0_C

Table 2.31 SegCtl0 Register Field Descriptions

Fields				
Name	**Bits**	**Description**	**Read / Write**	**Reset State**
CFG1_PA	31:25	Physical address bits 31:29 for segment 1. For use when unmapped. Bits 27:25 correspond to physical address bits 31:29. Bits 31:28 are reserved for future expansion. For more information, refer to Section 3.5.2.4 "Defining the Physical Address Range for Each Memory Segment"	R/W	Configuration Dependent
0	24:23	Reserved.	RO	0
CFG1_AM	22:20	Configuration 1 access control mode. See Table 2.34 for encoding. For more information, refer to Section 3.5.2.3 "Setting the Access Control Mode".	R/W	Configuration Dependent
CFG1_EU	19	Error condition behavior. Configuration segment 1 becomes unmapped and uncached when $Status_{ERL}$ = 1.	R/W	Configuration Dependent
CFG1_C	18:16	Cache coherency attribute for segment 1. The encoding of the CFG1_C field is the same as the C field of the EntryLo0/EntryLo1 registers described in Section 2.3.2.3. Refer to Table 2.21 for the encoding of this field. For more information, refer to Section 3.5.2.3 "Setting the Access Control Mode".	R/W	Configuration Dependent
CFG0_PA	15:9	Physical address bits 31:29 for segment 0. For use when unmapped. Bits 11:9 correspond to physical address bits 31:29 for segment 0. Bits 15:12 are reserved for future expansion. For more information, refer to Section 3.5.2.4 "Defining the Physical Address Range for Each Memory Segment".	R/W	Configuration Dependent
0	8:7	Reserved.	RO	0
CFG0_AM	6:4	Configuration 0 access control mode. See Table 2.34 for encoding. For more information, refer to Section 3.5.2.3 "Setting the Access Control Mode".	R/W	Configuration Dependent
CFG0_EU	3	Error condition behavior. Configuration segment 0 becomes unmapped and uncached when $Status_{ERL}$ = 1.	R/W	Configuration Dependent
CFG0_C	2:0	Cache coherency attribute for segment 0. The encoding of the CFG0_C field is the same as the C field of the EntryLo0/EntryLo1 registers described in Section 2.3.2.3. Refer to Table 2.21 for the encoding of this field. For more information, refer to Section 3.5.2.3 "Setting the Access Control Mode".	R/W	Configuration Dependent

2.3.3.2 Segmentation Control 1 — SegCtl1 (CP0 Register 5, Select 3)

The *SegCtl1* register works in conjunction with the *SegCtl0* and *SegCtl2* registers to allow for configuration of the memory segmentation system. The address is split into six segments defined in Table 2.30.

Segmentation Control allows address-specific behaviors defined by the Privileged Resource Architecture to be modified or disabled.

Figure 2.26 shows the format of the *SegCtl1* Register. Note that the $Config3_{SR}$ bit must be set to enable this register. For more information on the reset states of these fields, refer to Chapter 3, "", Section 3.6.6 "Switching From Legacy Mode to EVA Mode After Boot-up".

Figure 2.26 SegCtl1 Register Format (CP0 Register 5, Select 3)

31 25	24 23	22 20	19	18 16	15 9	8 7	6 4	3	2 0
CFG3_PA	0	CFG3_AM	CFG3_EU	CFG3_C	CFG2_PA	0	CFG2_AM	CFG2_EU	CFG2_C

Table 2.32 SegCtl1 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
CFG3_PA	31:25	Physical address bits 31:29 for segment 3. For use when unmapped. Bits 27:25 correspond to physical address bits 31:29. Bits 31:28 are reserved for future expansion. For more information, refer to Section 3.5.2.4 "Defining the Physical Address Range for Each Memory Segment".	R/W	Configuration Dependent
0	24:23	Reserved. Must be written as zeros; returns zeros on reads.	RO	0
CFG3_AM	22:20	Configuration 3 access control mode. See Table 2.34 for encoding. For more information, refer to Section 3.5.2.3 "Setting the Access Control Mode".	R/W	Configuration Dependent
CFG3_EU	19	Error condition behavior. Configuration segment 3 becomes unmapped and uncached when Status$_{ERL}$ = 1.	R/W	Configuration Dependent
CFG3_C	18:16	Cache coherency attribute for segment 3, for use when unmapped. As defined by the base architecture. For more information, refer to Section 3.5.2.3 "Setting the Access Control Mode".	R/W	Configuration Dependent
CFG2_PA	15:9	Physical address bits 31:29 for segment 2. For use when unmapped. Bits 11:9 correspond to physical address bits 31:29 for segment 0. Bits 15:12 are reserved for future expansion. For more information, refer to Section 3.5.2.4 "Defining the Physical Address Range for Each Memory Segment".	R/W	Configuration Dependent
0	8:7	Reserved. Must be written as zeros; returns zeros on reads.	RO	0
CFG2_AM	6:4	Configuration 2 access control mode. See Table 2.34 for encoding. For more information, refer to Section 3.5.2.3 "Setting the Access Control Mode".	R/W	Configuration Dependent
CFG2_EU	3	Error condition behavior. Configuration segment 2 becomes unmapped and uncached when Status$_{ERL}$ = 1.	R/W	Configuration Dependent
CFG2_C	2:0	Cache coherency attribute for segment 2, for use when unmapped. As defined by the base architecture. For more information, refer to Section 3.5.2.3 "Setting the Access Control Mode".	R/W	Configuration Dependent

2.3.3.3 Segmentation Control 2 — SegCtl2 (CP0 Register 5, Select 4)

The *SegCtl2* register works in conjunction with the *SegCtl0* and *SegCtl1* registers to allow for configuration of the memory segmentation system. The address is split into six segments defined in Table 2.30.

Segmentation Control allows address-specific behaviors defined by the Privileged Resource Architecture to be modified or disabled.

Figure 2.27 shows the format of the *SegCtl2* Register. Note that the *Config3$_{SR}$* bit must be set to enable this register. For more information on the reset states of these fields, refer to Chapter 3, "", Section 3.6.6 "Switching From Legacy Mode to EVA Mode After Boot-up".

Figure 2.27 SegCtl2 Register Format (CP0 Register 5, Select 4)

31 25	24 23	22 20	19	18 16	15 9	8 7	6 4	3	2 0
CFG5_PA	0	CFG5_AM	CFG5_EU	CFG5_C	CFG4_PA	0	CFG4_AM	CFG4_EU	CFG4_C

Table 2.33 SegCtl2 Register Field Descriptions

Fields				
Name	**Bits**	**Description**	**Read / Write**	**Reset State**
CFG5_PA	31:25	Physical address bits 31:29 for segment 5. For use when unmapped. Bits 27:25 correspond to physical address bits 31:29. Bits 31:28 are reserved for future expansion. Note that for this field, bit 25 is ignored since CFG5 is mapped to a 1 GByte boundary. For more information, refer to Section 3.5.2.4 “Defining the Physical Address Range for Each Memory Segment”.	R/W	Configuration Dependent
0	24:23	Reserved.	RO	
CFG5_AM	22:20	Configuration 5 access control mode. See Table 2.34 for encoding. For more information, refer to Section 3.5.2.3 “Setting the Access Control Mode”.	R/W	Configuration Dependent
CFG5_EU	19	Error condition behavior. Configuration segment 5 becomes unmapped and uncached when Status$_{ERL}$ = 1.	R/W	Configuration Dependent
CFG5_C	18:16	Cache coherency attribute for segment 5. The encoding of the CFG5_C field is the same as the C field of the EntryLo0/EntryLo1 registers described in Section 2.3.2.3. Refer to Table 2.21 for the encoding of this field. For additional information, refer to Section 3.5.2.3 “Setting the Access Control Mode”	R/W	Configuration Dependent
CFG4_PA	15:9	Physical address bits 31:29 for segment 4. For use when unmapped. Bits 11:9 correspond to physical address bits 31:29 for segment 0. Bits 15:12 are reserved for future expansion. Note that for this field, bit 9 is ignored since CFG4 is mapped to a 1 GByte boundary. For more information, refer to Section 3.5.2.4 “Defining the Physical Address Range for Each Memory Segment”.	R/W	Configuration Dependent
0	8:7	Reserved.	RO	
CFG4_AM	6:4	Configuration 4 access control mode. See Table 2.34 for encoding. For more information, refer to Section 3.5.2.3 “Setting the Access Control Mode”.	R/W	Configuration Dependent
CFG4_EU	3	Error condition behavior. Configuration segment 4 becomes unmapped and uncached when Status$_{ERL}$ = 1.	R/W	Configuration Dependent
CFG4_C	2:0	Cache coherency attribute for segment 4. The encoding of the CFG4_C field is the same as the C field of the EntryLo0/EntryLo1 registers described in Section 2.3.2.3. Refer to Table 2.21 for the encoding of this field. For additional information, refer to Section 3.5.2.3 “Setting the Access Control Mode”	R/W	Configuration Dependent

Table 2.34 describes the access control modes specifiable in the CFG_{AM} fields.

Table 2.34 Segment Configuration Access Control Modes

Mode		Action when referenced from Operating Mode			Description
		User mode	Supervisor mode	Kernel mode	
UK	000	Address Error	Address Error	Unmapped	Kernel-only unmapped region e.g. kseg0, kseg1
MK	001	Address Error	Address Error	Mapped	Kernel-only mapped region e.g. kseg3
MSK	010	Address Error	Mapped	Mapped	Supervisor and kernel mapped region e.g. ksseg, sseg
MUSK	011	Mapped	Mapped	Mapped	User, supervisor and kernel mapped region e.g. useg, kuseg, suseg
MUSUK	100	Mapped	Mapped	Unmapped	Used to implement a fully-mapped flat address space in user and supervisor modes, with unmapped regions which appear in kernel mode.
USK	101	Address Error	Unmapped	Unmapped	Supervisor and kernel unmapped region e.g. sseg in a fixed mapping TLB.
-	110	Undefined	Undefined	Undefined	Reserved
UUSK	111	Unmapped	Unmapped	Unmapped	Unrestricted unmapped region

Table 2.35 describes the state of each Segment Configuration register at reset in legacy mode.

Table 2.35 Segment Configuration Reset States in Legacy Mode

CFG	Segment	CFG_{AM}	CFG_{PA}	CFG_{C}	CFG_{EU}
0	kseg3	MK	Undefined	Undefined	0
1	ksseg, sseg	MSK	Undefined	Undefined	0
2	kseg1	UK	3’b000	2	0
3	kseg0	UK	3’b000	3	0
4	kuseg, suseg, useg	MUSK	3’b010	Undefined	1
5	kuseg, suseg, useg	MUSK	3’b000	Undefined	1

2.3.4 Exception Control Registers

This section contains the following exception control registers.

- Section 2.3.4.1, "Cause (CP0 Register 13, Select 0)" on page 124
- Section 2.3.4.2, "Exception Program Counter — EPC (CP0 Register 14, Select 0)" on page 127
- Section 2.3.4.3, "Error Exception Program Counter — ErrorEPC (CP0 Register 30, Select 0)" on page 128

Also refer to the Interrupt Control register in Section 2.3.1.12, "Interrupt Control — IntCtl (CP0 Register 12, Select 1)" on page 106.

2.3.4.1 Cause (CP0 Register 13, Select 0)

The *Cause* register describes the cause of the most recent exception and controls software interrupt requests and the vector through which interrupts are dispatched. With the exception of the *IP1:0*, *DC*, *IV*, and *WP* fields, all fields in the *Cause* register are read-only. *IP7:2* are interpreted as the Requested Interrupt Priority Level (RIPL) in External Interrupt Controller (EIC) interrupt mode.

Figure 2.28 Cause Register Format

31	30	29 28	27	26	25 24	23	22	21	20 16	15 10	9 8	7	6 2	1 0
BD	TI	CE	DC	PCI	0	IV	WP	FDCI	0	IP7-2	IP1-0	0	ExcCode	0

Table 2.36 Field Descriptions for Cause Register

Name	Bit(s)	Description	Read/ Write	Reset State
BD	31	Indicates whether the last exception taken occurred in a branch delay slot. 0: Exception taken was not in delay slot 1: Exception taken was in delay slot The processor updates *BD* only if the *EXL* bit in the *Status* register was zero when the exception occurred. If the exception occurred in a branch delay slot, the exception program counter (*EPC)* is set to restart execution at the branch. Software should read this bit to determine if the exception was taken in a delay slot.	R	Undefined
TI	30	Timer Interrupt. Denotes whether a timer interrupt is pending (analogous to the IP bits for other interrupt types) 0: No timer interrupt is pending 1: Timer interrupt is pending Hardware sets this bit based on the state of the external *SI_TimerInt* signal. See also the descriptions of the *Count* and *Compare* registers.	R	Undefined
CE	29:28	Coprocessor unit number referenced when a Coprocessor Unusable exception is taken. This field is loaded by hardware on every exception, but is **UNPREDICTABLE** for all exceptions except Coprocessor Unusable. 00: Coprocessor 0 01: Coprocessor 1 10: Coprocessor 2 (not supported in proAptiv 11: Coprocessor 3 (not supported in proAptiv)	R	Undefined

Table 2.36 Field Descriptions for Cause Register*(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
DC	27	Disable *Count* register. In some power-sensitive applications, the *Count* register is not used but may still be the source of some noticeable power dissipation. This bit allows the *Count* register to be stopped in such situations. For example, this can be useful during low-power operation following a **`wait`** instruction. 0: Enable counting of *Count* register 1: Disable counting of *Count* register	R/W	0
PCI	26	Performance Counter Interrupt. Indicates whether a performance counter interrupt is pending (analogous to the *IP* bits for other interrupt types). 0: No performance counter interrupt is pending 1: Performance counter interrupt is pending See also the description of the *PerfCnt* registers.	R	Undefined
0	25:24	Reserved. Write as zero. Ignored on reads.	R	0
IV	23	Indicates whether an interrupt exception uses the general exception vector or a special interrupt vector: 0: Use the general exception vector (0x180) 1: Use the special interrupt vector (0x200) When the *IV* bit in the *Cause* register is 1 and the *BEV* bit in the *Status* register is 0, the special interrupt vector represents the base of the vector interrupt table.	R/W	Undefined
WP	22	Indicates that a watch exception was deferred because either the *Status*$_{\text{EXL}}$ bit or the *Status*$_{\text{ERL}}$ bit was a logic '1' at the time the watch exception was detected. This bit both indicates that the watch exception was deferred, and causes the exception to be initiated when *Status*$_{\text{EXL}}$ and *Status*$_{\text{ERL}}$ are both zero. As such, software must clear this bit as part of the watch exception handler to prevent a watch exception loop. Software should never write a 1 to this bit when its value is a 0, thereby causing a 0-to-1 transition. If such a transition is caused by software, it is **UNPREDICTABLE** whether hardware ignores the write, accepts the write with no side effects, or accepts the write and initiates a watch exception once *Status*$_{\text{EXL}}$ and *Status*$_{\text{ERL}}$ are both zero. Software should clear this bit, but never set it. It is set by hardware.	R/W	Undefined
FDCI	21	Fast Debug Channel Interrupt: This bit denotes whether an FDC interrupt is pending (analogous to the *IP* bits for other interrupt types). 0: No FDC interrupt is pending 1: FDC interrupt is pending This bit is set by hardware based on the state of the external *SI_FDCInt* signal.	R	Undefined
0	20:16	Reserved. Write as zero. Ignored on reads.	R	0

Table 2.36 Field Descriptions for Cause Register*(continued)*

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Description</th><th>Read/
Write</th><th>Reset State</th></tr>
<tr><td><i>IP7-2</i>
<i>RIPL</i></td><td>15:10</td><td>Indicates an interrupt is pending.
If External Interrupt Controller (EIC) mode is disabled (<i>Config3</i>$_{VEIC}$ = 0), timer interrupts are combined in a system-dependent way with any hardware interrupt. Each bit of this field maps to an individual hardware interrupt.
<table>
<tr><th>Bit</th><th>Name</th><th>Meaning</th></tr>
<tr><td>15</td><td>IP7</td><td>Hardware interrupt 5</td></tr>
<tr><td>14</td><td>IP6</td><td>Hardware interrupt 4</td></tr>
<tr><td>13</td><td>IP5</td><td>Hardware interrupt 3</td></tr>
<tr><td>12</td><td>IP4</td><td>Hardware interrupt 2</td></tr>
<tr><td>11</td><td>IP3</td><td>Hardware interrupt 1</td></tr>
<tr><td>10</td><td>IP2</td><td>Hardware interrupt 0</td></tr>
</table>
If EIC interrupt mode is enabled (<i>Config3</i>$_{VEIC}$ = 1), these bits take on a different meaning and are interpreted as the Requested Interrupt Priority Level (<i>RIPL)</i> field.
When EIC interrupt mode is enabled, this field (RIPL) contains the encoded (0 - 63) value of the requested interrupt. A value of zero indicates that no interrupt is requested.</td><td>R</td><td>Undefined</td></tr>
<tr><td><i>IP1-0</i></td><td>9:8</td><td>Controls the request for software interrupts:
<table>
<tr><th>Bit</th><th>Name</th><th>Meaning</th></tr>
<tr><td>9</td><td>IP1</td><td>Request software interrupt 1</td></tr>
<tr><td>8</td><td>IP0</td><td>Request software interrupt 0</td></tr>
</table>
These bits are exported to an external interrupt controller for prioritization in EIC interrupt mode with other interrupt sources. The state of these bits are driven onto the external <i>SI_SWInt[1:0]</i> bus.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>0</td><td>7</td><td>Reserved. Write as zero. Ignored on reads.</td><td>R</td><td>0</td></tr>
<tr><td><i>ExcCode</i></td><td>6:2</td><td>Encodes the cause of the last exception as described in Table 2.37.</td><td>R</td><td>Undefined</td></tr>
<tr><td>0</td><td>1:0</td><td>Reserved. Write as zero. Ignored on reads.</td><td>R</td><td>0</td></tr>
</table>

Table 2.37 Exception Code Values in ExcCode Field of Cause Register

Value (decimal)	Value (hex)	Code	Description
0	0x0	Int	Interrupt
1	0x1	Mod	Store, but page marked as read-only in the TLB
2	0x2	TLBL	Load or fetch, but page not present or marked as invalid in the TLB
3	0x3	TLBS	Store, but page not present or marked as invalid in the TLB

Table 2.37 Exception Code Values in ExcCode Field of Cause Register *(continued)*

Value (decimal)	Value (hex)	Code	Description
4	0x4	AdEL	Address error on load/fetch or store respectively. Address is either wrongly aligned, or a privilege violation.
5	0x5	AdES	
6	0x6	IBE	Bus error signaled on instruction fetch
7	0x7	DBE	Bus error signaled on load/store (imprecise)
8	0x8	Sys	System call, i.e. **`syscall`** instruction executed.
9	0x9	Bp	Breakpoint, i.e. **`break`** instruction executed. If an SDBBP instruction is executed while the processor is running in EJTAG Debug Mode, this value is written to the $Debug_{DExcCode}$ field to denote an SDBBP in Debug mode.
10	0xA	RI	Reserved instruction. Instruction code not recognized (or not legal)
11	0xB	CpU	Coprocessor Unusable Exception. Instruction code was for a co-processor which is not enabled in $Status_{CU3-0}$.
12	0xC	Ov	Overflow exception. Overflow from a trapping variant of integer arithmetic instructions.
13	0xD	Tr	Trap exception. Condition met on one of the conditional trap instructions **`teq`** etc.
14	0xE	-	Reserved
15	0xF	FPE	Floating point unit exception — more details in the FPU control/status registers.
16	0x10	TLBPAR	TLB parity error exception.
17 - 18	0x11 - 0x12	-	Available for implementation-dependent use.
19	0x13	TLBRI	TLB read inhibit exception.
20	0x14	TLBXI	TLB execute inhibit exception.
21-22	0x15 - 0x16	-	Reserved.
23	0x17	WATCH	Instruction or data reference matched a watchpoint. Refer to *WatchHi*/*WatchLo* address.
24	0x18	MCheck	Machine check exception.
25	0x19	-	Reserved
26	0x1A	DSPDis	DSP ASE not enabled or not present exception. This exception occurs when trying to run an instruction from the MIPS DSP ASE, but the ASE is either not enabled or not available. If this exception occurs and the DSP ASE is present in the system, check the state of the $Status_{MX}$ bit to make sure it is set to '1'.
27-29	0x1B - 0x1D	-	Reserved.
30	0x1E	CacheErr	Parity/ECC error occurred somewhere in the proAptiv core, on either an instruction fetch, load, or cache refill. This exception does not occur during normal operation, but can occur while in debug mode. Refer to Section 2.3.9.1 "Debug (CP0 Register 23, Select 0)" for more information.
31	0x1F	-	Reserved.

2.3.4.2 Exception Program Counter — EPC (CP0 Register 14, Select 0)

Following an exception other than an error or debug exception, the *Exception Program Counter* (*EPC*) contains the address at which processing resumes after the exception has been serviced (the corresponding debug and error exception use *DEPC* and *ErrorEPC* respectively).

Unless the *EXL* bit in the *Status* register is set (indicating, among other things, that interrupts are disabled), the processor writes the *EPC* register when an exception occurs.

- For synchronous (precise) exceptions, *EPC* contains either:
  - The virtual address of the instruction that was the direct cause of the exception, or
  - The virtual address of the immediately preceding branch or jump instruction, when the exception causing instruction is in a branch delay slot, and the *Branch Delay* bit in the *Cause* register is set.
- For asynchronous (imprecise) exceptions, *EPC* contains the address of the instruction at which to resume execution.

The processor reads the *EPC* register as the result of execution of the **`eret`** instruction.

Figure 2.29 EPC Register Format

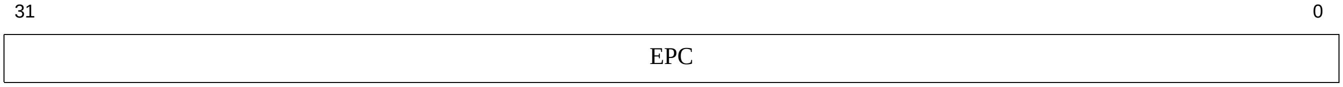


Table 2.38 EPC Register Field Description

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

2.3.4.3 Error Exception Program Counter — ErrorEPC (CP0 Register 30, Select 0)

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

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 2.30 ErrorEPC Register Format

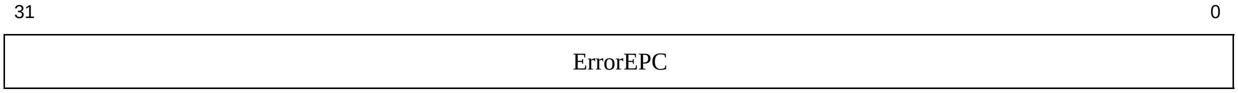


Table 2.39 ErrorEPC Register Field Description

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

2.3.5 Timer Registers

This section contains the following timer registers.

- Section 2.3.5.1, "Count (CP0 Register 9, Select 0)" on page 129
- Section 2.3.5.2, "Compare (CP0 Register 11, Select 0)" on page 129

2.3.5.1 Count (CP0 Register 9, Select 0)

The *Count* register acts as a timer, incrementing at a constant rate. Incrementing of this register occurs whether or not an instruction is executed, retired, or any forward progress is made through the pipeline. When enabled by clearing the *DC* bit in the *Cause* register, the counter increments every other clock (half the clock rate).

The *Count* may be stopped in either of the following two circumstances.

- Some implementations may stop *Count* in the low-power mode, for example, through the **`wait`** instruction, but only if the $\mathit{Cause}_{\mathit{DC}}$ flag is set to 1.
- When the device is in debug mode, the *Count* register can be stopped by setting $\mathit{Debug}_{\mathit{CountDM.}}$ By writing the $\mathit{Count}_{\mathit{DM}}$ bit, it is possible to control whether the *Count* register continues incrementing while the processor is in debug mode.

The *Count* field starts counting from whatever value is loaded into it. However, OS timers are usually implemented by leaving *Count* free-running and writing *Compare* as necessary. This counter rolls over when reaching it maximum value.

By writing the $\mathit{Count}_{\mathit{DM}}$ bit in the *Debug* register, it is possible to control whether the *Count* register continues incrementing while the processor is in debug mode.

Figure 2.31 Count Register Format

31	0
Count	

Table 2.40 Count Register Field Description

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

2.3.5.2 Compare (CP0 Register 11, Select 0)

The *Compare* register acts in conjunction with the *Count* register to implement a timer and timer interrupt function. When the value of the *Count* register equals the value of the *Compare* register, the *SI_TimerInt* output pin is asserted. *SI_TimerInt* remains asserted until the *Compare* register is written.

The *SI_TimerInt* output can be fed back into the proAptiv core on one of the interrupt pins to generate an interrupt. Traditionally, this has been done by multiplexing it with hardware interrupt 5 in order to set interrupt bit *IP*(7) in the *Cause* register.

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

Figure 2.32 Compare Register Format

31 … 0
Compare

Table 2.41 Compare Register Field Description

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

2.3.6 Cache Management Registers

This section contains the following cache management registers.

- Section 2.3.6.1, "Level 1 Instruction Cache Tag Low — ITagLo (CP0 Register 28, Select 0)" on page 130
- Section 2.3.6.2, "Level 1 Instruction Cache Tag High — ITagHi (CP0 Register 29, Select 0)" on page 132
- Section 2.3.6.3, "Level 1 Instruction Cache Data Low — IDataLo (CP0 Register 28, Select 1)" on page 133
- Section 2.3.6.4, "Level 1 Instruction Cache Data High — IDataHi (CP0 Register 29, Select 1)" on page 133
- Section 2.3.6.5, "Level 1 Data Cache Tag Low — DTagLo (CP0 Register 28, Select 2)" on page 134
- Section 2.3.6.6, "Level 1 Data Cache Data Low — DDataLo (CP0 Register 28, Select 3)" on page 140
- Section 2.3.6.7, "Level 2/3 Cache Tag Low — L23TagLo (CP0 Register 28, Select 4)" on page 141
- Section 2.3.6.8, "Level 2/3 Cache Data Low — L23DataLo (CP0 Register 28, Select 5)" on page 141
- Section 2.3.6.9, "Level 2/3 Cache Data High — L23DataHi (CP0 Register 29, Select 5)" on page 142
- Section 2.3.6.10, "ErrCtl (CP0 Register 26, Select 0)" on page 142
- Section 2.3.6.11, "Cache Error — CacheErr (CP0 Register 27, Select 0)" on page 144

2.3.6.1 Level 1 Instruction Cache Tag Low — ITagLo (CP0 Register 28, Select 0)

The *ITagLo* register acts as the interface to the instruction cache tag array. The Index Store Tag and Index Load Tag operations of the CACHE instruction use the *ITagLo* register as the source of tag information. Note that the proAptiv Multiprocessing System CPU does not implement the *ITagHi* register.

When the *WST* bit of the *ErrCtl* register is asserted, this register becomes the interface to the way-selection RAM. In this mode, the fields are redefined to give appropriate access the contents of the WS array instead of the Tag array.

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.

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 2.33 ITagLo Register Format (ErrCtl $_{WST}$ = 0, ErrCtl$_{SPR}$ = 0)

31 12	11 10	9 8	7	6	5	4 1	0
PTagLo	U	0	V	0	L	0	P

Table 2.42 Field Descriptions for ITagLo Register

Name	Bit(s)	Description	Read/ Write	Reset State
PTagLo	31:12	The cache address tag, which is a physical address because the proAptiv Multiprocessing System's caches are physically tagged. It holds bits 31:12 of the physical address, i.e., the low-order 12 bits of the address are implied by the position of the data in the cache.	R/W	Undefined
Unused	31:16	Unused field.	R/W	Undefined
0	9:8	Must be written as zero; returns zero on read.	0	0
V	7	Set to 1 if this cache entry is valid (set to zero to initialize the cache).	R/W	Undefined
0	6	Must be written as zero; returns zero on read.	0	0
L	5	Specifies the lock bit for the cache tag. This bit is set to lock this cache entry, preventing it from being replaced by another line when a cache miss occurs. When this bit is set, and the V bit is set, the corresponding cache line will not be replaced by the cache replacement algorithm. This can be used for critical data that must not be removed from the cache. However, this can reduce the efficiency of the cache for memory data competing for space at this index.	R/W	Undefined
0	4:1	Must be written as zero; returns zero on read.	0	0
P	0	Parity bit over the cache tag entries (excluding the *D* bit). This bit is updated with tag array parity on CACHE Index Load Tag operations and used as tag array parity on Index Store Tag operations when the *PO* bit of the *ErrCtl* register is set.	R/W	Undefined

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 the data in these fields either by `cache` load-tag or store-tag operations when *ErrCtl$_{WST}$* is set.

Figure 2.34 ITagLo Register Format (ErrCtl$_{WST}$ = 1, ErrCtl$_{SPR}$ = 0)

31 16	15 10	9 8	7	6	5	4 1	0
U	LRU	0	U	0	U	0	U

Table 2.43 Field Descriptions for ITagLo-WST Register

Name	Bit(s)	Description	Read/ Write	Reset State
LRU	15:10	LRU bits. This field contains the value read from the WS array after a CACHE Index Load WS operation. It is used to store into the WS array during CACHE Index Store WS operations. When reading or writing 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

ITagLo-WST (ErrCtl$_{WST}$ = 0, ErrCtl$_{SPR}$ = 1)

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

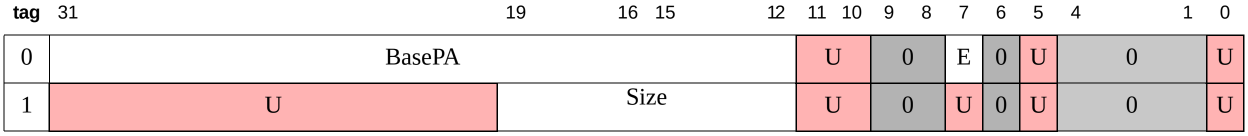


Figure 2.35 ITagLo Register Format (ErrCtl$_{WST}$ = 0, ErrCtl$_{SPR}$ = 1)

Table 2.44 Field Descriptions for ITagLo-SPR Register

Name	Bit(s)	Description	Read/ Write	Reset State
ErrCtl$_{WST}$ = 0, ErrCtl$_{SPR}$ = 1 — Tag 0				
BasePA	31:12	When reading pseudo-tag 0 of a scratchpad RAM, this field contains 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 indicates whether the scratchpad is enabled.	R/W	Undefined
ErrCtl$_{WST}$ = 0, ErrCtl$_{SPR}$ = 1 — Tag 1				
Size	19:12	When reading pseudo-tag 1 of a scratchpad RAM, this field indicates the size of the scratchpad array. This field is the number of 4KB sections it contains. Combined with bits 11:0, the register will contain the number of bytes in the scratchpad region.	R/W	Undefined

2.3.6.2 Level 1 Instruction Cache Tag High — ITagHi (CP0 Register 29, Select 0)

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

Figure 2.36 ITagHi Register Format

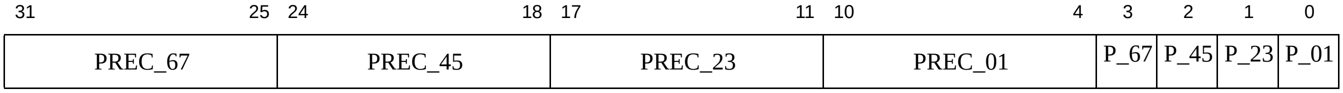


Table 2.45 Field Descriptions for ITagHi Register

Name	Bit(s)	Description	Read/ Write	Reset State
PREC_67	31:25	proAptiv Multiprocessing System 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

2.3.6.3 Level 1 Instruction Cache Data Low — IDataLo (CP0 Register 28, Select 1)

The *IDataLo* register is a register that acts as the interface to the instruction cache data array and is intended for diagnostic operations only. The Index Load Tag operation of the CACHE instruction reads the corresponding data values into the *IDataLo* register. If the *WST* bit in the *ErrCtl* register is set, then the contents of *IDataLo* can be written to the cache data array by doing an Index Store Data CACHE instruction. If the *SPR* bit in the *ErrCtl* register is set, then the contents of *IDataLo* can be written to the scratchpad RAM data array by doing an Index Store Data CACHE instruction.

Two registers (*IDataHi*, *IDataLo*) are needed, because the proAptiv Multiprocessing System loads I-cache data at least 64 bits at a time.

Figure 2.37 IDataLo Register Format

31 … 0
DATA

Table 2.46 IDataLo Register Field Description

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

2.3.6.4 Level 1 Instruction Cache Data High — IDataHi (CP0 Register 29, Select 1)

The *IDataHi* register is a register that acts as the interface to the cache data array and is intended for diagnostic operations only. The Index Load Tag operation of the CACHE instruction reads the corresponding data values into the *IDataHi* register. If the *WST* bit in the *ErrCtl* register is set, then the contents of *IDataHi* can be written to the cache data array by doing an Index Store Data CACHE instruction. If the *SPR* bit in the *ErrCtl* register is set, then the contents of *IDataHi* can be written to the scratchpad RAM data array by doing an Index Store Data CACHE instruction.

Because the interface to the I-cache only operates on pairs of instructions, two registers (*IDataHi*, *IDataLo*) are needed because the proAptiv Multiprocessing System loads I-cache data at least 64-bits at a time. The high instruction is written into the *IDataHi* register. Note that *IDataHi* and *IDataLo* reflect the memory ordering of the instructions.

Depending on the endianness of the system, Instruction0 belongs in either *IDataHi* (BigEndian) or *IDataLo* (LittleEndian) and vice versa for Instruction1.

Figure 2.38 IDataHi Register Format

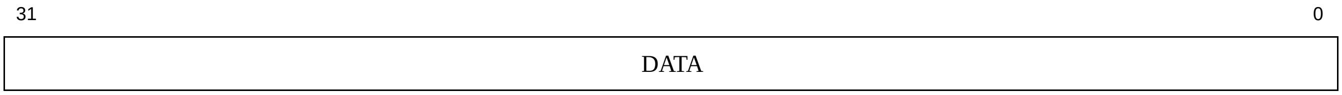


Table 2.47 IDataHi Register Field Description

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

2.3.6.5 Level 1 Data Cache Tag Low — DTagLo (CP0 Register 28, Select 2)

The *DTagLo* register acts as the interface to the data cache tag array. The Index Store Tag and Index Load Tag operations of the CACHE instruction use the *DTagLo* register as the source of tag information.

When the *WST* bit of the *ErrCtl* register is asserted, this register becomes the interface to the way-selection RAM. In this mode, the fields are redefined to give appropriate access the contents of the WS array instead of the Tag array.

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

The D-cache has five logical memory arrays associated with this *DTagLo* register. The tag RAM stores tags and other state bits with special attention to the needs of the CPU. The duplicate tag RAM also stores tags and state, but is optimized for the needs of interventions. Both of these arrays are set-associative (4-way). The Dirty RAM and duplicate Dirty RAM store the dirty bits (indicating modified data) for intervention uses, and each combine their ways together in a single entry per set. The WS RAM also combines the lock and LRU data in a single entry per set. Accessing these arrays for index cache loads and stores is controlled by using three bits in the *ErrCtl* register to create modes that allow the correct access to these arrays.

Note that the proAptiv core does not implement the *DTagHi* register.

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

- Default cache interface mode (*ErrCtl*$_{WST}$ = 0, *ErrCtl*$_{DYT}$ = 0, *ErrCtl*$_{SPR}$ = 0)
- Diagnostic "way select test mode" (*ErrCtl*$_{WST}$ = 1, *ErrCtl*$_{DYT}$ = 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)
- Diagnostic “duplicate tag array test mode” (*ErrCtl*$_{WST}$ = 1, *ErrCtl*$_{DYT}$ = 0, *ErrCtl*$_{SPR}$= 1)
- Diagnostic “duplicate dirty array test mode” (*ErrCtl*$_{WST}$ = 1, *ErrCtl*$_{DYT}$ = 1, *ErrCtl*$_{SPR}$= 1)

For all modes, the data RAM, tag RAM, WS RAM, and duplicate tag RAM are read. In addition, for duplicate tag array test mode, the duplicate tag RAM is also read, and for duplicate dirty array test mode, the duplicate Dirty RAM is read. Table 2.48 shows which RAMs are accessed for each mode for Loads and Stores.

Table 2.48 Summary of D-cache RAM accesses for Index Loads and Stores

Index Cacheop	Mode			RAM Being Accessed					
	WST	**DYT**	**SPR**	**Primary Tag RAM**	**WS RAM**	**Data RAM**	**Dirty RAM**	**Duplicate Tag RAM**	**Duplicate Dirty RAM**
Tag Store	0	0	0	WR	partial WR	RD	—	WR	—
Tag Load	0	0	0	RD	RD	RD	RD	—	—
Tag Store	1	0	0	—	partial WR	RD	—	—	—
Tag Load	1	0	0	RD	RD	RD	RD	—	—
Data Store	1	0	0	—	—	WR	—	—	—
Tag Store	0	1	0	—	—	RD	WR	—	WR
Tag Load	0	1	0	RD	RD	RD	RD	—	—
Tag Store	1	0	1	—	—	RD	—	WR	—
Tag Load	1	0	1	RD	RD	RD	RD	RD	—
Tag Store	1	1	1	—	—	RD	—	—	WR
Tag Load	1	1	1	RD	RD	RD	RD	—	RD

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. For stores in this mode, the tag RAM, WS RAM, and duplicate tag RAM are written. Also for stores, the *ErrCtl$_{PO}$* bit controls whether the tag RAM is written with P bit or with generated parity; the other RAMs written in this mode always use generated parity.

Figure 2.39 DTagLo Register Format (ErrCtl$_{WST}$ = 0, ErrCtl$_{DYT}$ = 0, ErrCtl$_{SPR}$ = 0)

31	12	11	10	9	8	7	6	5	4	1	0
PTagLo		VA11	U	0		V	E	L	0		P

Table 2.49 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 proAptiv Multiprocessing System 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
VA11	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

Table 2.49 Field Descriptions for DTagLo Register *(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
U	10	Unused	R/W	Undefined
0	9:8	Reserved. Write as zero. Ignored on reads.	R	0
V	7	Valid entry: This bit is set if this cache entry is valid (set zero to initialize the cache). Index Load: load from V field in primary tag RAM Index Store: store to V field in primary and duplicate tag RAM	R/W	Undefined
E	6	Exclusive entry: This bit is set if this cache entry is exclusive (set zero to initialize the cache). Index Load: load from E field in primary tag RAM Index Store: store to E field in primary tag RAM	R/W	Undefined
L	5	Locked entry: This bit is set 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. Index Load: load from appropriate way of L field in WS RAM Index Store: store to appropriate way of L and LP field in WS RAM, and if V is set, make selected way MRU in WS RAM; also, store to L field of duplicate tag RAM.	R/W	Undefined
0	4:1	Reserved. Write as zero. Ignored on reads.	R	0
P	0	Parity bit over the PTAG, E, and V bits of the cache tag entries Index Load: load from P field in primary tag RAM Index Store: possible write value for the P field of the primary tag RAM; write this bit if *ErrCtl.PO* = 1, else generate; parity written to other RAMs is generated.	R/W	Undefined

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. For stores in this mode, the WS RAM is written. Also for stores, the ErrCtl$_{PO}$ bit controls whether the WS RAM is written with LP bits or with generated parity; the other RAMs written in this mode always use generated parity. Also for stores, the LP and L fields only have the appropriate way written in the WS RAM. It is software's responsibility to maintain consistency with the value of the L field written into the duplicate tag RAM.

Figure 2.40 DTagLo Register Format (ErrCtl$_{WST}$ = 1, ErrCtl$_{DYT}$ = 0, ErrCtl$_{SPR}$ = 0)

31 24	23 20	19 16	15 10	9 8	7 5	4 1	0
U	LP	L	LRU	0	U	0	U

Table 2.50 Field Descriptions for DTagLo-WST Register

Name	Bit(s)	Description	Read/ Write	Reset State
U	31:24	Undefined.	R	0
LP	23:20	Parity for Cache-line locking control bits, held in the way select RAM. Each bit of this field is a parity bit for the corresponding bit in the L field. Index Load: load from LP field of WS RAM Index Store: store to appropriate way of LP field of WS RAM if $ErrCtl_{PO}$=1, else generate;	R/W	Undefined
L	19:16	Cache-line locking control bits, held in the way select RAM. Index Load: load from L field of WS RAM Index Store: store to appropriate way of L field of WS RAM.	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. Index Load: load from LRU field of WS RAM Index Store: store to LRU field of WS RAM	R/W	Undefined
0	9:8	Reserved. Write as zero. Ignored on reads.	R	0
U	7:5	Undefined.	R	0
0	4:1	Reserved. Write as zero. Ignored on reads.	R	0
U	0	Undefined.	R	0

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 can access either by **`cache`** load-tag/store-tag operations when $ErrCtl_{DYT}$ is set: then you get the data in these fields. For stores, the Dirty RAM is written. For stores, the Dirty RAM and duplicate Dirty RAM are written. Also for stores, the $ErrCtl_{PO}$ bit controls whether the Dirty RAM is written with DP bits or with generated parity; the other RAMs written in this mode always use generated parity.

Figure 2.41 Field Descriptions for DTagLo-DYT Register

31							24	23			20	19			16	15			12	11	10	9	8	7		5	4			1	0
U								DP				D				U				A		0		U			0				U

Table 2.51 Field Descriptions for DTagLo-DYT Register

Name	Bit(s)	Description	Read/ Write	Reset State
U	31:24	Undefined.	R	0
DP	23:20	Parity for Cache line "dirty" bits. Index Load: load from DP field of Dirty RAM Index Store: store to DP field of Dirty RAM if $ErrCtl_{PO}$=1, else generate;	R/W	Undefined

Table 2.51 Field Descriptions for DTagLo-DYT Register

Name	Bit(s)	Description	Read/ Write	Reset State
D	19:16	Cache line "dirty" bits. Index Load: load from D field of Dirty RAM Index Store: store to D field of Dirty RAM	R/W	Undefined
U	15:12	Undefined.	R	0
A	11:10	Cache line "alias" bits. Index Load: load from A field of Dirty RAM Index Store: store 0 and A[10] to A field of Dirty RAM	R/W	Undefined
0	9:8	Reserved. Write as zero. Ignored on reads.	R	0
U	7:5	Undefined.	R	0
0	4:1	Reserved. Write as zero. Ignored on reads.	R	0
U	0	Undefined.	R	0

If a *scratchpad* RAM has been implemented, it must be initialized and managed using `cache` load/store operations while *ErrCtl*$_{SPR}$ is set. The tag load/store operations are used to read and write control registers: During these operations, the DTagLo register has the following bit assignments.

Figure 2.42 DTagLo Register Format (ErrCtl$_{WST}$ = 0, ErrCtl$_{DYT}$ = 0, ErrCtl$_{SPR}$ = 1)

31 12	11 10	9 8	7	6 5	4 1	0
PTAG	U	0	E	U	0	U

Table 2.52 Field Descriptions for DTagLo-SPR Register

Name	Bit(s)	Description	Read/ Write	Reset State
PTAG	31:12	Scratchpad control. Sets base address.	R/W	Undefined
U	11:10	Undefined.	R	0
0	9:8	Reserved. Write as zero. Ignored on reads.	R	0
E	7	Scratchpad control enable.	R/W	Undefined
U	6:5	Undefined.	R	0
0	4:1	Reserved. Write as zero. Ignored on reads.	R	0
U	0	Undefined.	R	0

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

The duplicate tag RAM keeps tag state that is checked by interventions due to multiprocessing. In this mode, the duplicate tag RAM can be loaded and stored independently of the (primary) tag RAM. For stores in this mode, only the duplicate tag RAM is written.

Figure 2.43 DTagLo Register Format ($ErrCtl_{WST}$ = 1, $ErrCtl_{DYT}$ = 0, $ErrCtl_{SPR}$ = 1)

31	12	11	10	9	8	7	6	5	4	2	1	0
PTag		U		0		V	0	L	0		P1	P0

Table 2.53 Field Descriptions for DTagLo-DDTag Register

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Description</th><th>Read/ Write</th><th>Reset State</th></tr>
<tr><td>PTag</td><td>31:12</td><td>The cache address tag — a physical address because the proAptiv Multiprocessing System CPU’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.
Index Load: load from PTag filed in duplicate tag RAM
Index Store: store to PTag field in duplicate tag RAM</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>U</td><td>11:10</td><td>Unused</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>V, L</td><td>7, 5</td><td>For duplicate tag ram, these 2 bits encode the state of the cache line (set zero to initialize the cache).
Index Load: load from V and L field in duplicate tag RAM
Index Store: store to V and L field in duplicate tag RAM
<table><tr><th>V,L encoding</th><th>Meaning</th></tr><tr><td>00</td><td>Invalid</td></tr><tr><td>01</td><td>Shared</td></tr><tr><td>10</td><td>Exclusive (Modified if Duplicate Dirty is set)</td></tr><tr><td>11</td><td>Lock</td></tr></table></td><td>R/W</td><td>Undefined</td></tr>
<tr><td>0</td><td>4:2</td><td>Reserved. Write as zero. Ignored on reads.</td><td>R</td><td>0</td></tr>
<tr><td>P1</td><td>1</td><td>Parity bit over the L bit of the cache tag entries.
Index Load: load from P field in duplicate tag RAM.
Index Store: possible write value for the P field of the duplicate tag RAM; write this bit if $ErrCtl_{PO}$ = 1, else generate.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>P0</td><td>0</td><td>Parity bit over the all bits except the L bit of the cache tag entries.
Index Load: load from P field in duplicate tag RAM.
Index Store: possible write value for the P field of the duplicate tag RAM; write this bit if $ErrCtl_{PO}$ = 1, else generate.</td><td>R/W</td><td>Undefined</td></tr>
</table>

DTagLo-DDYT ($ErrCtl_{WST}$ = 1, $ErrCtl_{DYT}$ = 1, $ErrCtl_{SPR}$= 1)

The duplicate Dirty RAM keeps dirty state that is checked by interventions due to multiprocessing. In this mode, the duplicate Dirty RAM can be loaded and stored independently of the (primary) Dirty RAM. For stores in this mode, only the duplicate Dirty RAM is written.

Figure 2.44 Field Descriptions for DTagLo-DDYT Register

31	24	23	20	19	16	15	12	11	10	9	8	7	6	5	4	1	0
U		DP		D		U		A		0		U	0	U	0	U	U

Table 2.54 Field Descriptions for DTagLo-DDYT Register

Name	Bit(s)	Description	Read/ Write	Reset State
U	31:24	Unused	R/W	Undefined
DP	23:20	Parity for Cache line "dirty" bits. Index Load: load from DP field of duplicate Dirty RAM Index Store: store to DP field of duplicate Dirty RAM if $ErrCtl_{PO} = 1$, else generate	R/W	Undefined
D	19:16	Cache line "dirty" bits. Index Load: load from D field of duplicate Dirty RAM Index Store: store to D field of duplicate Dirty RAM	R/W	Undefined
U	15:12	Unused	R/W	Undefined
A	11:10	Cache line "alias" bits. Index Load: load zeroes Index Store: none	R/W	00
0	9:8	Reserved. Write as zero. Ignored on reads.	R	0
U	7	Undefined.	R	0
0	6	Reserved. Write as zero. Ignored on reads.	R	0
U	5	Undefined.	R	0
0	4:1	Reserved. Write as zero. Ignored on reads.	R	0
U	0	Undefined.	R	0

2.3.6.6 Level 1 Data Cache Data Low — DDataLo (CP0 Register 28, Select 3)

In the proAptiv core, 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.

The *DDataLo* register acts as the interface to the data cache data array and is intended for diagnostic operations only. The Index Load Tag operation of the CACHE instruction reads the corresponding data values into the *DDataLo* register. If the *WST* bit in the *ErrCtl* register is set, then the contents of *DDataLo* can be written to the cache data array by doing an Index Store Data CACHE instruction. If the *SPR* bit in the *ErrCtl* register is set, then the contents of *DDataLo* can be written to the scratchpad RAM data array by doing an Index Store Data CACHE instruction.

Figure 2.45 DDataLo Register Format

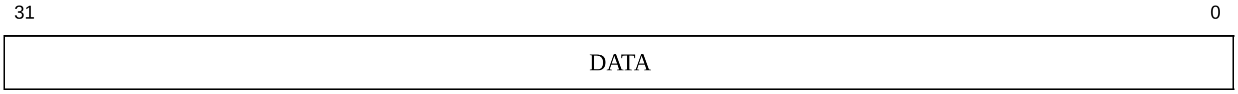


Table 2.55 DDataLo Register Field Description

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

2.3.6.7 Level 2/3 Cache Tag Low — L23TagLo (CP0 Register 28, Select 4)

The *L23TagLo* register acts as the interface to the L2 or L3 cache tag array. The L2 and L3 Index Store Tag and Index Load Tag operations of the CACHE instruction use the *L23TagLo* register as the source of tag information. Note that the proAptiv Multiprocessing System CPU does not implement the *L23TagHi* register.

The core can be configured without L2/L3 cache support. In this case, this register will be a read-only register that reads as 0.

Figure 2.46 L23TagLo Register Format

31	0
DATA	

2.3.6.8 Level 2/3 Cache Data Low — L23DataLo (CP0 Register 28, Select 5)

The *L23DataLo* register is a register that acts as the interface to the L2 or L3 cache data array and is intended for diagnostic operations only. The Index Load Tag operation of the CACHE instruction reads the corresponding data values into the *L23DataLo* register. If the *WST* bit in the *ErrCtl* register is set, then the contents of *L23DataLo* can be written to the cache data array by doing an Index Store Data CACHE instruction.

The core can be configured without L2/L3 cache support. In this case, this register will be a read-only register that reads as 0.

On proAptiv Multiprocessing System 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 2.47 L23DataLo Register Format

31	0
DATA	

Table 2.56 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

2.3.6.9 Level 2/3 Cache Data High — L23DataHi (CP0 Register 29, Select 5)

On proAptiv Multiprocessing System 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 2.48 L23DataHi Register Format

31	0
DATA	

Table 2.57 L23DataHi 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

2.3.6.10 ErrCtl (CP0 Register 26, Select 0)

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.

Figure 2.49 Error Control Register Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18 12	11 4	3 0
PE	PO	WST	SPR	PCO	0	LBE	WABE	L2EccEn	PCD	DYT	SE	FE	0	PI	PD

Table 2.58 Field Descriptions for ErrCtl Register

Name	Bit(s)	Description	Read/ Write	Reset State
PE	31	This bit is set to 1 to enable cache parity checking. Hard-wired to zero if parity is not 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*$_P$. 0 = User calculated parity 1 = Override calculated parity	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 implemented.	R/W	0

Table 2.58 Field Descriptions for ErrCtl Register *(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
PCO	27	Precode override. Used for diagnostic/test of the instruction 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 operations.	R/W	0
0	26	Reserved.	R	0
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/W0	Undefined
WABE	24		R/W0	Undefined
L2EccEn	23	L2 cache ECC enable. Indicates whether ECC is enabled on the L2Cache if present. If the L2 cache is not present, this bit has no meaning. 0: L2 cache present, L2 ECC disabled 1: L2 cache present, L2 ECC enabled	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
DYT	21	Setting this bit allows `cache` load/store data operations to work on the "dirty array" — the slice of cache memory which holds the "dirty"/ "stored-into" bits.	R/W	0
SE	20	Indicates that a second cache or TLB 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 or TLB error and therefore potentially recoverable. Error handling software should clear this bit when the error has been processed. This bit is set by hardware and cleared by software on reset. Refer to the *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
0	18:12	Reserved.	R	0
PI	11:4	Parity bits per double-word (two instructions) of data being read/written to the instruction cache data when the *PO* bit is set. During a read of IDataHi and IDataLo registers, the parity bits are stored here. This field is updated by hardware on every instruction fetch and also during a CacheOp store. During a CacheOp store, this field can be used for instruction cache data parity error injection apart from the Instruction cache store index. During a CacheOp read, this field can be used to check/read the instruction cache parity bits and also for storing the parity bits when an index load tag is executed.	R/W	0x00
PD	3:0	Parity bits being read/written to the data cache when *PO* is set.	R/W	0x0

2.3.6.11 Cache Error — CacheErr (CP0 Register 27, Select 0)

Read-only register used to analyze the details of a parity error. It may also be a good idea to have a separate table to indicate the valid bits for an FTLB parity error.

The FTLB parity error will set the EREC field to ‘b11, it will either set the ED or ET bits indicating a data or tag parity error (not both) and it will update the index and way fields. The other bits are left as 0. Note that the index field will contain the FTLB set and not the index value from the Index CP0 register.

I have also attached an email that indicates when the data and tag RAMs are checked for parity errors. Let’s talk if any of this is not clear.

Figure 2.50 CacheErr Register Format

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

Table 2.59 Field Descriptions for CacheErr Register

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Description</th><th>Read/ Write</th><th>Reset State</th></tr>
<tr><td><i>EREC</i></td><td>31:30</td><td>This 2-bit field indicates the block where the error occurred and is encoded as follows:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>00</td><td>L1 instruction cache error</td></tr>
<tr><td>01</td><td>External cache error</td></tr>
<tr><td>10</td><td>L1 data cache error</td></tr>
<tr><td>11</td><td>FTLB parity error</td></tr>
</table>
The FTLB parity error sets the <i>EREC</i> field to ‘b11, and sets either the <i>ED</i> or <i>ET</i> bits indicating a data or tag parity error (not both). It also updates the <i>Index</i> (bits 16:0) and <i>Way</i> (bits 21:19) fields. The other bits are left as 0. Note that the index field will contain the FTLB set and not the index value from the CP0 Index register.</td><td>R</td><td>Undefined</td></tr>
<tr><td><i>ED</i></td><td>29</td><td rowspan="2">The encoding of these two bits depends on the state of the EREC field above. If the state of this feld contains an encoding of 00, 01, or 10, indicating a cache error, the encoding of this field is as shown in the table below.
<b><i>ED and ET Bit Encoding on Cache Errors</i></b>
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>00</td><td>No tag or data RAM error detected</td></tr>
<tr><td>01</td><td>Primary tag RAM error</td></tr>
<tr><td>10</td><td>Data RAM error</td></tr>
<tr><td>11</td><td>Duplicate tag RAM error</td></tr>
</table>
If the state of the EREC feld contains an encoding of 11, indicating a TLB error, the encoding of this field is as follows.
A parity error in the FTLB tag sets the ET bit (28), while a parity error in the FTLB data sets the ED bit (29). One or both of these bits may be set.</td><td>R</td><td>Undefined</td></tr>
<tr><td><i>ET</i></td><td>28</td><td>R</td><td>Undefined</td></tr>
</table>

Table 2.59 Field Descriptions for CacheErr Register*(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
ES	27	Error source. In a multi-core system, this bit teads 0 if the error was caused by one of the cores and 1 if the error was caused by an external snoop request. In a single-core system, this bit is not supported.	R	Undefined
EE	26	Error external: In a multi-core system, this bit indicates that a parity error was seen on a coherent L1 cache in another CPU. In a single-core system, this bit is not supported.	R	Undefined
EB/EM	25	If *EC* equals 0 indicating an error in the L1 cache, this bit is *EB,* indicating an error in Both caches*.* If data and instruction-fetch errors 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 cache, this bit is *EM,* indicating there are errors in multiple locations in the cache.	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. However, if this bit is set, it indicates the error cannot be fixed. Here are some possible scenarios of when the EF bit might be set by hardware: • Dirty parity error in dirty line being displaced from cache • Line being displaced from cache has a tag parity error. • The line being displaced from cache 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. • Writeback store miss and *CacheErr*$_{EW}$ error. • At least one more cache error happened concurrently with or after this one, but before the original error reached the cache error exception handler. • If *EC* equals 0, and a second L2 error occurs 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. On a FTLB error, bits 20:19 indicate the number of ways in each set. Bit 21 is not used on a FTLB error.	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 2.59 Field Descriptions for CacheErr Register*(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
Index	16:0	The cache index or Scratchpad RAM index of the double word entry where the error occurred. The way of the faulty cache is written by hardware in the Way field. The CacheErr bits [16:0] represents the Address index bits [19:3]. 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. On a TLB error, this field indicates the number of sets in the FTLB. The number of bits is implementation dependent and is always right-justified in the Index field.	R	Undefined

2.3.7 Thead Context Registers

Although the proAptiv Multiprocessing System does not support thread contexts or shadow registers, the Shadow Register Set Control (SRSCtl) register is implemented to allow software to read this register to determine that shadow registers are not implemented.

2.3.7.1 SRSCtl Register (CP0 Register 12, Select 2)

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

Figure 2.51 SRSCtl Register Format

31 30	29 26	25 22	21 18	17 16	15 12	11 10	9 6	5 4	3 0
0	HSS	0	EICSS	0	ESS	0	PSS	0	CSS

Table 2.60 SRSCtl Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
HSS	29:26	Highest Shadow Set. This field contains the highest shadow set number that is implemented by this processor. A value of zero in this field indicates that only the normal GPRs are implemented. Possible values of this field for the proAptiv Multiprocessing System processor are: The value in this field also represents the highest value that can be written to the *ESS*, *EICSS*, *PSS*, and *CSS* fields of this register, or to any of the fields of the *SRSMap* register. The operation of the processor is **UNDEFINED** if a value larger than the one in this field is written to any of these other fieldsThis field is automatically updated when *SRSConf0* is written.	R	Preset

Table 2.60 SRSCtl Register Field Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Name	Bits			
EICSS	21:18	EIC interrupt mode shadow set. If $Config3_{VEIC}$ is 1 (EIC interrupt mode is enabled), this field is loaded from the external interrupt controller for each interrupt request and is used in place of the *SRSMap* register to select the current shadow set for the interrupt. If $Config3_{VEIC}$ is 0, this field returns zero on read.	R	Undefined
ESS	15:12	Exception Shadow Set. This field specifies the shadow set to use on entry to Kernel Mode caused by any exception other than a vectored interrupt. The operation of the processor is **UNDEFINED** if software writes a value into this field that is greater than the value in the *HSS* field.	R/W	0
PSS	9:6	Previous Shadow Set. If GPR shadow registers are implemented, and with the exclusions noted in the next paragraph, this field is copied from the CSS field when an exception or interrupt occurs. An ERET instruction copies this value back into the CSS field if $Status_{BEV}$ = 0. This field is not updated on any exception which sets $Status_{ERL}$ to 1 (i.e., Reset, Soft Reset, NMI, cache error), an entry into EJTAG Debug mode, or any exception or interrupt that occurs with $Status_{EXL}$ = 1, or $Status_{BEV}$ = 1. This field is not updated on an exception that occurs while $Status_{ERL}$ = 1. The operation of the processor is **UNDEFINED** if software writes a value into this field that is greater than the value in the *HSS* field.	R/W	0
CSS	3:0	Current Shadow Set. If GPR shadow registers are implemented, this field is the number of the current GPR set. With the exclusions noted in the next paragraph, this field is updated with a new value on any interrupt or exception, and restored from the PSS field on an ERET. Table 2.61 describes the various sources from which the CSS field is updated on an exception or interrupt. This field is not updated on any exception which sets *StatusERL* to 1 (i.e., Reset, Soft Reset, NMI, cache error), an entry into EJTAG Debug mode, or any exception or interrupt that occurs with $Status_{EXL}$ = 1, or $Status_{BEV}$ = 1. Neither is it updated on an ERET with $Status_{ERL}$ = 1 or $Status_{BEV}$ = 1. This field is not updated on an exception that occurs while $Status_{ERL}$ = 1. The value of CSS can be changed directly by software only by writing the PSS field and executing an ERET instruction.	R	0
0	31:30, 25:22, 17:16, 11:10, 5:4	Must be written as zeros; returns zero on read.	0	0

Table 2.61 Sources for new $SRSCtl_{CSS}$ on an Exception or Interrupt

Exception Type	Condition	$SRSCtl_{CSS}$ Source	Comment
Exception	All	$SRSCtl_{ESS}$	
Non-Vectored Interrupt	$Cause_{IV}$ = 0	$SRSCtl_{ESS}$	Treat as exception

Table 2.61 Sources for new $SRSCtl_{CSS}$ on an Exception or Interrupt *(continued)*

Exception Type	Condition	$SRSCtl_{CSS}$ Source	Comment
Vectored Interrupt	$Cause_{IV}$ = 1 and $Config3_{VEIC}$ = 0 and *Config3VInt* = 1	$SRSMap_{VECTNUM}$	Source is internal map register.
Vectored EIC Interrupt	$Cause_{IV}$ = 1 and $Config3_{VEIC}$ = 1	$SRSCtl_{EICSS}$	Source is external interrupt controller.

2.3.8 Performance Monitoring Registers

This section contains the following performance monitoring registers.

- Section 2.3.8.1, "Performance Counter Control 0 - 3 — PerfCtl0-3 (CP0 Register 25, Select 0, 2, 4, 6)" on page 149
- Section 2.3.8.2, "Performance Counter 0 - 3 — PerfCnt0-3 (CP0 Register 25, Select 1, 3, 5, 7)" on page 158

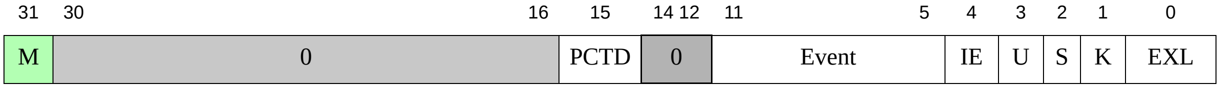
2.3.8.1 Performance Counter Control 0 - 3 — PerfCtl0-3 (CP0 Register 25, Select 0, 2, 4, 6)

Cores in the proAptiv Multiprocessing System family provide four performance counters that provide the capability to count events or cycles for use in performance analysis. Each performance counter consists of a pair of registers: a 32-bit control register (*PerfCtl*) and a 32-bit counter register (*PerfCnt*).

Performance counters can be configured to count implementation-dependent events or cycles under a specified set of conditions that are determined by the performance counter’s control register. The counter register increments once for each enabled event; when the most-significant bit of the counter register is a one (the counter overflows), and the counter is enabled, the performance counter optionally requests an interrupt.

The *IE* flag in the performance counter control register is used to enable an interrupt to be signalled when bit 31 of the corresponding counter overflows. The OR of all the performance counter register interrupts becomes the CPU output *SI_PCI*, which is typically fed back 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.

Figure 2.52 PerfCtl0-3 Register Format



31	30 16	15	14 12	11 5	4	3	2	1	0
M	0	PCTD	0	Event	IE	U	S	K	EXL

Table 2.62 Field Descriptions for PerfCtl0-3 Register

Name	Bit(s)	Description	Read/ Write	Reset State
M	31	Set to 1 if there is another *PerfCtl* register after this one. This field is set for *PerfCtl0-2* and cleared on *PerfCtl3.*	R	X
0	30:16	Reserved. Must be written as zeros; returns zeros on reads.	R	0
PCTD	15	Performance Counter Trace Disable. Setting this bit will prevent the tracing of data from this performance counter when performance counter trace mode in PDTrace is enabled.	R/W	Undefined
0	14:12	Reserved. Must be written as zeros; returns zeros on reads.	R	0
Event	11:5	Determines which event to count. Available events are listed in Table 2.63, "Performance Counter Events and Codes".	R/W	Undefined
IE	4	Set to cause an interrupt when the counter overflows into bit 31. 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. When this bit is set, events can be counted in User mode.	R/W	Undefined
S	2	Count events in Supervisor mode. When this bit is set, events can be counted in Supervisor mode.	R/W	Undefined
K	1	Count events in Kernel mode. When this bit is set, events can be counted in Kernel mode.	R/W	Undefined

Table 2.62 Field Descriptions for PerfCtl0-3 Register *(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
EXL	0	Count events in Exception mode. When this bit is set, events can be counted in Exception mode (when *StatusEXL* is set).	R/W	Undefined

Table 2.63 provides a list of performance counter events as encoded into the *Event* field in bits 11:5.

Table 2.63 Performance Counter Events and Codes

Event Number	Counter 0/2	Counter 1/3
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` Redirect Stall cycles due to: • Stalls due to register indirect jumps including non-predicted JR $31. • Stalls due to ERET, WAIT instructions. • Stalls due to IFU determined exception. 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 an MMU access. ITLB misses seen at the ID stage (this is the same for MMU instruction accesses). It is possible that a pending ITLB is killed before accessing the MMU.
5	Reserved	JTLB instruction access misses (will lead to an exception)
6	Instruction Cache accesses. proAptiv 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 proAptiv 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. I-cache miss stall cycles. This includes the cycles where the IFU state machine for a given TC is in the miss state. It is possible that multiple TCs requesting the same line will all count the same miss cycles.	Reserved
8	Uncached Instruction Fetch stall cycles. Cycles where no instruction is fetched because we are waiting for an I-fetch from uncached memory.	PDTrace back stalls

Table 2.63 Performance Counter Events and Codes *(continued)*

Event Number	Counter 0/2	Counter 1/3
9	Number of IFU fetch stalls due to lack of credits on the IBUF interface.	Valid fetch slots killed due to taken branches/jumps or stalling instructions.
10	Reserved in single-core environments In a multi-core environment, store misses transitioning to I->M or S->M	Reserved in single-core environments In a multi-core environment, load misses transitioning to I->S or I->E
11	Cycles IFU-IDU gate is closed due to mispredicted branch. This counts the time from when IEU closes the gate to when GRU opens.	Cycles IFU-IDU gate is open but no instructions fetched by IFU. May be overridden by changing *Config6.IFUPerfSel* field. See Table 2.10, "Field Descriptions for Config6 Register" for a description of the other overloading events.
12	Cycles IFU-IDU gate is closed due to other reasons: • MTC0/MFC0 sequence in pipe • EHB • DD_DR_DS is blocked	Reserved in single-core environments. In a multi-core environment, intervention hits.
13	Number of cycles where no instruction is inserted in DDQ0 because it is full.	Number of cycles where no instruction is inserted in DDQ1 because it is full.
14	Number of cycles where no instructions can be issued because there are no completion buffer ID’s.	Reserved.
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 - 17	Reserved	Reserved
18	Cycles when three instructions are issued.	Cycles when four instructions are issued.
19	Reserved	Reserved
20	Cycles when only one instruction is issued.	Cycles when two instructions are issued.
21	Number of `jr` (not `$31`) instructions mispredicted at graduation.	Number of `jr $31` instructions graduated.
22	Number of graduated JAR/JALR.HB	D-cache line refill (not LD/ST misses)
23	Counts the number of speculative loads. Pairs of loads or stores that are bonded count as one.	Speculative data cache accesses. Pairs of loads or stores that are bonded count as one.
24	Number of data cache misses at graduation.	D-cache misses. This count is per instruction at graduation and includes load, store, prefetch, `synci` and address based cacheops.
25	JTLB translation fails on d-side (data side as opposed to instruction side) accesses. This pertains to graduated instructions only.	Reserved
26	Load/store instruction redirects, which happen when the load/store follows too closely on a possibly matching cacheop. Load/Store generated replays - typically, a load following a CacheOp that has matches the Index match of the CacheOp.	Reserved

Table 2.63 Performance Counter Events and Codes *(continued)*

Event Number	Counter 0/2	Counter 1/3
27	LSGB graduation blocked cycles. Reasons for block: • CP1/2 store data not ready • SYNC, SYNCI at the head • **`sc`** at the head • CACHEOP at the head FSB, LDQ, WBB, or ITU FIFO full.	LSGB graduation that does not result in a request going out on the bus. Reasons include: • Misses at integer pipe graduation turn into hit. • Miss merges with outstanding fill request.
28	L2 cache writebacks	L2 cache accesses
29	L2 cache misses	L2 cache miss cycles
30	Cycles Fill Store Buffer (FSB) are full and cause a pipe stall	Cycles Fill Store Buffer (FSB) > 1/2 full
31	Cycles Load Data Queue (LDQ) are full and cause a pipe stall	Cycles Load Data Queue (LDQ) > 1/2 full
32	Cycles Writeback Buffer (WBB) are full and cause a pipe stall	Cycles Writeback Buffer (WBB) > 1/2 full
33	Reserved in single-core environments In a multi-core environment, counts requests that will receive data from the Coherence Manager.	Reserved in single-core environments. In a multi-core environment, request latency to first data word of data from the Coherence Manager.
34	Reserved in single-core environments In a multi-core environment, invalidate intervention hits.	Reserved in single-core environments. In a multi-core environment, all invalidate interventions.
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.
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. Bonded load/store counted as 2.	Stores graduated. Bonded load/store counted as 2.
42	**`j/jal`** graduated.	MIPS16e instructions graduated.
43	no-ops graduated - included **`sll, nop, ssnop,`** and **`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 in single-core environments. In a multi-core environment, writebacks due to evictions.	Reserved in single-core environments. In a multi-core environment, writebacks due to any reason.

Table 2.63 Performance Counter Events and Codes *(continued)*

Event Number	Counter 0/2	Counter 1/3
48	Reserved in single-core environments. In a multi-core environment, count of all invalidates (M,E,S)->I	Reserved in single-core environments. In a multi-core environment, count of transitions from (I,S)->E.
49	EJTAG instruction triggers.	EJTAG data triggers.
50	CP1 branches mispredicted.	Reserved
51	`sc` instructions graduated.	`sc` instructions failed.
52	`prefetch` instructions graduated at the top of LSGB.	`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	GFifo blocked cycles.	Floating point stores graduated.
56	GFifo blocked due to TLB or Cacheop.	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	Mispredicted branch instruction graduations without the delay slot (in the same cycle).	Cycles waiting for delay slot to graduate on a mispredicted branch.
58	Exceptions taken.	Replays initiated from graduation.
59	Implementation-specific CorExtend event. The integrator of the proAptiv core may connect the *UDI_perfcnt_event* pin to an event to be counted. This is intended for use with the CorExtend interface.	Reserved
60	Reserved in single-core environments. In a multi-core environment, state transition from S->M (coherent and non-coh).	Reserved in single-core environments. In a multi-core environment, state transitions from (M,E)->S.
61	Reserved in single-core environments. In a multi-core environment, request latency to self-intervention.	Reserved in single-core environments. In a multi-core environment, count of requests that will receive self-intervention.
62	Implementation-specific ISPRAM event.	Implementation-specific DSPRAM event. The integrator of the proAptiv core may connect the *SP_prf_c13_e62_xx* pin to the event to be counted.
63	L2 single-bit errors detected.	Reserved in single-core environments. In a multi-core environment, all interventions.
64	SI_Event[0] - Implementation-specific system event. The system integrator of the proAptiv core may connect the *SI_PCEvent[0]* pin to an event to be counted.	SI_Event[1] - Implementation-specific system event. The system integrator of the proAptiv core may connect the *SI_PCEvent[1]* pin to an event to be counted.

Table 2.63 Performance Counter Events and Codes *(continued)*

Event Number	Counter 0/2	Counter 1/3
65	SI_Event[2] - Implementation-specific system event. The system integrator of the proAptiv core may connect the *SI_PCEvent[2]* pin to an event to be counted.	SI_Event[3] - Implementation-specific system event. The system integrator of the proAptiv core may connect the *SI_PCEvent[3]* pin to an event to be counted.
66	SI_Event[4] - Implementation-specific system event. The system integrator of the proAptiv core may connect the *SI_PCEvent[4]* pin to an event to be counted.	SI_Event[5] - Implementation-specific system event. The system integrator of the proAptiv core may connect the *SI_PCEvent[5]* pin to an event to be counted.
67	SI_Event[7] - Implementation-specific system event. The system integrator of the proAptiv core may connect the *SI_PCEvent[7]* pin to an event to be counted.	SI_Event[8] - Implementation-specific system event. The system integrator of the proAptiv core may connect the *SI_PCEvent[8]* pin to an event to be counted.
68	All OCP requests accepted.	All OCP cacheable requests accepted.
69	OCP read requests accepted.	OCP cacheable read requests accepted.
70	OCP write requests accepted.	OCP cacheable write requests accepted.
71	Reserved	OCP write data sent.
72	Reserved	OCP read data received.
73	Reserved in single-core environments. In a multi-core environment, OCP Intervention write data stalled (valid but not accepted).	Reserved in single-core environments. In a multi-core environment, OCP Intervention write data valid (accepted or not).
74	Cycles Fill Store Buffer (FSB) < 1/4 full.	Cycles Fill Store Buffer (FSB) 1/4 to 1/2 full.
75	Cycles Load Data Queue (LDQ) < 1/4 full.	Cycles Load Data Queue (LDQ) 1/4 to 1/2 full.
76	Cycles Writeback Buffer (WBB) < 1/4 full.	Cycles Writeback Buffer (WBB) 1/4 to 1/2 full.
77	Counts the number of times that the L1 Branch Target Buffer (L1BTB) caused a redirect without IFU predecode-based prediction, causing a redirect or replay. Measues the number of true hits for the Return Prediction Stack (RPS) portion of the L1BTB.	Counts the number of times that the L1 Branch Target Buffer (L1BTB) caused a redirect without IFU predecode-based prediction causing a redirect or replay. Measues the number of true hits for the branch portion of the L1BTB.
78	Counts the number of times that the L1 Branch Target Buffer (L1BTB) caused a redirect with IFU predecode-based prediction causing a redirect or replay. Measues the number of mispredicts for the Return Prediction Stack (RPS) portion of the L1BTB.	Counts the number of times that the L1 Branch Target Buffer (L1BTB) caused a redirect with IFU predecode-based prediction causing a redirect or replay. Measues the number of mispredicts for the branch portion of the L1BTB.
79	Counts the number of writes to the Return Prediction Stack (RPS) portion of the L1 Branch Target Buffer (L1BTB) with no L1BTB hit (cold miss).	Counts the number of writes to the branch portion of the L1 Branch Target Buffer (L1BTB) with no L1BTB hit (cold miss).
80	Number of L1 Branch Target Buffer masked hits due to lack of credit for DS.	Number of L1 Branch Target Buffer masked hits due to lack of credit for target.
81	Number of NFW or L1 Branch Target Buffer mispredicts for instruction cache way-hit prediction.	Reserved
82 - 83	Reserved	Reserved

Table 2.63 Performance Counter Events and Codes *(continued)*

Event Number	Counter 0/2	Counter 1/3
84	Counts the number of times a Write-Back Buffer (WBB) entry is newly allocated for an Uncached Accelerated (UCA) store and there is one UCA store already active in the WBB.	Counts the number of times a Write-Back Buffer (WBB) entry is newly allocated for an Uncached Accelerated (UCA) store and there are two UCA stores already active in the WBB.
85	Number of times an uncached instruction arrives at BIU while there is an actively gathering UCA buffer.	Reserved
86	Reserved	Reserved
87	Number of stall cycles due to the lack of load/store queue (LSQ) ID.	Number of stall cycles due to the lack of IID.
88	Number of stall cycles due to the lack of DSP ID.	Reserved.
89	Number of cycles when no FP instructions are dispatched.	Number of cycles when no integer instructions are dispatched.
90	Number of cycles when one FP instruction is dispatched.	Number of cycles when one integer instruction is dispatched.
91	Number of cycles when two FP instructions are dispatched.	Number of cycles when two integer instructions are dispatched.
92 - 93	Reserved	Reserved
94	Number of cycles when three instructions are issued.	Number of cycles when four instructions are issued.
95 - 96	Reserved	Reserved
97	Number of instructions issued on AGU port from DDQ1.	Number of instructions issued on BSU port from DDQ1.
98	Number of instructions issued on MDU/ALU2 port from DDQ1.	Number of instructions issued on ALU1 port from DDQ0.
99	Number of DTLB accesses (speculative).	Number of DTLB misses (speculative).
100	Data side hits in the VTLB/FTLB. This includes FTLB and VTLB hits and unmapped region accesses.	Instruction side hits in the VTLB/FTLB. This includes FTLB and VTLB hits and unmapped region accesses.
101	Number of data side hits in the VTLB/FTLB in an unmapped region.	Number of instruction side hits in the VTLB/FTLB in an unmapped region.
102 - 104	Reserved	Reserved
105	Number of DTLB hits to the half of *EntryLo* that caused a fill (speculative).	Number of DTLB hits to the half of *EntryLo* that did not cause a fill (speculative).
106	Number of pairs of bonded stores at graduation.	Number of pairs of bonded loads at graduation.
107	Reserved	Speculative count of ‘over-eager’ loads that hit a store without the data being available.
108	Number of times a load is not issued because it is tagged by the ‘over-eager’ predictor.	Reserved
109	Speculative count of incorrectly bonded loads and stores.	Reserved
110	Number of misaligned loads that graduated.	Number of misaligned stores that graduated.

Table 2.63 Performance Counter Events and Codes *(continued)*

Event Number	Counter 0/2	Counter 1/3
FPU Performance Counters		
112	Counts the number of cycles that the arithmetic channel is full and signalling busy to the integer core.	Counts the number of cycles the to-from channel is full and signalling busy to the integer core.
113	Counts the total number of arithmetic instructions issued.	Counts the total number of to-from instructions issued.
114	Counts the total number of add/multiply class instructions (add, sub, cvt, ceil, floor, round, trunc, mul).	Counts the total number of arithmetic multiply-add instructions (madd, msub, nmadd, nmsub).
115	Counts the total number of arithmetic iteration class instructions (div, recip, sqrt, rsqrt).	Counts the total number of arithmetic compare class instructions (C.cond).
116	Counts the total number of arithmetic miscellaneous class instructions (abs, neg, move, bra).	Counts the total number of data stall retires due to an scheduled load queue preempt write.
117	The retire stage is stalled if there is an older atirhmetic to-from instruction in the other channel with the same FD, or if there is an older, unissued arithmetic to-from instruction in the other channel.	Counts the total number of data channel conflict stalls.
118	Counts the total number of arithmetic channel kill received stalls. This retire stall occurs if the instruction has not received a kill strobe.	Counts the total number of data channel kill received stalls. Same as arithmetic condition.
119	Counts the total number of arithmetic channel result valid stalls. This retire stall occurs if the result is not yet available from the APU.	Counts the total number of data channel due to no room in the Scheduled Load Queue.
120	Counts the total number of arithmetic channel instruction issue stalls. This retire stall occurs if the instruction has not yet been issued.	Counts the total number of data channel instruction issue stalls.
121	Counts the total number of arithmetic channel retire stall cycles. This is the sum of all of the retire stall conditions on counters 0/2 as described in events 116 - 120.	Counts the total number of data channel retire stall cycles. This is the sum of all of the retire stall conditions on counters 1/3 as described in events 116 - 120.
122	Counts the total number of arithmetic channel indeterminate dependency or format mismatch stalls. A stall occurs when: a. The youngest instruction is unknown due to taking an additional clock cycle to determine which of many instructions is the youngest. b. The youngest instruction is unknown because it has not yet received a null strobe. c. The youngest dependent instruction has a format mismatch that precludes the bypassing of data.	Counts the total number of data channel indeterminate dependency or format mismatch stalls. Stall conditions for the data channel are the same as for arithmetic channel.
123	Counts the total number of arithmetic channel APU stalls. This type of stall occurs when the APU is unable to take this class of instruction.	Reserved.

Table 2.63 Performance Counter Events and Codes *(continued)*

Event Number	Counter 0/2	Counter 1/3
124	Counts the total number of arithmetic channel arithmetic data stalls. This type of stall occurs when the arithmetic unit is waiting for data from an arithmetic instruction.	Counts the total number of data channel arithmetic data stalls.
125	Counts the total number of arithmetic channel to-data stalls. This type of stall occurs when the arithmetic unit is waiting for data from a to-from instruction.	Counts the total number of data channel to-data stalls.
126	Counts the number of arithmetic channel stalls due to a pipecleaner instruction. Either the instruction is a pipecleaner and is stalling while waiting for all older instructions to retire, or an instruction is stalled while waiting for an older pipecleaner instruction to retire.	Counts the number of data channel stalls due to a pipecleaner instruction. Either the instruction is a pipecleaner and is stalling while waiting for all older instructions to retire, or an instruction is stalled while waiting for an older pipecleaner instruction to retire.
127	Counts the number of all arithmetic channel issue stall cycles. This is the sum of all of the above arithmetic channel issue stall conditions.	Counts the number of all data channel issue stall cycles. This is the sum of all of the above data channel issue stall conditions.

2.3.8.2 Performance Counter 0 - 3 — PerfCnt0-3 (CP0 Register 25, Select 1, 3, 5, 7)

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

Figure 2.53 Performance Counter 0 - 3 Register

31 0
Counter

Table 2.64 Performance Counter 0 - 3 Register Field Descriptions

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

2.3.9 Debug Registers

This section contains the following debug registers.

- Section 2.3.9.1, "Debug (CP0 Register 23, Select 0)" on page 158
- Section 2.3.9.2, "Debug Exception Program Counter — DEPC (CP0 Register 24, Select 0)" on page 161
- Section 2.3.9.3, "Debug Save — DESAVE (CP0 Register 31, Select 0)" on page 162
- Section 2.3.9.4, "Watch Low 0 - 3 — WatchLo0-3 (CP0 Register 18, Select 0-3)" on page 162
- Section 2.3.9.5, "Watch High 0 - 3 — WatchHi0-3 (CP0 Register 19, Select 0-3)" on page 163

2.3.9.1 Debug (CP0 Register 23, Select 0)

The *Debug* register provides control and status information while in debug mode. During normal operation (non-debug mode), this register may not be written at all, and only the *DM* bit and the *EJTAGver* field returns valid data.

The read-only bits are updated by hardware 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.

Figure 2.54 Debug Register Format

31	30	29	28	27	26	25	24	23	22	21	20
DBD	DM	NoDCR	LSNM	Doze	Halt	CountDM	IBusEP	MCheckP	CacheEP	DBusEP	IEXI

19	18	17 15	14 10	9	8	7 6	5	4	3	2	1	0
DDBSImpr	DDBLImpr	EJTAGver	DExcCode	NoSSt	SSt	0	DINT	DIB	DDBS	DDBL	DBp	DSS

Table 2.65 Field Descriptions for Debug Register

Name	Bit(s)	Description	Read/ Write	Reset State
DBD	31	Indicates if the last debug exception or exception in debug mode occurred in a branch delay slot. 0: Not in delay slot 1: In delay slot When set to 1, the Debug Exception Program Counter (*DEPC)* points to the branch instruction, which is usually the correct place to restart.	R	Undefined
DM	30	Indicates if the processor is operating in debug mode. 0: Processor is operating in non-debug mode 1: Processor is operating in debug mode In debug mode, this bit is set on any debug exception and is cleared by **deret**.	R	0
NoDCR	29	Indicates if the dseg memory segment and a memory-mapped *DCR* register is present. 0: dseg address space is present 1: dseg address space is not present	R	0
LSNM	28	Controls access of load/store between dseg and main memory. 0: Load/stores in dseg address range goes to dseg 1: Load/stores in dseg address range goes to main memory Setting this bit causes debug-mode accesses to dseg addresses to be sent to system memory. This makes most of the EJTAG unit's control systems unavailable, so will probably only be done around a particular load/store.	R/W	0
Doze	27	Indicates that the processor was in any kind of low power mode when a debug exception occurred. 0: Processor not in low power mode when debug exception occurred 1: Processor in low power mode when debug exception occurred Before the debug exception, CPU was in one of the reduced power mode.	R	Undefined
Halt	26	Indicates that the internal system bus clock was stopped when the debug exception occurred. 0: Internal system bus clock running 1: Internal system bus clock stopped Before the debug exception, the CPU was stopped — probably asleep following a **wait** instruction.	R	Undefined

Table 2.65 Field Descriptions for Debug Register *(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
CountDM	25	Controls or indicates the Count register behavior in debug mode. 0: Count register stopped in debug mode 1: Count register is running in debug mode	R/W	1
IBusEP	24	These "pending exception" flags remember exception events caused by instructions run in debug mode, but which have not yet occurred because they are imprecise and $Debug_{IEXI}$ is set. Note that you can write a 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 the **`deret`** clears the bit, the exception is taken and the pending bit cleared. *IBusEP* remembers a bus error on an instruction fetch. This exception is precise, so it cannot occur and the field is always zero. *MCheckP* machine check condition (usually an illegal TLB update). The machine check can be either precise or imprecise dependgin on the type of error. Refer to the Machine Check exception in the Exception chapter for more information. *CacheEP* indicates a precise cache parity error is pending. Data access Bus Error exception Pending: *DBusEP* remembers a bus error on a data access. Set when an data bus error event occurs or if a 1 is written to the bit by software. Cleared when a Data Bus Error exception is taken by the processor, and by reset. If *DBusEP* is set when *IEXI* is cleared, a Data Bus Error exception is taken by the processor, and *DBusEP* is cleared	R/W	0
MCheckP	23		R/W	0
CacheEP	22		R/W	0
DBusEP	21		R/W	0
IEXI	20	Imprecise Error eXception Inhibit. Set to 1 to defer imprecise exceptions. By default, this bit is set on entry to debug mode and cleared on exit. The deferred exception returns when and if this bit is cleared, and until then the occurrence of the imprecise exception can be observed in the “pending exception” flags described in bits 24:21 above.	R/W	0
DDBSImpr	19	Imprecise store breakpoint. *DEPC* probably points to an instruction some time later in the sequence than the store which triggered the breakpoint.	R	0
DDBLImpr	18	Imprecise load breakpoint. *DEPC* probably points to an instruction some time later in the sequence than the store which triggered the breakpoint. The debugger or user (or both) have to cope as best they can.	R	0
EJTAGver	17:15	These read-only bits encode the revision of the EJTAG specification to which this implementation conforms. The legal values are. 101: Version 5.0 All other values are reserved.	R	5
DExcCode	14:10	Indicates the cause of the latest exception in debug mode. Following initial entry to debug mode, this field is undefined. The subsequent value will be one of those defined in $Cause_{ExcCode}$. See Table 2.37 for a list of values. Value is undefined after a debug exception.	R	Undefined

Table 2.65 Field Descriptions for Debug Register *(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
NoSSt	9	Indicates whether the single-step feature controllable by the *SSt* bit is available in this implementation. This read-only bit is always zero on the proAptiv core because single-step is implemented.	R	0
SSt	8	Controls if debug single step exception is enabled. 0 = No debug single-step exception enabled 1 = Debug single-step exception enabled	R/W	0
0	7:6	Reserved. Must be written as zeros; returns zeros on reads.	R	0
Offline	7	Implemented per-TC. When this bit is 1, TC is allowed to execute only in Debug mode.	R/W	0
R	6	Reserved. Must be written as zeros; returns zeros on reads.	R	0
DINT	5	Indicates that a debug interrupt exception (from EJTAG pin) occurred. Cleared on exception in debug mode. 0: No debug interrupt exception 1: Debug interrupt exception	R	Undefined
DIB	4	Instruction breakpoint. This bit is set by hardware when an instruction breakpoint occurs. 0: No debug exception breakpoint 1: Debug exception breakpoint occurred	R	Undefined
DDBS	3	Indicates that a debug data break exception occurred on a store. Cleared on exception in debug mode. 0: No debug data exception on a store 1: Debug instruction exception on a store	R	Undefined
DDBL	2	Indicates that a debug data break exception occurred on a load. Cleared on exception in debug mode. 0: No debug data exception on a load 1: Debug instruction exception on a load	R	Undefined
DBp	1	Indicates that a debug software breakpoint exception occurred. Cleared on exception in debug mode. 0: No debug software breakpoint exception 1: Debug software breakpoint exception	R	Undefined
DSS	0	Indicates that a debug single-step exception occurred. Cleared on exception in debug mode. 0: No debug single-step exception 1: Debug single-step exception	R	Undefined

2.3.9.2 Debug Exception Program Counter — DEPC (CP0 Register 24, Select 0)

The Debug Exception Program Counter (DEPC) points to the instruction to restart when a **`deret`** is executed to exit 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 2.55 DEPC Register Format

31	0
DEPC	

Table 2.66 DEPC Register Formats

Field				
Name	**Bit(s)**	**Description**	**Read / Write**	**Reset**
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

2.3.9.3 Debug Save — DESAVE (CP0 Register 31, Select 0)

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 2.56 DeSave Register Format

31	0
DESAVE	

Table 2.67 DeSave Register Field Description

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

2.3.9.4 Watch Low 0 - 3 — WatchLo0-3 (CP0 Register 18, Select 0-3)

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. The bit assignments for each of the *WatchLo* registers is identical. Hence, only one register is shown below.

Figure 2.57 WatchLo0-3 Register Format

31 3	2	1	0
VAddr	I	R	W

Table 2.68 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 = Instruction fetches
		R = Reads (loads)
		W = 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

2.3.9.5 Watch High 0 - 3 — WatchHi0-3 (CP0 Register 19, Select 0-3)

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 2.58 WatchHi0-3 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 2.69 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. Software can use these four bits (starting with *WatchHi0*) to determine how many watchpoints there are. This field is set for *WatchHi0-2* and cleared on *WatchHi3.*	R	1 (*WatchHi0-2*) 0 (*WatchHi3*)
G	30	*WatchHi0-3*$_{ASID}$ matches addresses from a particular address space (the "ASID" is like that in TLB entries) — except that the *WatchHi0-3*$_G$ ("global") can be set to match the address in any address space. If the *WatchHi0-3*$_G$ bit is set, the address is always matched, regardless of the ASID value.	R/W	Undefined
0	29:24	Reserved. Write as zero. Ignored on reads.	R	0

Table 2.69 Field Descriptions for WatchHi0-3 Register*(continued)*

Name	Bit(s)	Description	Read/ Write	Reset State
ASID	23:16	*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. The match a particular address, the *WatchHi0-3*$_{G}$ bit is cleared and the *WatchHi0-3*$_{ASID}$ value is used to ensure that the match is to the correct address space. If the If the *WatchHi0-3*$_{G}$ bit is set, the address is always matched, regardless of the *WatchHi0-3*$_{ASID}$ value.	R/W	Undefined
0	15:12	Reserved. Write as zero. Ignored on reads.	R	0
Mask	11:3	Watch mask. This field marks the corresponding *WatchLo0-3*$_{VAddr}$ address bits to be ignored when deciding whether this is a match.	R/W	Undefined

I	2	Watch exception type. These bits indicate what type of access (if any) matched after a watch exception.
		I = Instruction fetches
		R = Reads (loads)
		W = Writes (stores)
		Write a 1 to any of these bits in order to *clear* it (and therefore prevent the exception from immediately happening again). This behavior 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

2.3.10 PDTrace Registers

This section contains the following MIPS PDTrace registers.

- Section 2.3.10.1, "Trace Control Register — TraceControl (CP0 Register 23, Select 1)" on page 164
- Section 2.3.10.2, "Trace Control 2 Register — TraceControl2 (CP0 Register 23, Select 2)" on page 167
- Section 2.3.10.3, "Trace Control 3 Register — TraceControl3 (CP0 Register 24, Select 2)" on page 169
- Section 2.3.10.4, "User Trace Data 1 Register — UserTraceData1 (CP0 Register 23, Select 3)" on page 170
- Section 2.3.10.5, "User Trace Data 2 Register — UserDataTrace2 (CP0 Register 24, Select 3)" on page 170
- Section 2.3.10.6, "Trace Instruction Breakpoint Condition Register — TraceIBPC (CP0 Register 23, Select 4)" on page 170
- Section 2.3.10.7, "Trace Data Breakpoint Condition Register — TraceDBPC (CP0 Register 23, Select 5)" on page 171

2.3.10.1 Trace Control Register — TraceControl (CP0 Register 23, Select 1)

The *TraceControl* register configuration is shown below.

Figure 2.59 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 2.70 TraceControl Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
TS	31	The trace select bit is used to select between the hardware and the software trace control bits. A value of zero selects the external hardware trace block signals, and a value of one selects the trace control bits in the *TraceControl* register.	R/W	0
UT	30	This bit is 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. The actual triggering of a user trace record happens on a write to the *UserTraceData* register.	0	Undefined
0	29	Reserved. Must be written as zero; returns zero on read.	0	0
Ineff	28	When set to 1, 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 (Instruction Completion Indicator). 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 proAptiv Multiprocessing System trace logic that slow but complete tracing is desired. Hence, the proAptiv Multiprocessing System 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	Debug mode. When set to one, this enables tracing in debug mode. For a trace to be enabled in Debug mode, the *On* bit must also be set, and either the *G* bit must be set, or the current process ASID must match the *ASID* field in this register. When set to zero, trace is disabled in debug mode.	R/W	Undefined
E	24	Exception mode. When set to one, tracing is enabled in Exception mode. For a trace to be enabled in Exception mode, the *On* bit must be set, and either the *G* bit must be set, or the current process ASID must match the *ASID* field in this register. When set to zero, trace is disabled in Exception Mode.	R/W	Undefined
K	23	Kernel mode. When set to one, enables tracing in Kernel mode. For a trace to be enabled in Kernel mode, the *On* bit must be set, and either the *G* bit must be set, or the current process ASID must match the *ASID* field in this register. When set to zero, trace is disabled in Kernel Mode.	R/W	Undefined

Table 2.70 TraceControl Register Field Descriptions*(continued)*

Fields		Description	Read / Write	Reset State
Name	Bits			
S	22	Supervisor mode. When set to one, tracing is enabled in Supervisor Mode. For a trace to be enabled in Supervisor mode, the On bit must be set, and either the *G* bit must be set, 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
U	21	User mode. When set to one, tracing is enabled in User mode. For a trace to be enabled in User mode, the *On* bit must be set, and either the *G* bit must be set, 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	ASID mask. 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.	R/W	Undefined
ASID	12:5	Address space identifier. This field stores 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	Global enable. When set, 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, 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	Load/Store Miss trace. 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	Trace IM bit. 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

2.3.10.2 Trace Control 2 Register — TraceControl2 (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). 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 2.60 TraceControl2 Register Format

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

Table 2.71 TraceControl2 Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td><i>SyPExt</i></td><td>31:30</td><td>Sync period extension. Extension to the <i>SyP</i> (sync period) field for implementations that need higher numbers of cycles between synchronization events.
The value of <i>SyP</i> is extended by assuming that these two bits are juxtaposed to the left of the three bits of <i>SyP</i> (<i>SypExt</i>$_{SyP}$). When only <i>SyP</i> was used to specify the synchronization period, the value was 2x, where x was computed from <i>SyP</i> 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 <i>SyPExt</i> and <i>SyP</i>. 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.</td><td>R/W</td><td>0</td></tr>
<tr><td><i>CPUIdV</i></td><td>29</td><td>When set, this bit specifies that the CPU defined in <i>CPUId</i> must be traced. Otherwise, instructions from all CPUs are traced when other conditions for tracing are valid. This bit is ignored if <i>TCV</i> is asserted.</td><td>R/W</td><td>0</td></tr>
<tr><td><i>CPUId</i></td><td>28:21</td><td>This field specifies the number of the CPU to trace when <i>CPUIdV</i> is set.</td><td>R/W</td><td>0</td></tr>
<tr><td>R</td><td>20:12</td><td>Reserved. Write as zero. Ignored on reads.</td><td>R</td><td>0</td></tr>
<tr><td><i>Mode</i></td><td>11:7</td><td>When tracing is turned on, these five bits specify what information is to be traced by the core. 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 corresponding trace (shown in the table below) is not traced by the processor.
<table>
<tr><th>Bit</th><th>Trace the Following</th></tr>
<tr><td>7</td><td>PC</td></tr>
<tr><td>8</td><td>Load address</td></tr>
<tr><td>9</td><td>Store address</td></tr>
<tr><td>10</td><td>Load data</td></tr>
<tr><td>11</td><td>Store data</td></tr>
</table></td><td>R/W</td><td>Undefined</td></tr>
</table>

Table 2.71 TraceControl2 Register Field Descriptions*(continued)*

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td><i>ValidModes</i></td><td>6:5</td><td>This field specifies the subset of tracing that is supported by the processor.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>00</td><td>PC tracing only</td></tr>
<tr><td>01</td><td>PC and load and store address tracing only</td></tr>
<tr><td>10</td><td>PC, load and store address, and load and store data</td></tr>
<tr><td>11</td><td>Reserved</td></tr>
</table></td><td>R</td><td>Preset</td></tr>
<tr><td><i>TBI</i></td><td>4</td><td>This bit indicates how many trace buffers are implemented by the TCB, as follows.
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.</td><td>R</td><td>Undefined</td></tr>
<tr><td><i>TBU</i></td><td>3</td><td>This bit denotes to which trace buffer the trace is currently being written and is used to select the appropriate interpretation of the <i>TraceControl2</i>$_{SyP}$ field.
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 <i>TCBCONTROLB</i>$_{\text{OfC}}$.</td><td>R</td><td>Undefined</td></tr>
<tr><td><i>SyP</i></td><td>2:0</td><td>The period (in cycles) to which the internal synchronization counter is reset when tracing is started, or when the synchronization counter has overflowed.
<table>
<tr><th>SyP</th><th>Sync Period</th></tr>
<tr><td>000</td><td>2^5</td></tr>
<tr><td>001</td><td>2^6</td></tr>
<tr><td>010</td><td>2^7</td></tr>
<tr><td>011</td><td>2^8</td></tr>
<tr><td>100</td><td>2^9</td></tr>
<tr><td>101</td><td>2^{10}</td></tr>
<tr><td>110</td><td>2^{11}</td></tr>
<tr><td>111</td><td>2^{12}</td></tr>
</table>
This field is loaded from <i>TCBCONTROLA</i>$_{\text{SyP}}$.</td><td>R</td><td>Undefined</td></tr>
</table>

2.3.10.3 Trace Control 3 Register — TraceControl3 (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). 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 PDtrace capability is present.

Figure 2.61 TraceControl3 Register Format

31 14	13	12	11	10	9	8	7 3	3	1	0
0	PeCOvf	PeCFCR	PeCBP	PeCSync	PeCE	PeC	0	TRIDLE	TRPAD	FDT

Table 2.72 TraceControl3 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:14	Reserved. Must be written as zeros; returns zeros on reads.	R	0
PeCOvf	13	Performance counter overflow. Setting this bit enables the trace control logic to trace a performance counter overflow.	R/W	0
PeCFCR	12	Performance counter function/call return. Setting this bit enables the trace control logic to trace a function call/return condition or an exception handler entry.	R/W	0
PeCBP	11	Performance counter hardware breakpoint. Setting this bit enables the trace control logic to trace a hardware breakpoint condition.	R/W	0
PeCSync	10	Performance counter synchronization counter expiration. Setting this bit enables the trace control logic to trace a synchronization counter expiration condition.	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
0	7:3	Reserved. Must be written as zeros; returns zeros on reads.	R	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. 0: Filtered data trace mode is disabled. 1: Filtered data trace mode is enabled.	R/W	0

2.3.10.4 User Trace Data 1 Register — UserTraceData1 (CP0 Register 23, Select 3)

A software write to any bits in the *UserTraceData1* register triggers a trace record to be written with a type indicator TU1.

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

Figure 2.62 User Trace Data 1 Register Format

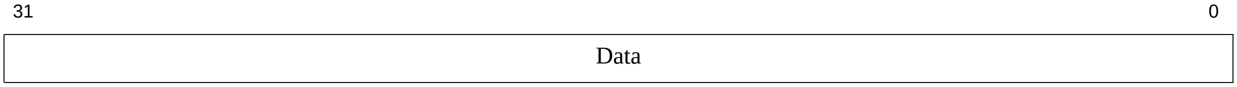


Table 2.73 User Trace Data 1 Register Field Descriptions

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

2.3.10.5 User Trace Data 2 Register — UserDataTrace2 (CP0 Register 24, Select 3)

A software write to any bits in the *UserTraceData2* register triggers a trace record to be written with a type indicator TU2.

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

Figure 2.63 User Trace Data 2 Register Format

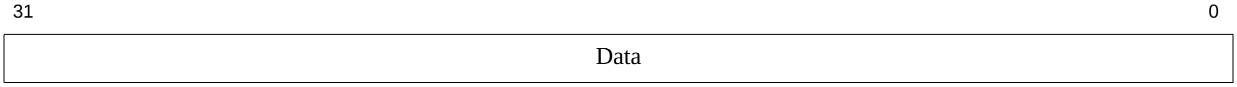


Table 2.74 User Trace Data 2 Register Field Descriptions

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

2.3.10.6 Trace Instruction Breakpoint Condition Register — TraceIBPC (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 2.64 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 2.75 TraceIBPC Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:30	Reserved. Must be written as zeros; returns zeros on reads.	R	0
PCT	29	Used to specify whether a performance counter trigger signal is generated when an EJTAG instruction breakpoint match occurs. 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 or not the trigger signal from EJTAG instruction breakpoint should trigger tracing functions. 0: Disables trigger signals from instruction breakpoints 1: Enables trigger signals from instruction breakpoints	R/W	0
0	27:12	Reserved. Must be written as zeros; returns zeros on reads.	R	0
IBPC3 *IBPC2* *IBPC1* *IBPC0*	11:9 9:6 5:3 2:0	The four 3-bit fields are decoded to enable different tracing modes. Table 2.77 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

2.3.10.7 Trace Data Breakpoint Condition Register — TraceDBPC (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 2.65 TraceDBPC Register Format

31 30	29	28	27 6	5 3	2 0
0	PCT	DE	0	$DBPC_1$	$DBPC_0$

Table 2.76 TraceDBPC Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:30	Reserved. Must be written as zeros; returns zeros on reads.	R	0
PCT	29	Used to specify whether a performance counter trigger signal is generated when an EJTAG data breakpoint match occurs. 0: Disables performance counter trigger signal from data breakpoints 1: Enables performance trigger signals from data breakpoints	R/W	0

Table 2.76 TraceDBPC Register Field Descriptions*(continued)*

Fields		Description	Read / Write	Reset State
Name	Bits			
DE	28	Used to specify whether the trigger signal from EJTAG data breakpoint should trigger tracing functions. 0: Disables trigger signals from data breakpoints 1: Enables trigger signals from data breakpoints	R/W	0
0	27:26	Reserved. Must be written as zeros; returns zeros on reads.	R	0
DBPC0 *DBPC1*	2:0 5:3	The two 3-bit fields are decoded to enable different tracing modes. Table 2.77 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 2.77 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 implementations.
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.

2.3.11 User Mode Support Registers

This section contains the following hardware access registers.

- Section 2.3.11.1, "Hardware Enable — HWREna (CP0 Register 7, Select 0)" on page 172
- Section 2.3.11.2, "UserLocal (CP0 Register 4, Select 2)" on page 174

2.3.11.1 Hardware Enable — HWREna (CP0 Register 7, Select 0)

The *HWREna* register contains a bit mask that determines which hardware registers are accessible via the **`rdhwr`** instruction when that instruction is executed in user mode.

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

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

Software may determine which registers are implemented by writing all ones to the *HWREna* register, then reading the value back. If a bit reads back as a one, the processor implements that hardware register.

Figure 2.66 HWREna Register Format

31 30	29	28 4	3	2	1	0
Impl	UL	0	CCRes	CC	SYNCI_Step	CPUNum

Table 2.78 Field Descriptions for HWREna Register

<table>
<tr><th>Name</th><th>Bit(s)</th><th>Description</th><th>Read/
Write</th><th>Reset State</th></tr>
<tr><td><i>Impl</i></td><td>31:30</td><td>These bits control access to implementation-dependent hardware registers. These registers are not currently implemented in any proAptiv Multiprocessing System family processor. Attempts to access these bits results in a Reserved Instruction Exception.</td><td>R</td><td>0</td></tr>
<tr><td><i>UL</i></td><td>29</td><td><i>UserLocal</i> register present. This register provides read access to the coprocessor 0 <i>UserLocal</i> register. Set this bit to 1 to permit user programs to obtain the value of the <i>UserLocal</i> CP0 register using <b>rdhwr 29.</b></td><td>R/W</td><td>0</td></tr>
<tr><td>0</td><td>28:4</td><td>Ignored on write; returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td><i>CCRes</i></td><td>3</td><td>Resolution of the <i>Count</i> register. This value denotes the number of cycles between updates of the <i>Count</i> register. Setting this bit allows selected instructions to read the <i>Count</i> register. For example, if this bit is set, the execution of a user-mode <b>rdhwr 3</b> instruction read the interval at which the <i>Count</i> register increments.
<table>
<tr><th>CCRes Value</th><th>Meaning</th></tr>
<tr><td>1</td><td><i>Count</i> register increments every cycle</td></tr>
<tr><td>2</td><td><i>Count</i> register increments every second cycle</td></tr>
<tr><td>3</td><td><i>Count</i> register increments every third cycle</td></tr>
<tr><td colspan="2">etc.</td></tr>
</table></td><td>R/W</td><td>0</td></tr>
<tr><td><i>CC</i></td><td>2</td><td><i>Count</i> register present. This register provides read access to the coprocessor 0 <i>Count</i> Register. Set this bit to 1 so a user-mode <b>rdhwr 2</b> can read out the value of the <i>Count</i> register.</td><td>R/W</td><td>0</td></tr>
</table>

Table 2.78 Field Descriptions for HWREna Register

Name	Bit(s)	Description	Read/ Write	Reset State
SYNCI_Step	1	L1 cache line size. Setting this bit allows hardware to read the line size of the L1 cache. This field is used in conjunction **`synci`** instruction. See that instruction’s description for the use of this value. In the typical implementation, this value should be zero if there are no caches in the system that must be synchronized (either because there are no caches, or because the instruction cache tracks writes to the data cache). In other cases, the return value should be the smallest line size of the caches that must be synchronized. For the proAptiv Multiprocessing System core, the SYNCI_Step value is 32 since the line size is 32 bytes. Set this bit to 1 so that 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 the instructions.	R/W	0
CPUNum	0	This register provides read access to the coprocessor 0 $EBase_{CPUNum}$ field. Set this bit 1 so a user-mode **`rdhwr 0`** reads out the CPU ID number.	R/W	0

2.3.11.2 UserLocal (CP0 Register 4, Select 2)

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

The presence of the *UserLocal* register is indicated by $Config3_{ULRI}$ = 1.

Figure 2.67 UserLocal Register Format

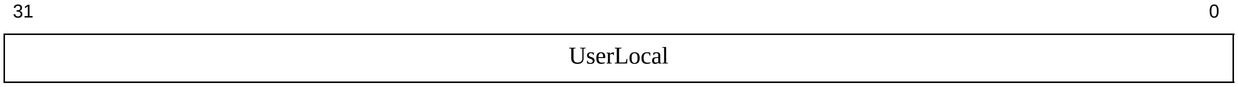


Table 2.79 UserLocal Register Field Description

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

2.3.12 Kernel Mode Support Registers

This section contains the following hardware access registers.

- Section 2.3.12.1, "Kernel Scratch Register 1 — KScratch1 (CP0 Register 31, Select 2)" on page 175

- Section 2.3.12.2, "Kernel Scratch Register 2 — KScratch2 (CP0 Register 31, Select 3)" on page 175
- Section 2.3.12.3, "Kernel Scratch Register 3 — KScratch3 (CP0 Register 31, Select 4)" on page 175

2.3.12.1 Kernel Scratch Register 1 — KScratch1 (CP0 Register 31, Select 2)

KScratch1 is a read-write 32-bit register that is used by the kernel for temporary storage of information.

The presence of the *KScratch1* register is indicated by $Config4_{KScrExist[2]}$ = 1’b1.

Figure 2.68 KScratch1 Register Format

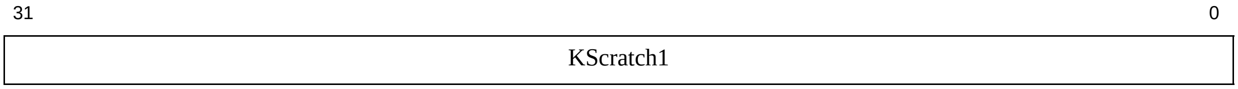


Table 2.80 KScratch0 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
KScratch1	31:0	Used by the kernel for temporary storage of information.	R/W	Undefined

2.3.12.2 Kernel Scratch Register 2 — KScratch2 (CP0 Register 31, Select 3)

KScratch2 is a read-write 32-bit register that is used by the kernel for temporary storage of information.

The presence of the *KScratch2* register is indicated by $Config4_{KScrExist[3]}$ = 1’b1.

Figure 2.69 KScratch2 Register Format

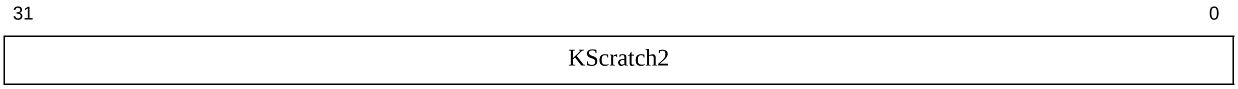


Table 2.81 KScratch2 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
KScratch2	31:0	Used by the kernel for temporary storage of information.	R/W	Undefined

2.3.12.3 Kernel Scratch Register 3 — KScratch3 (CP0 Register 31, Select 4)

KScratch3 is a read-write 32-bit register that is used by the kernel for temporary storage of information.

The presence of the *KScratch3* register is indicated by $Config4_{KScrExist[4]}$ = 1’b1.

Figure 2.70 KScratch3 Register Format

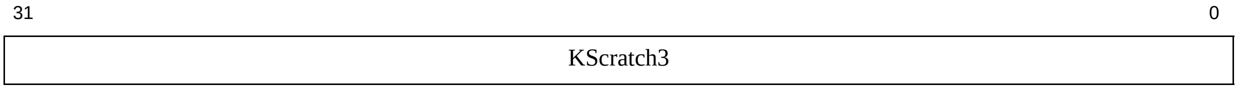


Table 2.82 KScratch2 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
KScratch3	31:0	Used by the kernel for temporary storage of information.	R/W	Undefined

2.3.13 Memory Mapped Registers

This section contains the following memory mapped registers. As shown by the blue text in Section 2.3.13.2, this register is only available in a multi-core environment implementing the Coherent Processing System (CPS).

- Section 2.3.13.1, "Common Device Memory Map Base Address — CDMMBase (CP0 Register 15, Select 2)" on page 176
- Section 2.3.13.2, "Coherency Manager Global Configuration Register Base Address — CMGCRBase (CP0 Register 15, Select 3)" on page 177

2.3.13.1 Common Device Memory Map Base Address — CDMMBase (CP0 Register 15, Select 2)

The 36-bit physical base address for the Common Device Memory Map facility is defined by this register. This register only exists if *Config3*$_{CDMM}$ is set to one.

Figure 2.71 shows the format of the *CDMMBase* register, and Table 2.83 describes the register fields.

Figure 2.71 CDMMBase Register

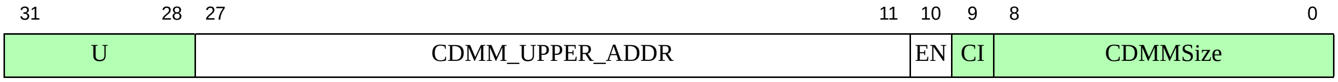


Table 2.83 CDMMBase Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
U	31:28	Unimplemented physical address bits. Writes are ignored, returns 0 on read	R	0
CDMM_UPPER_ADDR	27:11	Bits 31:15 of the base physical address of the common device memory-mapped registers.	R/W	Undefined
EN	10	Enables the CDMM region. If this bit is cleared, memory requests to this address region go to regular system memory. If this bit is set, memory requests to this region go to the CDMM logic. 0: CDMM region is disabled. 1: CDMM region is enabled.	R/W	0
CI	9	If set to 1 by hardware, this bit indicates that the first 64-byte Device Register Block (DRB) of the CDMM is reserved for additional registers which manage CDMM region behavior and are not IO device registers. This bit is always 0 in the proAptiv core since additional I/O device registers are not implemented.	R	0

Table 2.83 CDMMBase Register Field Descriptions*(continued)*

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td><i>CDMMSize</i></td><td>8:0</td><td>This field represents the number of 64-byte Device Register Blocks (DRB) instantiated in the proAptiv core.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>1 DRB</td></tr>
<tr><td>1</td><td>2 DRB’s</td></tr>
<tr><td>2</td><td>3 DRB’s</td></tr>
<tr><td>...</td><td>...</td></tr>
<tr><td>511</td><td>512 DRB’s</td></tr>
</table></td><td>R</td><td>2</td></tr>
</table>

2.3.13.2 Coherency Manager Global Configuration Register Base Address — CMGCRBase (CP0 Register 15, Select 3)

This register is used in a multi-core environment and defines the 36-bit physical base address for the memory-mapped Coherency Manager Global Configuration Register (CMGCR) space. This register only exists if Config3$_{\text{CMGCR}}$ is set.

Figure 2.72 shows the format of the *CMGCRBase* register, and Table 2.84 describes the register fields.

Figure 2.72 CMGCRBase Register

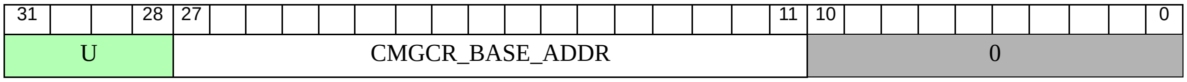


Table 2.84 CMGCRBase Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
U	31:28	Unimplemented physical address bits. Writes are ignored, returns 0 on read	R	0
CMGCR_ BASE_ADDR	31:11	Bits 35:15 of the base physical address of the memory-mapped Coherency Manager Global Configuration registers. The number of implemented physical address bits is implementation-specific. For the unimplemented address bits, writes are ignored, reads return zero. The reset value is set when the core is configured using the Configuration GUI.	R	Preset
0	10:0	Must be written as zero; returns zero on read	R	0

Chapter 3

Memory Management Unit

The proAptiv core includes a Memory Management Unit (MMU) that translates virtual addresses to physical addresses. The MMU consists of a 16-entry Instruction TLB (ITLB), a 32-entry data TLB (DTLB), 64 dual-entry Variable TLB (VTLB), and an optional 512 dual-entry Fixed TLB (FTLB). The FTLB is a build-time option.

This chapter contains the following sections:

3.1 Introduction

The MMU 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. The MMU also enforces the protection of memory areas and defines the cache protocols.

The MMU size in the proAptiv architecture has been increased from the traditional 64 dual entries up to 576 dual entries, thereby increasing performance by reducing the number of TLB misses.

An Enhanced Virtual Address (EVA) scheme allows physical addresses above 0.5 GB to be translated without using the *HighMem* protocol. The EVA scheme allows for user space segments up to 3.5 GB.

3.2 Memory Management Unit Architecture

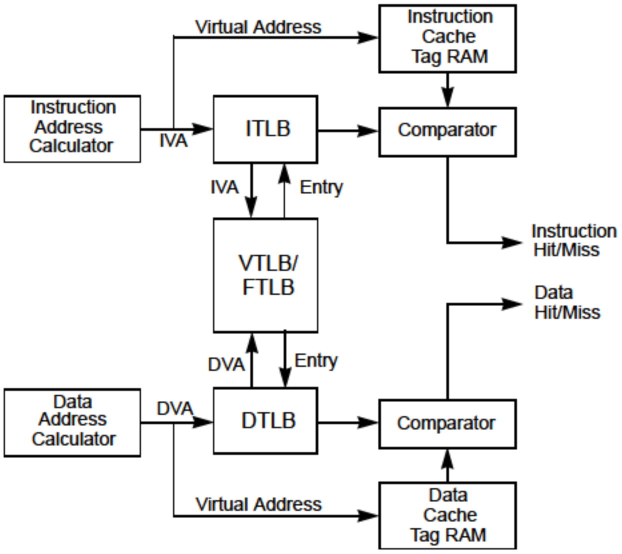
The Memory Management Unit (MMU) in the proAptiv core consists of four address-translation lookaside buffers (TLB):

- 16-entry Instruction TLB (ITLB)
- 32 dual-entry Data TLB (DTLB)
- 64 dual-entry Variable Page Size Translation Lookaside Buffer (VTLB)
- Optional 512 dual-entry Fixed Page Size Translation Lookaside Buffer (FTLB)

When an instruction address is to be translated, the ITLB is accessed first. If the translation is not found, the VTLB/FTLB is accessed. If there is a miss in the VTLB/FTLB, an exception is taken. Similarly, when a data reference is to be translated, the DTLB is accessed directly. If the address is not present in the DTLB, the VTLB/FTLB is accessed. If there is a miss in the VTLB/FTLB, an exception is taken.

Figure 3.1 shows an overview of the proAptiv MMU architecture.

Figure 3.1 Overview of MMU Architecture in the proAptiv Core



3.2.1 Instruction TLB (ITLB)

The ITLB is a 16-entry high speed TLB dedicated to performing translations for the instruction stream. The ITLB maps only 4 KB or 16 KB pages. For 4 KB or 16 KB pages, the entire page is mapped in the ITLB.

The ITLB is managed by hardware and is transparent to software. The larger VTLB/FTLB is used as a backup structure for the ITLB. If a fetch address cannot be translated by the ITLB, the VTLB/FTLB attempts to translate it in the following clock cycle or when available. If successful, the translation information is copied into the ITLB for future use.

The ITLB is functionally invisible to software and is automatically refilled from the VTLB/FTLB when required, and automatically cleared whenever the VTLB/FTLB is updated.

3.2.2 Data TLB (DTLB)

The DTLB is a 32 dual-entry high speed TLB dedicated to performing translations for the data stream. The DTLB maps only 4 KB or 16 KB pages. For 4 KB or 16 KB pages, the entire page is mapped in the DTLB.

The DTLB is managed by hardware and is transparent to software. The larger VTLB/FTLB is used as a backup structure for the DTLB. If a load/store address cannot be translated by the DTLB, the VTLB/FTLB/SegCtl logic[1] attempts to translate it in the following clock cycle or when available. If successful, the translation information is copied into the DTLB for future use.

The ITLB is functionally invisible to software and is automatically refilled from the VTLB/FTLB when required, and automatically cleared whenever the VTLB/FTLB is updated.

3.2.3 Variable Page Size TLB (VTLB)

The VTLB is a fully associative variable page size translation lookaside buffer with 64 dual entries. The purpose of the VTLB 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 VTLB structure. This structure is used to translate both instruction and data virtual addresses.

The VTLB is organized as 64 pairs of even and odd entries. The VTLB implements the following page sizes:

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

The VTLB/FTLB 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.

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. Software can determine which page sizes are supported by writing all ones to the *PageMask* register, then reading the value back.

The VTLB/FTLB entries are controlled through select CP0 registers. Refer to Section 3.5, "Relationship of TLB Entries and CP0 Registers" for more information.

3.2.4 Fixed Page Size TLB (FTLB)

The 512-entry FTLB is a fixed page size TLB organized as 128 sets and 4-ways. Each set of each way contains dual data RAM entries and one tag RAM entry. If the tag RAM contents matches the requested address, either the low or high RAM location of the dual data RAM is accessed depending on the state of the most-significant-bit (MSB) of the VPN2 offset portion of the virtual address. Refer to Section 3.5.3, "Address Translation Examples" for more information on VPN2 usage.

[1] The VTLB/FTLB is used during mapped accessed. The SegCtl registers are used during unmapped accesses.

The FTLB is organized as 512 pairs of even and odd entries. The FTLB implements the following page sizes:

4K, 16K

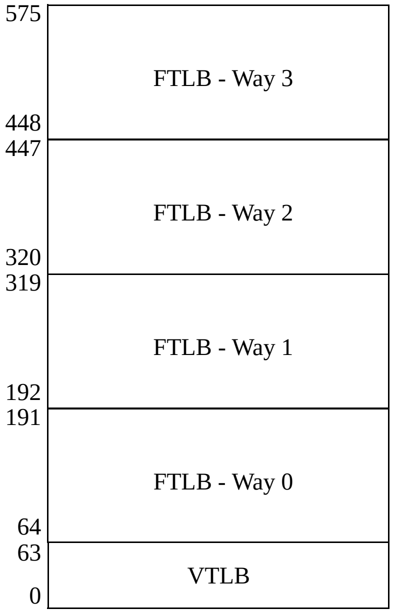
If the FTLB is implemented, the organization is as shown in Table 3.1. Note that all of the entries in the FTLB must be the same page size, either 4K or 16K. The size is determined by the *Config4*$_{\textit{FTLB Page Size}}$ field as described in the following table.

Table 3.1 FTLB Configuration Options

FTLB Parameter	Programmable Options	Register Reference
Ways	4 ways	*Config4*$_{\textit{FTLB Ways}}$
Sets	128 sets	*Config4*$_{\textit{FTLB Sets}}$
Page Size	4 KB 16KB	*Config4*$_{\textit{FTLB Page Size}}$

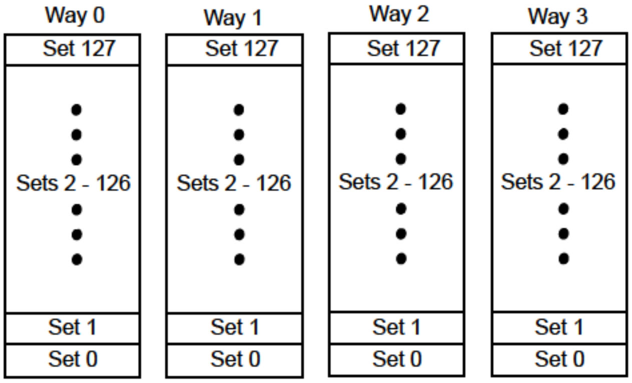
The FTLB resides at the top of the VTLB range as shown in Figure 3.2.

Figure 3.2 proAptiv VTLB and FTLB



As shown in Figure 3.3, the 512-entry FTLB contains four ways and 128 sets. Each set of each way contains one dual-entry.

Figure 3.3 FTLB Organization



3.3 MMU Configuration Options

The MMU in the proAptiv core can be configured with the following options.

- FTLB present/not present
- MMU type
- MMU size and organization

3.3.1 FTLB Present

The proAptiv core allows software to enable and disable the 512-entry FTLB. This is done via the *FTLBEn* bit in the *Config6* register (CP0 Register 16, Select 6). Depending on how this bit is set, one of the following will occur:

- If the FTLB is present (build time option) and the *Config6*$_{FTLBEn}$ bit is set by software, the FTLB is enabled and the hardware will configure the device accordingly.
- If the FTLB is present and the *Config6*$_{FTLBEn}$ bit is cleared by software, the FTLB is disabled. This mode allows the proAptiv core to remain backward compatible with existing software. Note that if the *Config6*$_{FTLBEn}$ bit is cleared, the address translation mechanism acts just like a Joint TLB (JTLB) in previous generation MIPS processors.
- If the FTLB is present and the *Config6*$_{FTLBEn}$ bit is not programmed by software, the FTLB is disabled by default because this bit is cleared automatically at reset.
- If the FTLB is not present (build time option), hardware ignores the state of the *Config6*$_{FTLBEn}$ bit and sets the *Config*$_{MT}$ field to 3'b001. Refer to the following subsection for more information.

These options are illustrated in the Table 3.2.

Table 3.2 FTLB Present or Not Present in the System

FTLB Present/Not Present (Build Time Option)	*Config6*$_{FTLBEn}$ Bit (Set by Software)	*Config*$_{MT}$ Field[1] (Set by Hardware)
Present	1	3'b100 (FTLB Present and Enabled)
Present	0	3'b001 (FTLB Present and Disabled)

Table 3.2 FTLB Present or Not Present in the System

FTLB Present/Not Present (Build Time Option)	*$Config6_{FTLBEn}$* Bit (Set by Software)	*$Config_{MT}$* Field[1] (Set by Hardware)
Not Present	Don’t Care	3’b001 (FTLB Not Present)

[1] See Section 3.3.2, "MMU Type".

Note that the size of the FTLB is fixed at 512 entries. The user cannot implement less than 512 entries if the FTLB is enabled.

3.3.2 MMU Type

The *MT* field of the *Config* register (CP0 Register 16, Select 0) is programmed depending on whether the FTLB is enabled. This is determined by the state of the *$Config6_{FTLBEn}$* bit described above. If *$Config6_{FTLBEn}$* is cleared, or if the FTLB is not present, hardware writes a value of 3’b001 to this field. If *$Config6_{FTLBEn}$* is set, hardware writes a value of 3’b100 to this field. The kernel code uses this field to determine how to configure the TLB.

The 3-bit *$Config_{MT}$* field supports the following two encodings. All other encodings are reserved.

- 3’b001: VTLB only (FTLB disabled or not present)
- 3’b100: VTLB and FTLB present

3.3.3 MMU Size and Organization

The proAptiv core uses the following CP0 register fields to determine the size and organization of the MMU. Each of the items below is described in the following subsections.

- Bits 30:25 of the *Config1* register (*$Config1_{MMUSIZE}$*). Determines VTLB size.
- Bits 15:14 of the *Config4* register (*$Config4_{MMUExtDef}$*). Determines how bits 12:0 of this register are used based on whether an FTLB is present. If the value in this field is 2’b01, the FTLB is not present and the lower 7 bits of the *Config4* register are used to extend the VTLB size as described below. If the value in this field is 2’b11, the FTLB is present and the lower 13 bits of the *Config4* register are used to determine the size and organization of the FTLB.
- Bits 12:8 of the *Config4* register (*$Config4_{FTLB\ Page\ Size}$*). Determines the FTLB page size. These bits are only used when the FTLB is present and enabled. If the FTLB is not present or disabled, this field is ignored.
- Bits 7:4 of the *Config4* register (*$Config4_{FTLB\ Ways}$*). If an FTLB is present and enabled, this field determines the number of ways in the FTLB. If the FTLB is not present or disabled, this field is used to extend the VTLB size as described below.
- Bits 3:0 of the *Config4* register (*$Config4_{FTLB\ Sets}$*). If an FTLB is present and enabled, this field determines the number of sets per way in the FTLB. If the FTLB is not present or disabled, this field is used to extend the VTLB size as described below.

3.3.3.1 Determining VTLB Size

The 6-bit *MMUSize* field in the *Config1* register (*$Config1_{MMUSize}$*) is used to determine the number of entries in the VTLB. Hardware writes a value of 0x3F into this field at reset, indicating 64 entries. Note that the number of VTLB entries in the proAptiv core is fixed at 64. The user cannot modify this value.

3.3.3.2 Interpreting the Config4 Fields

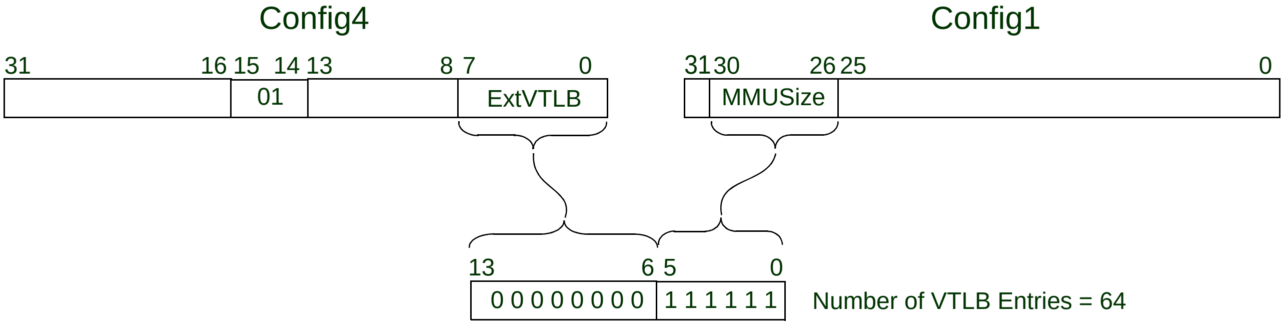
The 2-bit *MMUExtDef* field in the *Config4* register (*Config4*$_{MMUExtDef}$) is used to determine how bits 12:0 of this register are interpreted. This read-only field is written by hardware based on the setting of the *FTLBEn* bit in the *Config6* register (CP0 Register 16, Select 6). The *MMUExtDef* field can be written with one of the following two values. All other values are reserved.

- *Config4*$_{MMUExtDef}$ = 2'b01; VTLB enabled, FTLB disabled or not present
- *Config4*$_{MMUExtDef}$ = 2'b11; VTLB enabled, FTLB enabled

FTLB Not Present or Disabled

If the FTLB is either not present or disabled, bits 7:0 of the *Config4* register are used to extend the number of VTLB entries indicated by the 6-bit *Config4*$_{MMUSize}$ field described in the previous subsection. This extends the number of bits used to determine the number of VTLB entries from 6 to 14. This concept is shown in Figure 3.4 below.

Figure 3.4 Extending the Number of VTLB Entries — FTLB Not Present or Disabled



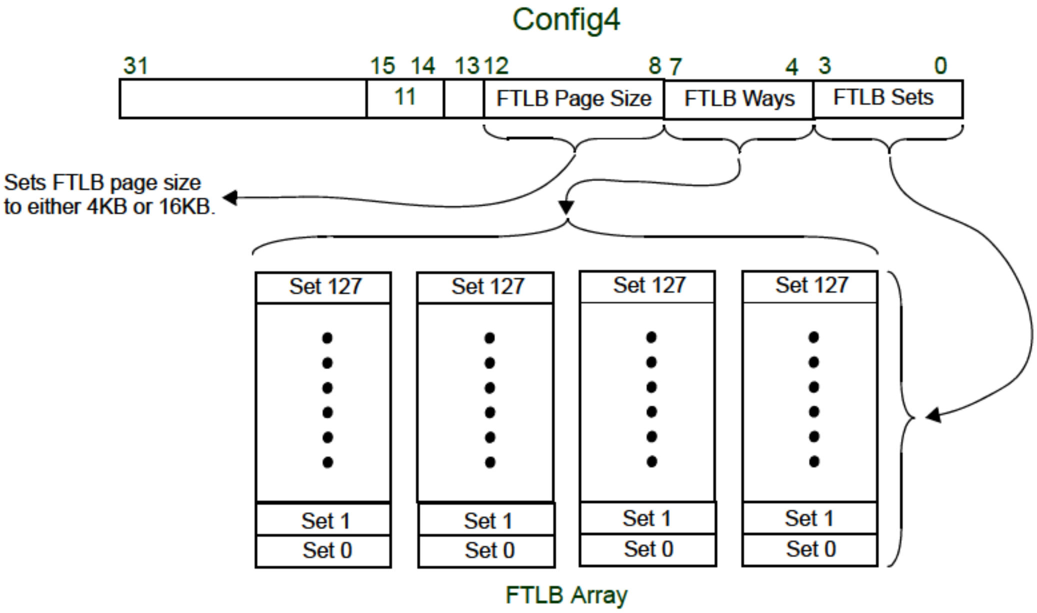
This field can be used to extend the number of VTLB entries in future generations of MIPS processors. Since there is a maximum of 64 VLTB entries in the proAptiv core, bits 7:0 of the *Config4* register always contain a value of 0 when only the VTLB is present.

FTLB Present and Enabled

If the *FTLBEn* bit is set, indicating the FTLB is present and enabled, hardware writes a value of 2'b11 into the *Config4*$_{MMUExtDef}$ field. In this case, bits 12:0 of the *Config4* register are used to determine the size and organization of the FTLB.

When the *Config4*$_{MMUExtDef}$ field is set to 2'b11, bits 12:0 of this register are used to indicate the FTLB page size (*Config4*$_{FTLB\ Page\ Size}$), the number of ways (*Config4*$_{FTLB\ Ways}$), and the number of sets (*Config4*$_{FTLB\ Sets}$). In the proAptiv core, only the FTLB page size is programmable. The number of ways is fixed at 4 and the number of sets is fixed at 128. The page size can be programmed to either 4KB or 16KB pages. This concept is shown in Figure 3.5.

Figure 3.5 Determining the FTLB Characteristics — FTLB Present and Enabled



3.4 Overview of 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 TLB entry after masking out the bits specified by the entries page size, and either:

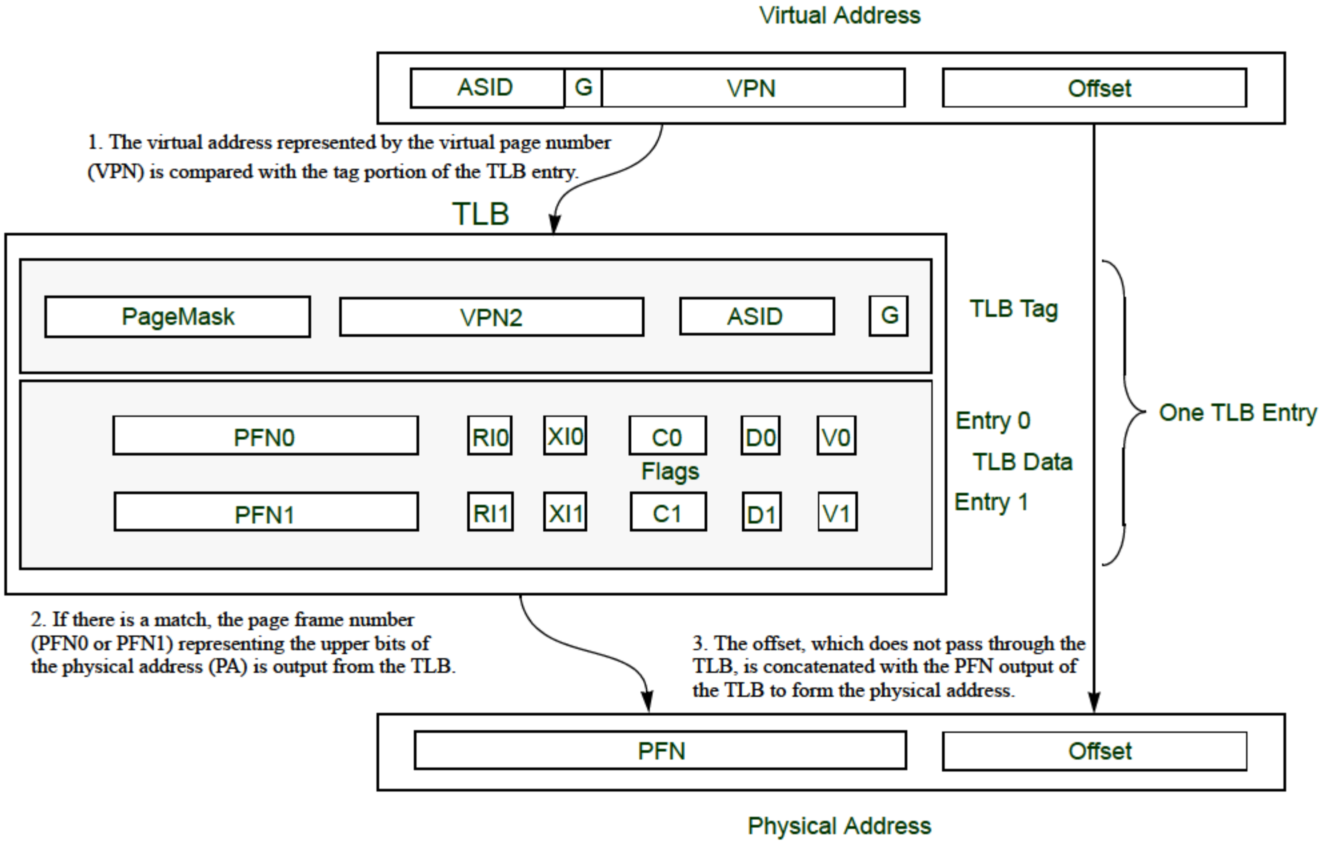
- The Global (G) bit of both the even and odd pages of the TLB entry is set, or
- The Global (G) bit is cleared and the ASID field of the virtual address is the same as the ASID field of the TLB entry

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

Figure 3.6 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.

Note that the various register fields used during a TLB translation are managed via CP0 registers as described in Section 3.5, "Relationship of TLB Entries and CP0 Registers".

Figure 3.6 Overview of Virtual to Physical Address Translation


Virtual Address
ASID
G
VPN
Offset
1. The virtual address represented by the virtual page number (VPN) is compared with the tag portion of the TLB entry.
TLB
PageMask
VPN2
ASID
G
TLB Tag
PFN0
RI0
XI0
C0
D0
V0
Flags
PFN1
RI1
XI1
C1
D1
V1
Entry 0
TLB Data
Entry 1
One TLB Entry
2. If there is a match, the page frame number (PFN0 or PFN1) representing the upper bits of the physical address (PA) is output from the TLB.
3. The offset, which does not pass through the TLB, is concatenated with the PFN output of the TLB to form the physical address.
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 3.6, the *Offset* does not pass through the TLB. Note that if the G bit is set, the ASID is ignored and the TLB compares only the VPN portion of the virtual address. The G bit is a logical AND of the G bit in the *EntryLo0* and *EntryLo1* registers.

Figure 3.7 shows a flow diagram of the address translation process 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).

Figure 3.7 32-bit Virtual Address Translation — 4 KB Page Size

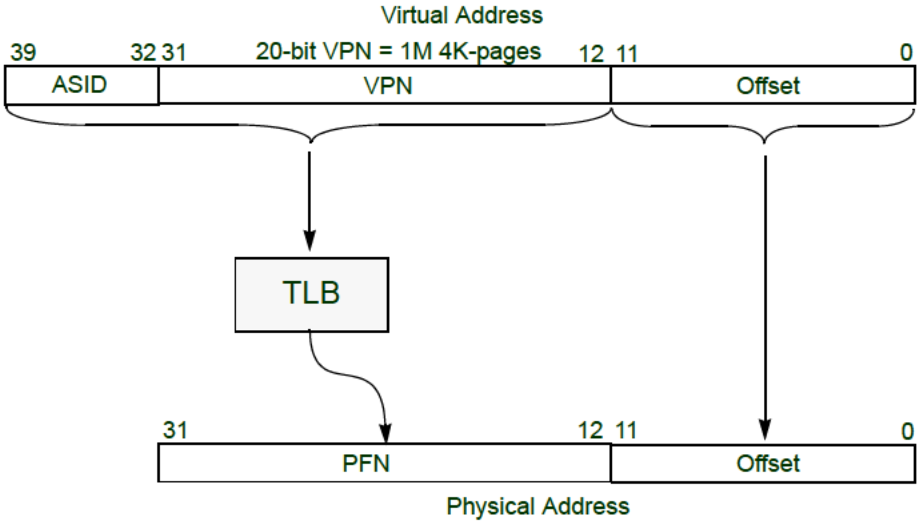
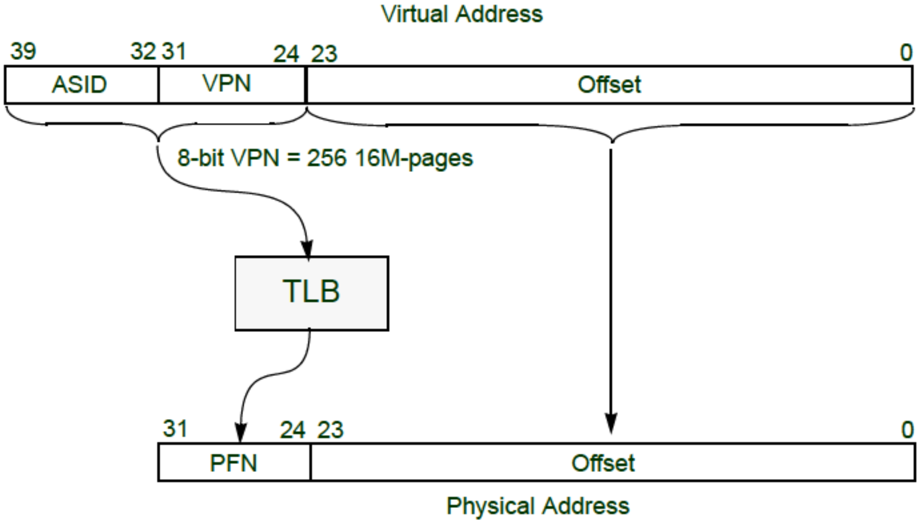


Figure 3.8 shows a flow diagram of the address translation process for a 16 MByte page size. The width of the *Offset* is defined by the page size. The remaining 8 bits of the address represent the virtual page number (VPN).

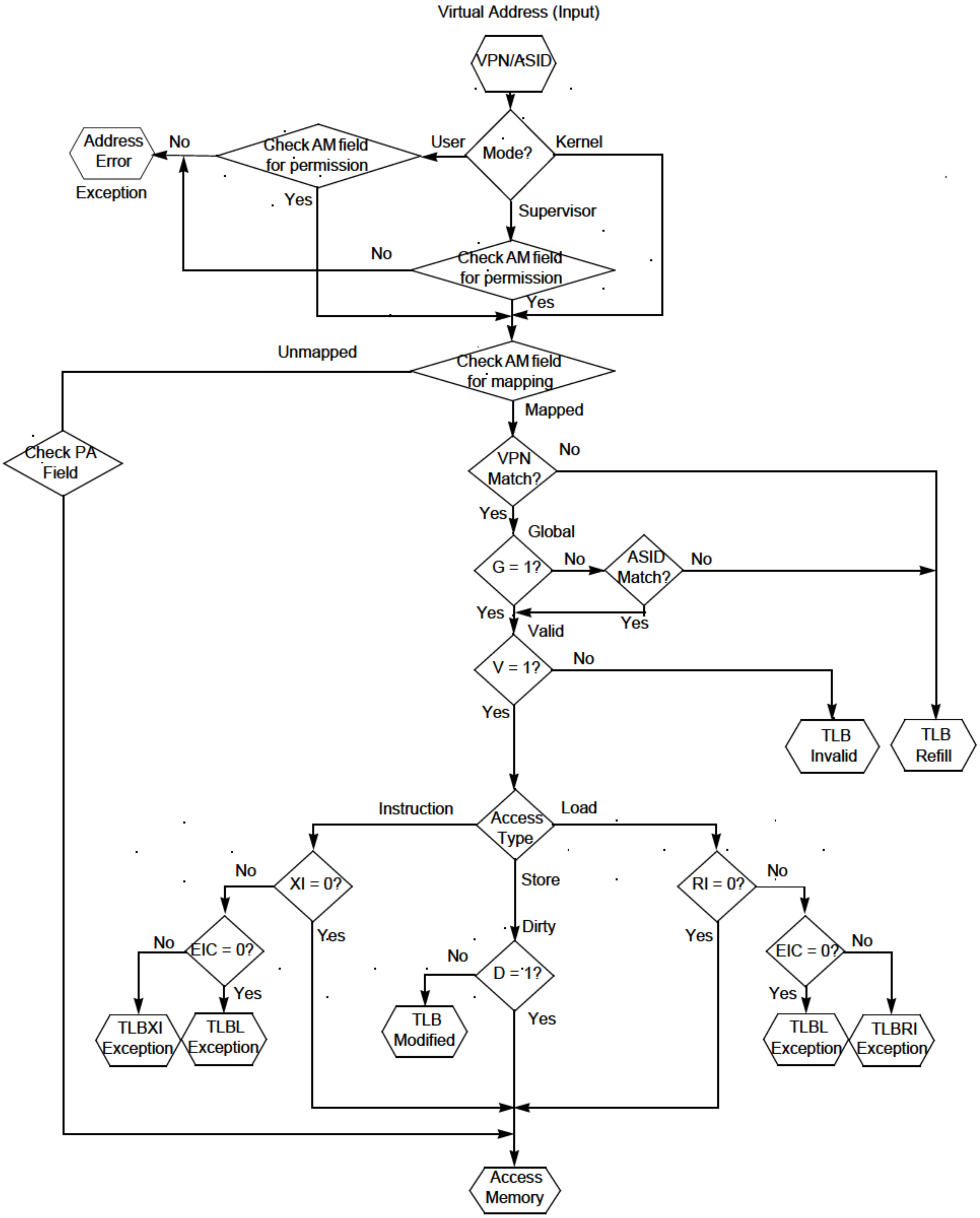
Figure 3.8 32-bit Virtual Address Translation — 16 MB Page Size



3.4.1 Address Translation Flow

During an address translation, the hardware checks for various conditions such as the addressing mode (user, kernel etc.), access permissions based on the mode, the access type (load/store, etc), and the state of selected bits in the TLB entry. If one or more of the conditions for translation are not met, a TLB exception is taken. This concept is shown in Figure 3.9.

Figure 3.9 Address Translation Flow



3.5 Relationship of TLB Entries and CP0 Registers

Each TLB entry in the VTLB/FTLB consists of a tag portion and dual-data portion as shown in Figure 3.10. In this figure, the following registers are used to manage the TLB entries.

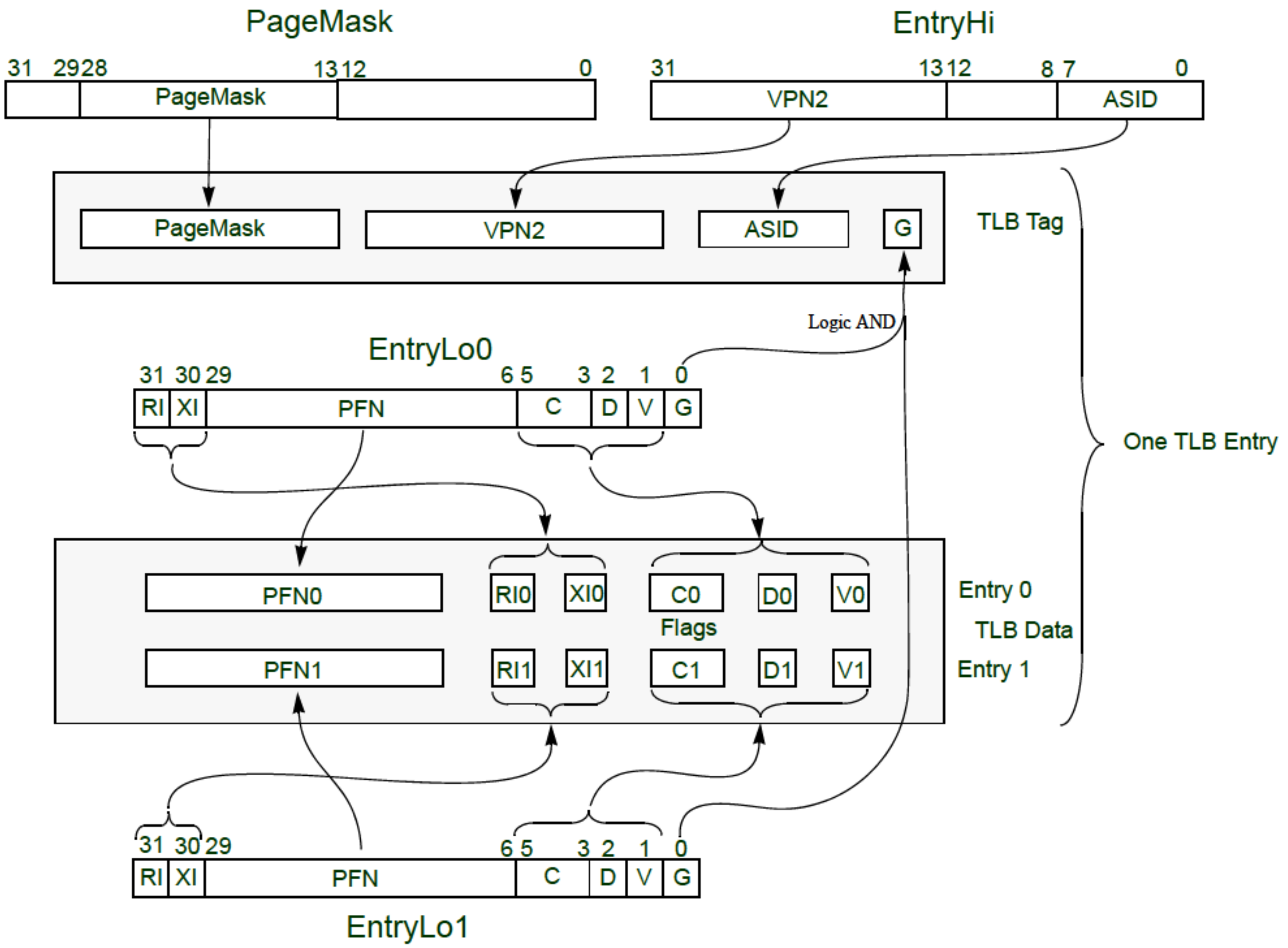
- *EntryLo0* (CP0 Register 2, Select 0)
- *EntryLo1* (CP0 Register 3, Select 0)
- *EntryHi* (CP0 Register 10, Select 0)
- *PageMask* (CP0 Register 5, Select 0)

In order to fill an entry in the VTLB/FTLB, software executes a **`TLBWI`** or **`TLBWR`** instruction (see Section 3.18). Prior to invoking one of these instructions, the CP0 registers listed above must be updated with the information to be written to the TLB entry:

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

These register fields and their relationship to a TLB entry is described in the following subsections.

Figure 3.10 Relationship Between CP0 Registers and TLB Entries



3.5.1 TLB Tag Entry

The tag portion of the TLB entry contains the fields necessary to match an incoming address against that entry. This section describes each field of the TLB tag entry shown in Figure 3.10.

3.5.1.1 VPN2 Field

The virtual page number (VPN) contains the high bits of the program (virtual) address. The 'VPN2' designation indicates that this address is for a double-page-size virtual region which will map to a pair of physical pages. The VPN2 field is generated using the *EntryHi* register.

Note that on a TLB-related exception, the *VPN2* field is automatically set to the virtual address that was being translated when the exception occurred. If the outcome of the exception handler is to find and install the translation to that address, the *VPN2* field will already contain the correct value.

3.5.1.2 ASID Field

The address space identifier (ASID) helps to reduce the frequency of TLB flushing on a context switch. The ASID field extends the virtual address with an 8-bit memory space identifier assigned by the operating system. The ASID allows translations for multiple different applications to co-exist in the TLB (in Linux, for example, each application has different code and data lying in the same virtual address region). The ASID field is generated using the *EntryHi* register.

3.5.1.3 PageMask Field

The size of the tag can be configured using the ‘PageMask’ field. This field determines how many incoming address bits to match. For the VTLB, the proAptiv core allows page sizes of 4 Kbytes up to 256 Mbytes in multiples of four. For the FTLB, the proAptiv core allows page sizes of 4 Kbytes and 16 Kbytes. The *PageMask* field is generated using the *PageMask* register.

In the *PageMask* field, a ‘1’ on a given bit means "don’t compare this address bit when matching this address". However, only a restricted range of *PageMask* values are legal. The values must start with "1"s filling the *PageMask* field from the low-order bits upward, two at a time. A list of valid 32-bit *PageMask* register values, the corresponding binary value of the PageMask[28:13] field, and the corresponding page size is shown in Table 3.3. For the PageMask[28:13] field, note that the bits are set two at a time from the least significant bit (LSB) to the most significant bit (MSB).

Table 3.3 PageMask Value and Corresponding Page Size

32-bit PageMask Register Value	PageMask[28:13]	Page Size	Even/Odd Bank Select Bit
0x0000_0000	0x00_0000_0000_0000_00	4 KBytes	VAddr[12]
0x0000_6000	0x00_0000_0000_0000_11	16 KBytes	VAddr[14]
0x0001_E000	0x00_0000_0000_0011_11	64 KBytes	VAddr[16]
0x0007_E000	0x00_0000_0000_1111_11	256 KBytes	VAddr[18]
0x001F_E000	0x00_0000_1111_1111_11	1 MByte	VAddr[20]
0x007F_E000	0x00_0011_1111_1111_11	4 MBytes	VAddr[22]
0x01FF_E000	0x00_0011_1111_1111_11	16 MBytes	VAddr[24]
0x07FF_E000	0x00_1111_1111_1111_11	64 MBytes	VAddr[26]
0x1FFF_E000	0x11_1111_1111_1111_11	256 MBytes	VAddr[28]

Note that the 4 KByte and 16 KByte entries in the above table correspond to the VTLB and the FTLB. All other entries correspond to the VTLB only.

3.5.1.4 Global (G) Bit

The ‘G’ (global) bit in the tag entry is a logical AND between the *G* bits of the *EntryLo0* and *EntryLo1* registers. When set, it causes addresses to match regardless of their ASID value, thus defining a part of the address space which will be shared by all applications. For example, Linux applications share some ‘kseg2’ space used for kernel extensions.

Note that since the G bit in the TLB tag entry is a logical AND between two *G* bits, software must be sure to set $EntryLo0_G$ and $EntryLo1_G$ to the same value.

3.5.2 TLB Data Entry

The data portion of the TLB entry contains the data and associated flag bits for the corresponding tag entry. This section describes each field of the TLB data entry shown in Figure 3.10.

3.5.2.1 Page Frame Number (PFN)

The Page Frame Number (PFN) contains the high-order bits of the physical address. The 20-bit *PFN*, together with the lower 12 bits of address that are not translated, make up the 32-bit physical address.

3.5.2.2 Flag Fields (C, D, V, RI, and XI)

These flag bits contain information about the translated address. All of these bits are generated by the *EntryLo0* and *EntryLo1* registers.

C Field: This field contains the cacheability attributes for the corresponding TLB entry. It indicates how to cache data for this page. Pages can be marked cacheable, uncacheable, coherent, non-coherent, uncached accelerated, write-back, write-allocate, etc.

D bit: The "dirty" flag. Setting this bit indicates that the page has been written, and/or is writable. If this bit is a one, stores to the page are permitted. If this bit is a cleared, stores to the page cause a *TLB Modified* exception. Software can use this bit to track pages that have been written to. When a page is first mapped, this bit should be cleared. It is set on the first write that causes an exception.

V bit: The "valid" flag. 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 this bit is a zero, accesses to the page cause a *TLB Invalid* exception.

RI bit: The ‘read inhibit’ flag. If this bit is set in a TLB entry, any attempt (other than a MIPS16 PC-relative load) to read data on the virtual page causes either a *TLBL* or a *TLBRI* exception depending on the state of the $PageGrain_{EIC}$ bit, even if the *V* (Valid) bit is set. If the $PageGrain_{EIC}$ bit is cleared, and *TLBL* exception is taken. If the $PageGrain_{EIC}$ bit is set, and *TLBRI* exception is taken. Note that the *RI* bit is writable only if the *RIE* bit of the *PageGrain* register is set.

XI bit: The ‘execute inhibit’ flag. If this bit is set in a TLB entry, any attempt to fetch an instruction or to load MIPS16 PC-relative data from the virtual page causes either a *TLBL* or *TLBXI* exception depending on the state of the $PageGrain_{EIC}$ bit, even if the *V* (Valid) bit is set. If the $PageGrain_{EIC}$ bit is cleared, and *TLBL* exception is taken. If the $PageGrain_{EIC}$ bit is set, and *TLBXI* exception is taken. Note that the *XI* bit is writable only if the *XIE* bit of the *PageGrain* register is set.

3.5.3 Address Translation Examples

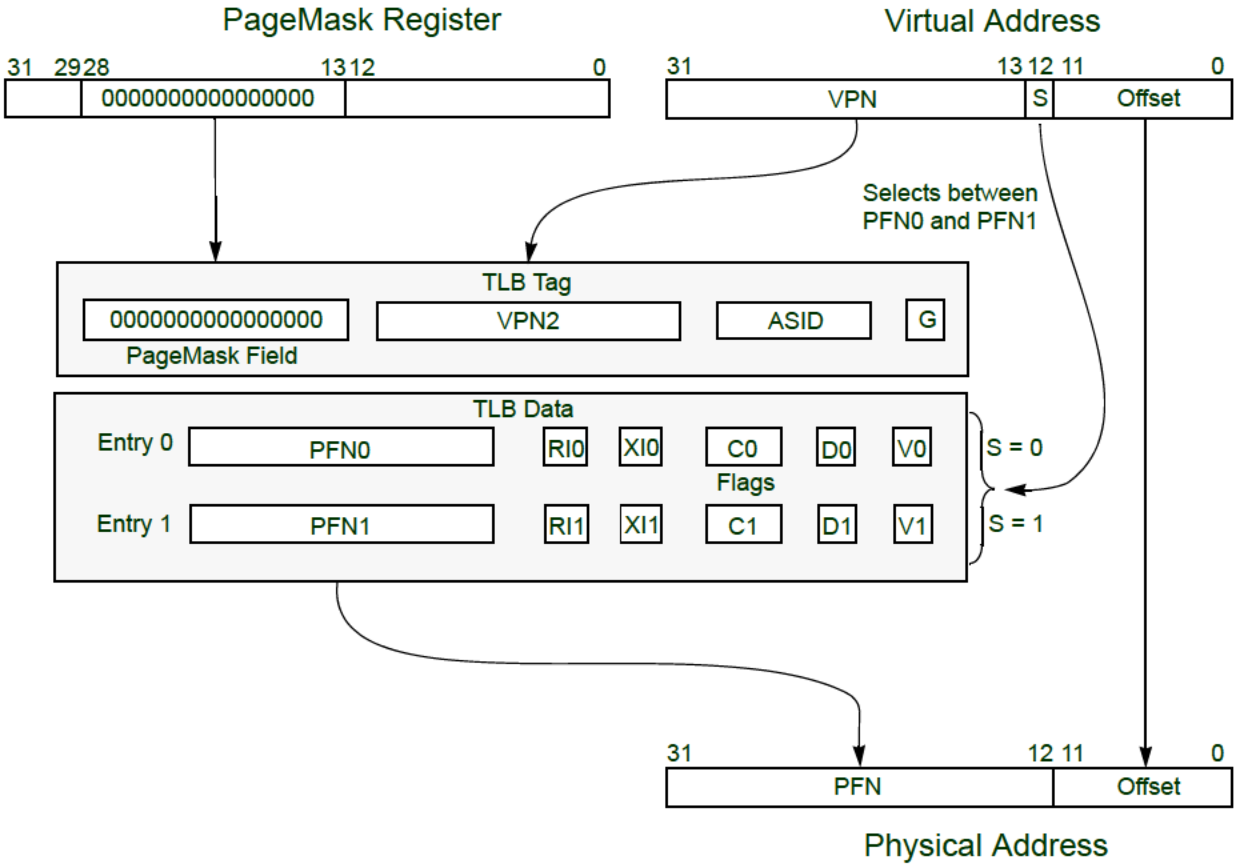
As shown in Figure 3.10, there are two PFN values for each tag match. Which of them is used is determined by the lowest-order bit of the VPN field of the address. So in standard form (using 4KByte pages) each entry translates an 8KByte region of virtual address, but each 4Kbyte page can be mapped onto any physical address (with any permission flag bits). This concept is described in the following subsections.

4 KByte Page Size Example

In a 4KB page size, 12 address bits are required to select an entry within the page. Therefore, 12 bits of the virtual address are used for the offset into the page table. The upper 20 bits of the virtual address are used as a pointer to the page table. With a 4 KByte page size, this allows support for up to 1M page table entries.

The upper 20 bits of virtual address pass through the TLB to generate the corresponding physical address. As described in Section 3.4, the proAptiv architecture implements a dual-entry VTLB/FTLB scheme, where each TLB tag corresponds to two data entries. To select between these two entries, hardware reads the low-order bit of the VPN (first bit after the offset, shown as the S bit in the figure below). In a 4 KByte page example, this equates to bit 12. This is shown in Figure 3.11.

Figure 3.11 Selecting Between PFN0 and PFN1 — 4 KByte Page Size



As shown in Figure 3.11, the *PageMask* field is derived from the *PageMask* register and is used to determine the page size for the application. Since the proAptiv architecture supports VTLB/FTLB page sizes in multiples of four (4 KByte, 16 KByte, 64 KByte, etc. up to 256 MByte), page masking is done in pairs. During translation, hardware checks the VPN against the contents of the *PageMask* field to determine the page size, and therefore how many VPN bits to compare. Refer to Table 3.3 for a list of valid *PageMask* values.

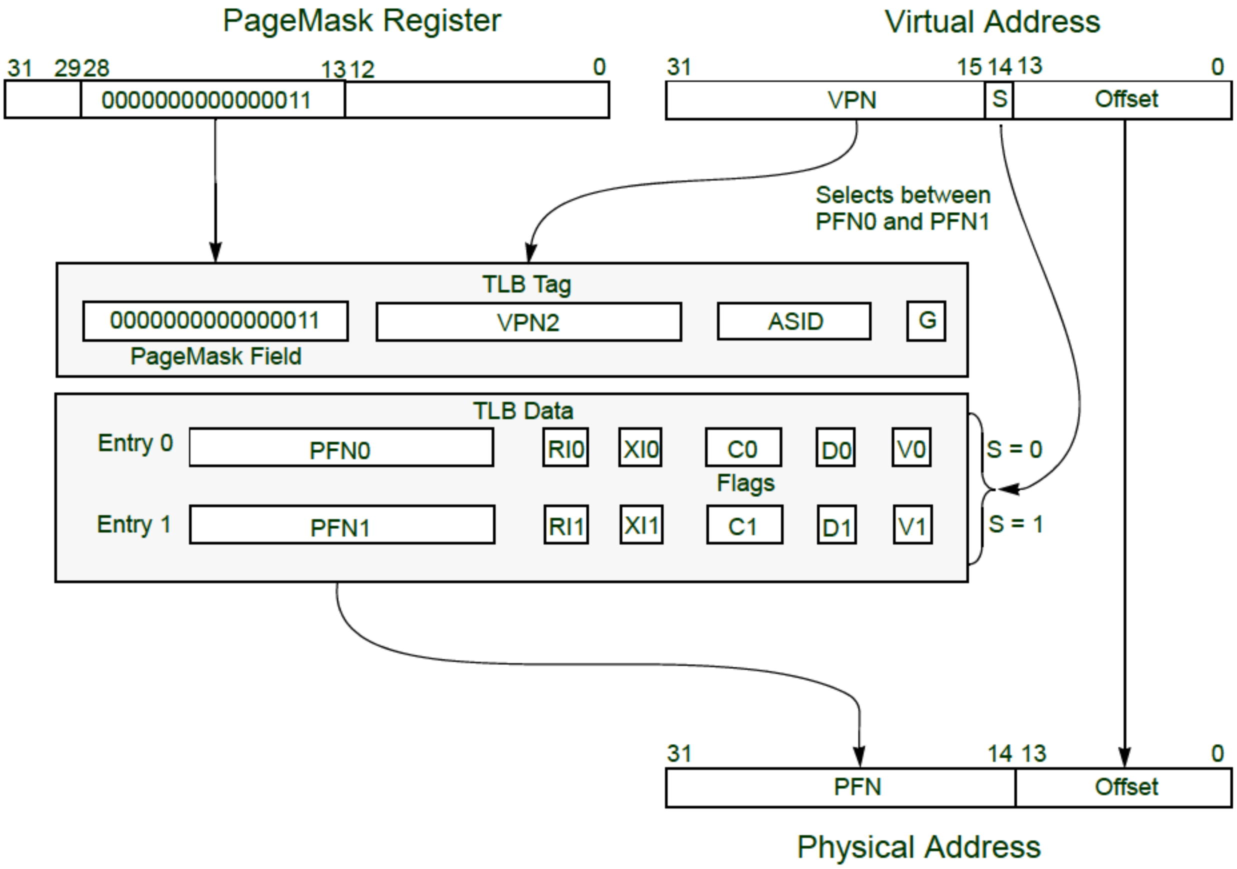
In the above example, all of the PageMask field bits are 0, indicating a 4 KByte page size. For a 16 KByte page size, bits 12 and 13 of the PageMask field would be set. This concept is described below.

16 KByte Page Size Example

In a 16 KByte page size, 14 address bits are required to select an entry within the page. Therefore, 14 bits of the virtual address are used for the offset into the page table. The upper 18 bits of the virtual address are used as a pointer to the page table. With a 16 KByte page size, this allows support for up to 256K page tables.

The upper 18 bits of virtual address pass through the TLB to generate the corresponding physical address. As described in Section 3.4, the proAptiv architecture implements a dual-entry VTLB/FTLB scheme, where each TLB tag corresponds to two data entries. To select between these two entries, hardware reads the low-order bit of the VPN (first bit after the offset, shown as the S bit in the figure below). In a 16 KByte page example, this equates to bit 14. This is shown in Figure 3.12.

Figure 3.12 Selecting Between PFN0 and PFN1 — 16 KByte Page Size



As shown in Figure 3.12, the *PageMask* field is used to determine the page size for the application. During translation, hardware checks the VPN against the contents of the *PageMask* field to determine the page size, and therefore how many VPN bits to compare. In the above example, the lower 2 bits of the *PageMask* field bits are 11, indicating a 16 KByte page size. Refer to Table 3.3 for a list of valid *PageMask* values.

3.6 Enhanced Virtual Address

Traditional MIPS virtual memory support divides up the virtual address space into fixed size segments, each with fixed attributes and access privileges. Such a scheme limits unmapped kernel access to 512 MBytes, the size of kseg0/kseg1. Furthermore, application sizes are growing beyond the 2GB limit imposed by the *useg* user segment.

Programmable Memory Segmentation relaxes these limitations. The size of virtual address space segments can be programmed, as can their attributes and privilege access. With this ability to overlap access modes, *kseg0* can now be extended up to 3.0GB[2], leaving at least one 1.0GB segment for mapped kernel accesses. This extended *kseg0* is called *xkseg0*. *xkseg0* overlaps with *useg*, because segments in *xkseg0* are programmed to support mapped user accesses and unmapped kernel accesses. Consequently, user space is equal to the size of *xkseg0*, which can be up to 3.0GB.

To allow for efficient kernel access to user space, new load and store instructions have been defined which allow kernel mapped access to *useg*. The new instructions, along with *Programmable Memory Segmentation*, are requirements for the scheme, called *Enhanced Virtual Address* or EVA, which allows for more efficient use of 32b address space.

3.6.1 Virtual and Physical Address Maps

In previous generation MIPS32 processors, the address map was fixed as shown in Figure 3.13. In this architecture, physical memory is limited by *kseg0* to 0.5GB, the amount of kernel unmapped cached address space. This memory must also be shared by the I/O and kernel, thus in reality less than 0.5GB is available to any user process.

Figure 3.13 Traditional Virtual Address Mapping in Previous Generation MIPS32 Processors

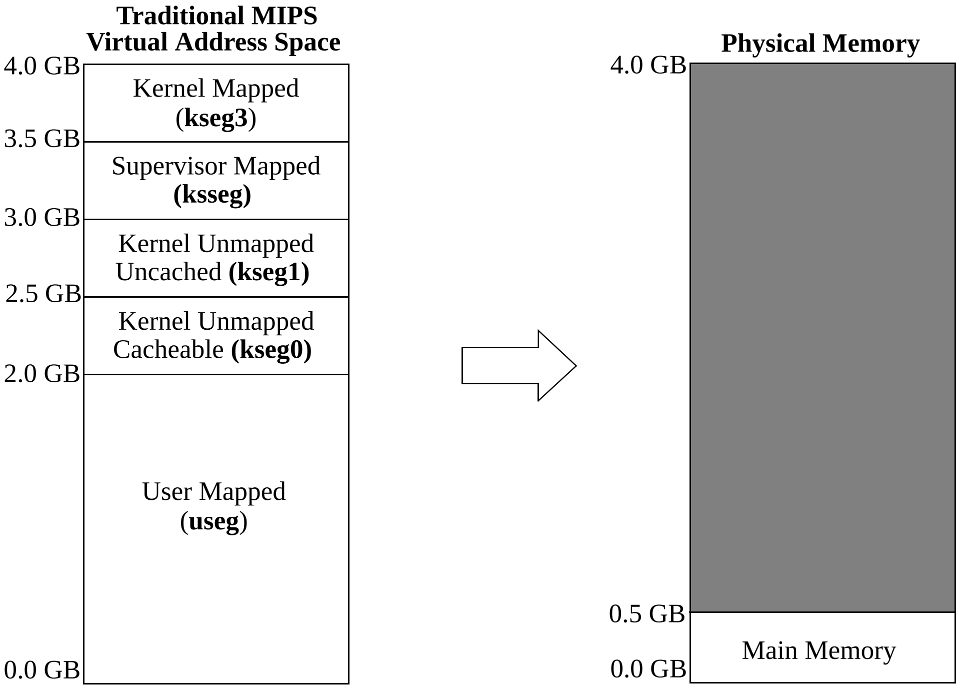
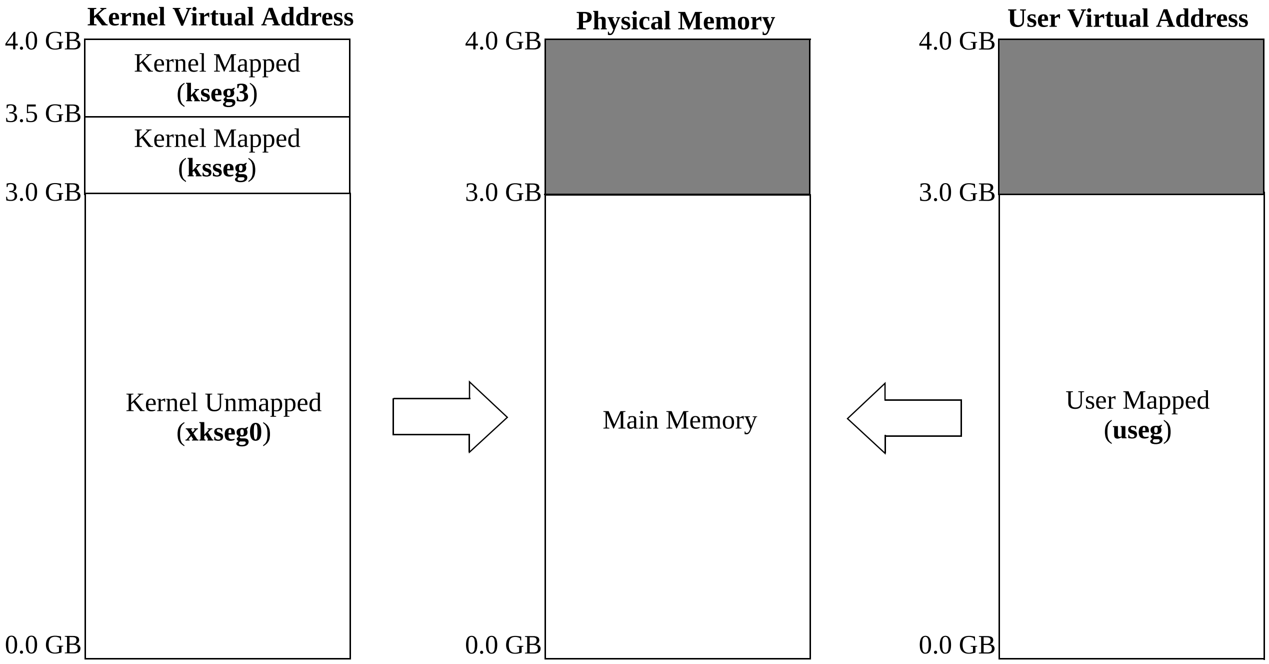


Figure 3.14 shows an example of how the traditional MIPS kernel virtual address space can be remapped using programmable memory segmentation to facilitate the EVA scheme. As a result of defining the larger kernel segment as *xkseg0*, the kernel has unmapped access to the lower 3GB of the virtual address space. The larger user segment could

[2] If necessary, *xkseg0* can be extended to 3.5GB, allowing 0.5GB for Kernel mapped virtual address space (now *kseg2*).

be defined because the address space is not statically partitioned. This allows for a total of 3.5GB of DRAM to be supported in the system.

Figure 3.14 Example of Remapping Kernel and User Virtual Address Space Using EVA



Note that *xkseg0* is equivalent to the previous *kseg0* space in that it is a kernel unmapped, cacheable region.

3.6.2 Programmable Segmentation Control

Programmable segmentation allows for the virtual address space segments to be programmed with different access modes and attributes. Control of the 4GB of virtual address space is divided into six segments that are controlled using three CP0 registers; *SegCtl0* through *SegCtl2*. Each register has two 16-bit fields. Each field controls one of the six address segments as shown in Table 3.4.

Table 3.4 Programmable Segmentation Register Interface

Register	CP0 Location	Memory Segment	Register Bits	Virtual Address Space Controlled	Virtual Address Range
SegCtl2	Register 5 Select 4	CFG5	31:16	0.0 GB to 1.0 GB	0x0000_0000 - 0x3FFF_FFFF
		CFG4	15:0	1.0 GB to 2.0 GB	0x4000_0000 - 0x7FFF_FFFF
SegCtl1	Register 5 Select 3	CFG3	31:16	2.0 GB to 2.5 GB	0x8000_0000 - 0x9FFF_FFFF
		CFG2	15:0	2.5 GB to 3.0 GB	0xA000_0000 - 0xBFFF_FFFF

Table 3.4 Programmable Segmentation Register Interface

Register	CP0 Location	Memory Segment	Register Bits	Virtual Address Space Controlled	Virtual Address Range
SegCtl0	Register 5 Select 2	CFG1	31:16	3.0 GB to 3.5 GB	0xC000_0000 - 0xDFFF_FFFF
		CFG0	15:0	3.5 GB to 4.0 GB	0xE000_0000 - 0xFFFF_FFFF

Each 16-bit field listed in the above table contains information on the corresponding memory segment such as address range (for kernel unmapped segments), access mode, and cache coherency attributes. Table 3.5 describes the 16-bit configuration fields (CFG0 - CFG5) defined in the *SegCtl0* - *SegCtl2* registers.

Table 3.5 CFG (Segment Configuration) Field Descriptions

CFGn Fields		
Name	Bits	Description
PA	15:9 and 31:25	Physical address bits 31:29 for segment, for use when unmapped. These bits are used when the virtual address space is configured as kernel unmapped to select the segment in memory to be accessed. For segments 0, 2, and 4, CFG[11:9] correspond to physical address bits 31:29. CFG[15:12] correspond to physical address bits 35:32 in a 36-bit addressing scheme and are reserved for future use. The state of CFG[15:12] are read/write and can be programmed, but these bits are not driven onto the address bus. For segments 1, 3, and 5, CFG[27:25] correspond to physical address bits 31:29. CFG[31:28] correspond to physical address bits 35:32 in a 36-bit addressing scheme and are reserved for future use. These bits are not used by the CFG4 and CFG5 spaces listed in Table 3.4 above when these segments are programmed to be kernel mapped and the physical address is determined by the TLB. They are also not used for any of the user mapped (useg) region for the same reason.
Reserved	8:7 and 24:23	Reserved.
AM	6:4 and 22:20	Access control mode. See Section 3.6.2.5, "Setting the Access Control Mode". For programmable segmentation, these bits are set as shown in Table 3.7. Bits 6:4 correspond to segments 0, 2, and 4. Bits 22:20 correspond to segments 1, 3, and 5.
EU	3 and 19	Error condition behavior. Segment becomes unmapped and uncached when $Status_{ERL}$ = 1. Bit 3 corresponds to segments 0, 2, and 4. Bit 19 corresponds to segments 1, 3, and 5.
C	2:0 and 18:16	Cache coherency attribute, for use when unmapped. For programmable segmentation, these bits are set as shown in Table 3.7. Bits 2:0 correspond to segments 0, 2, and 4. Bits 18:16 correspond to segments 1, 3, and 5.

3.6.2.1 Cache Coherency Attribute Control and the Segmentation Control Registers

The CP0 memory segmentation control registers (*SegCtl0* - *SegCtl2*) are new to the MIPS32 R3 architecture and are used to control the size and function of the various memory map segments in the proAptiv core.

In the previous generation MIPS32 R2 architecture, only the cache coherency attributes of the *kseg0* memory segment could be modified by the user. All other parameters were fixed. In the MIPS32 R3 proAptiv core, each segmentation control register (*SegCtl0* - *SegCtl2)* contains its own cache coherency attribute field to allow for maximum flexibility when assigning cacheability attributes to the memory. However, since existing code will not be aware of the existence of the *SegCtl0* - *SegCtl2* registers, the proAptiv core allows a mechanism for the cache coherency attributes (CCA) of kseg0 to be set either by the *Config*.$_{K0}$ field, as is done in the MIPS32 R2 architecture, or by the CFG3_C field (bits 18:16) of the SegCtl1 register. This allows existing code to configure virtual memory for the legacy setting.

To control where the cache coherency attributes for the memory are taken from, the *Config5*.$_K$ bit has been added to the CP0 *Config5* register. If the *Config5*.$_K$ bit is cleared, the cache coherency attributes for kseg0 are derived from the 3-bit *Config*.$_{K0}$ field of the CP0 *Config* register. This can be done when booting the proAptiv core using existing code. If the *Config5*.$_K$ bit is set, the cache coherency attributes are derived from the 3-bit *SegCtlx*.$_{CFGy\_C}$ field of the segmentation control registers (where *x* indicates the segmentation control register number 0 - 2, and *y* indicates memory segments 0 - 5). When configured for EVA, each of the six memory segments can be indivudually defined with its own cache coherency attributes. Refer to Section 2.3.3, "Memory Segmentation Registers" in the CP0 chapter for more information on the segmentation control registers.

The initial programming of *Config5*.$_K$ bit is determined by the state of the *SI_EVAReset* pin at reset as described in Section 3.6.2.3, "Setting the Memory Addressing Scheme — SI_EVAReset and CONFIG5.K".

3.6.2.2 Functions of the Config5.K Bit

The *Config5*.$_K$ bit effects the cache coherency attributes, the boot exception vector overlay mechanism, and the location of the exception vector as described below.

When the *Config5*.$_K$ bit is cleared, the following events occur:

1. The 3-bit *Config*.$_{K0}$ field is used to set the cache coherency attributes for the kseg0 region (0x8000_0000 - 0x9FFF_FFFF). See Section 3.6.2.1 above for more information.

2. Hardware creates two boot overlay segments, one for kseg0 and one for kseg1. Refer to Section 3.7.3, "Mapping of the Boot Exception Vector in the Legacy Configuration" for more information.

3. Hardware ignores the state of bits 31:30 of the *EBase* register as well as the *SI_ExceptionBase[31:30]* pins. Instead, hardware forces these bits to a value of 2'b10, causing the vectors to reside in kseg0/kseg1 space. Refer to Section 3.7, "Boot Exception Vector Relocation in Kernel Mode" for more information.

When the *Config5*.$_K$ bit is set, the following events occur:

1. The 3-bit *Config*.$_{K0}$ field is ignored and the cache coherency attributes are derived from the CFGn_C fields of the various segmentation control registers (*SegCtl0* - *SegCtl2*). Refer to Section 3.6.2.3, "Setting the Memory Addressing Scheme — SI_EVAReset and CONFIG5.K" for more information.

2. Hardware creates one boot overlay segment that can reside anywhere in virtual address space. Refer to Section 3.7, "Boot Exception Vector Relocation in Kernel Mode" for more information.

3. The exception vectors are not forced to reside in kseg0/kseg1. Rather, bits 31:30 of the *EBase* register, as well as the *SI_ExceptionBase[31:30]* signals are used to place the exception vectors anywhere within virtual address space. Refer to Section 3.14, "Exception Base Address Relocation" for more information.

3.6.2.3 Setting the Memory Addressing Scheme — SI_EVAReset and CONFIG5.K

The *SI_EVAReset* pin determines the addressing scheme and whether the device boots up in the legacy setting or the EVA setting. The legacy setting is defined as having the traditional MIPS virtual memory map used in previous generation processors. The EVA setting places the device in the enhanced virtual address configuration, where the initial size and function of each segment in the virtual memory map is determined from the segmentation control registers (*SegCtl0* - *SegCtl2*).

If the *SI_EVAReset* pin is deasserted at reset, the proAptiv core comes up in the legacy configuration and hardware takes the following actions:

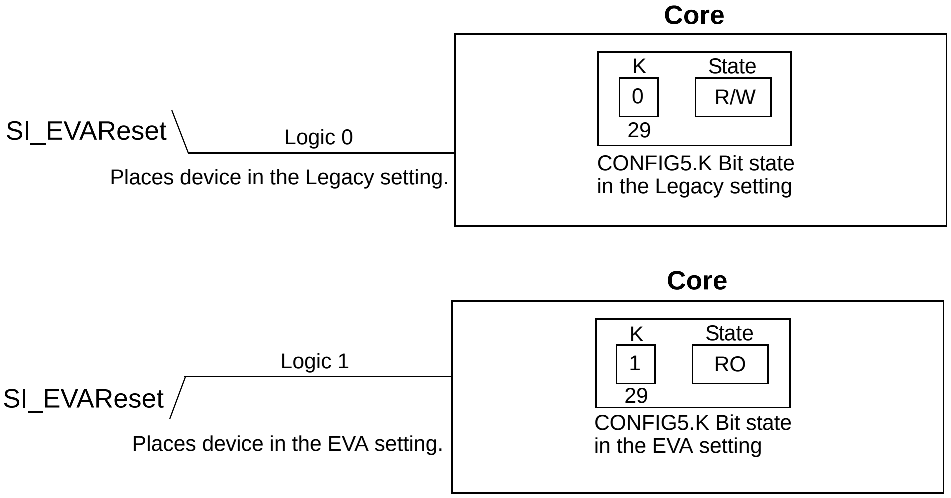
- The *CONFIG5.*$_K$ bit becomes read-write and is programmed by hardware to a value of 0 to indicate the legacy configuration. In this case, the cache coherency attributes for the kseg0 segment are derived from the *Config.*$_{K0}$ field as described in the previous subsection. In addition to selecting the location of the cache coherency attributes, the *CONFIG5.*$_K$ bit also causes hardware to generate two boot exception overlay segments, one for kseg0 and one for kseg1, as described in Section 3.7, "Boot Exception Vector Relocation in Kernel Mode".
- Hardware programs the CP0 memory segmentation registers (*SegCtl0* - *SegCtl2*) for the legacy setting. An example of this programming is shown in Table 3.13. Note that these registers are new in the proAptiv core and are not used by legacy software. However, they are used by hardware during normal operation, so their default values should not be changed.

If the *SI_EVAReset* pin is asserted at reset, the proAptiv core comes up in the EVA configuration (default is *xkseg0* space = 3 GB) and hardware takes the following actions:

- The *CONFIG5.*$_K$ bit becomes read-only and is forced to a value of 1 to indicate the EVA configuration. In this case, the *CONFIG.*$_{K0}$ field is ignored and is no longer used to determine the kseg0 cache coherency attributes (CCA). Rather, the values in bits 2:0 (segments 0, 2, and 4) and bits 18:16 (segments 1, 3, and 5) of the *SegCtl0* - *SegCtl2* registers are used to define the CCA for each memory segment as shown in Table 3.5. In this case, hardware generates only one BEV overlay segment as described in Section 3.7, "Boot Exception Vector Relocation in Kernel Mode".
- Hardware sets the CP0 memory segmentation registers (*SegCtl0* - *SegCtl2*) for the EVA configuration. An example of this programming is shown in Table 3.14.

These two options are illustrated in Figure 3.15. Refer to Section 2.3.1.6, "Device Configuration 5 — Config5 (CP0 Register 16, Select 5)" in the CP0 register chapter for more information on the *CONFIG5.*$_K$ bit.

Figure 3.15 Relationship Between SI_EVAReset and CONFIG5.K at Reset



3.6.2.4 Enhanced Virtual Address Detection and Support

As described above, the *SegCtl0* - *SegCtl2* registers are used to control the various memory segments. In addition to these registers, two other configuration registers are also used in enhanced virtual addressing (EVA).

The *EVA* bit in the *Config5* register (*Config5*$_{EVA}$) is used to detect support for the enhanced virtual address scheme. This read-only bit is always 1 to indicate support for EVA.

In addition to the *EVA* bit, the *SC* bit in the *Config3* register (*Config3*$_{SC}$) is used by hardware to detect the presence of the *SegCtl0* - *SegCtl2* registers. This read-only bit is always 1 in the proAptiv core to indicate the presence of these registers. Note that both of these features must be present to configure the virtual address space for EVA.

3.6.2.5 Setting the Access Control Mode

In addition to setting the *Config5*$_{EVA}$ and *Config3*$_{SC}$ bits described above, each memory segment must be set to the programmable segmentation mode. Bits 6:4 (segments 0, 2, and 4) and bits 22:20 (segments 1, 3, and 5) of the *SegCtl0* through *SegCtl2* registers define the access control mode.

To set the programmable segmentation registers to mimic the traditional MIPS32 virtual address mapping shown in Figure 3.13, the *AM* and *C* subfields (defined in Table 3.5) of each 16-bit *CFG* field of the *SegCtl0* - *SegCtl2* registers should be programmed as shown in Table 3.6.

Table 3.6 Setting the Access Control Mode for the Legacy Configuration

SegCtl Register	CFGn	CFGn Subfields		Segment Size	Location in Virtual Memory Map	Description
		AM	C			
0	0 (bits 15:0)	MK (bits 6:4 = 0x1)	0x3 (bits 2:0)	0.5GB	3.5 - 4.0 GB	Mapped kernel region.
0	1 (bits 31:16)	MSK (bits 22:20 = 0x2)	0x3 (bits 18:16)	0.5GB	3.0 - 3.5 GB	Mapped kernel, supervisor region.

Table 3.6 Setting the Access Control Mode for the Legacy Configuration *(continued)*

SegCtl Register	CFGn	CFGn Subfields AM	CFGn Subfields C	Segment Size	Location in Virtual Memory Map	Description
1	2 (bits 15:0)	UK (bits 6:4 = 0x0)	0x2 (bits 2:0)	0.5GB	2.5 - 3.0 GB	Kernel unmapped, uncached region.
1	3 (bits 31:16)	UK (bits 22:20 = 0x0)	0x3 (bits 18:16)	0.5GB	2.0 - 2.5 GB	Kernel unmapped, cached region.
2	4 (bits 15:0)	MUSK (bits 6:4 = 0x3)	0x3 (bits 2:0)	1.0GB	1.0 - 2.0 GB	User, supervisor, and kernel mapped region.
2	5 (bits 31:16)	MUSK (bits 22:20 = 0x3)	0x3 (bits 18:16)	1.0GB	0.0 - 1.0 GB	User, supervisor, and kernel mapped region.

To set the programmable segmentation registers to implement EVA with a 3.0 GB *xkseg0* space as shown in Figure 3.14, the *AM* and *C* subfields (defined in Table 3.5) of each *CFG* field of the *SegCtl0* - *SegCtl2* registers should be programmed as shown in Table 3.7.

Table 3.7 Setting the Access Control Mode for the EVA Configuration

SegCtl Register	CFGn	CFGn Subfields AM	CFGn Subfields C	Segment Size	Location in Virtual Memory Map	Description
0	0 (bits 15:0)	MK (bits 6:4 = 0x1)	0x3 (bits 2:0)	0.5GB	3.5 - 4.0 GB	Mapped kernel region.
0	1 (bits 31:16)	MK[1] (bits 22:20 = 0x1)	0x3 (bits 18:16)	0.5GB	3.0 - 3.5 GB	Mapped kernel region.
1	2 (bits 15:0)	MUSUK (bits 6:4 = 0x4)	0x3 (bits 2:0)	0.5GB	2.5 - 3.0 GB	Mapped user/supervisor, unmapped kernel region.
1	3 (bits 31:16)	MUSUK (bits 22:20 = 0x4)	0x3 (bits 18:16)	0.5GB	2.0 - 2.5 GB	Mapped user/supervisor, unmapped kernel region.
2	4 (bits 15:0)	MUSUK (bits 6:4 = 0x4)	0x3 (bits 2:0)	1.0GB	1.0 - 2.0 GB	Mapped user/supervisor, unmapped kernel region.
2	5 (bits 31:16)	MUSUK (bits 22:20 = 0x4)	0x3 (bits 18:16)	1.0GB	0.0 - 1.0 GB	Mapped user/supervisor, unmapped kernel region.

[1] This segment can also be mapped to MSK (bits 22:20 = 0x2) if supervisor mode is supported.

MUSUK is an acronym for *Mapped User/Supervisor, Unmapped Kernel*. This mode sets the kernel unmapped virtual address space to *xkseg0* as shown in Figure 3.14.

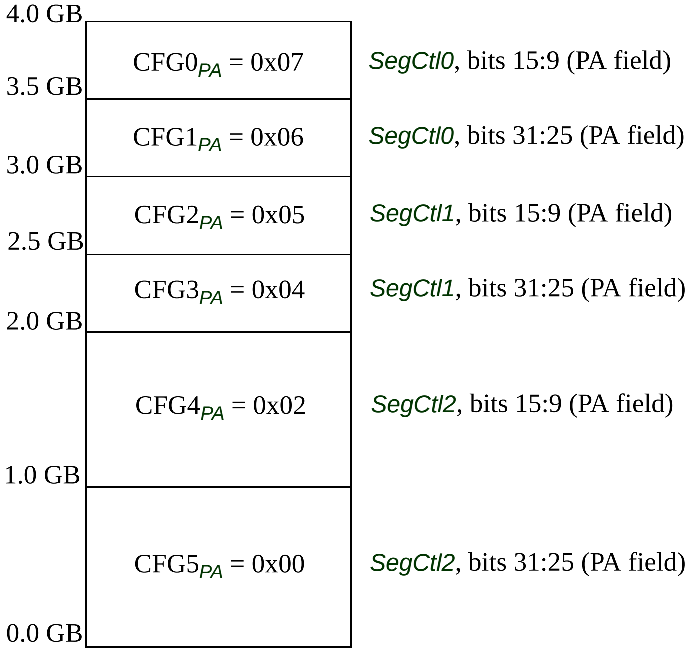
3.6.2.6 Defining the Physical Address Range for Each Memory Segment

As shown in Table 3.4, each of the six 16-bit CFGn fields of the *SegCtl0* through *SegCtl2* fields controls a specific portion of the physical address range. Bits 11:9 (segments 0, 2, and 4) and bits 27:25 (segments 1, 3, and 5) of the *SegCtl0* through *SegCtl2* registers represent the state of physical address bits 31:29 and defines the starting address of each segment. These bits control the six segments of the physical address.

Note that bits 31:28 and bits 15:12 are also part of the physical address field, but they are not used in the proAptiv core and are reserved for future use by devices that implement a 36-bit address.

Figure 3.16 below shows an example of how each segment of the physical address can be mapped to the *SegCtl0* through *SegCtl2* registers.

Figure 3.16 Mapping of SegCtl 0 - 2 Registers to Physical Address Space


4.0 GB
3.5 GB
3.0 GB
2.5 GB
2.0 GB
1.0 GB
0.0 GB
$CFG0_{PA}$ = 0x07
$CFG1_{PA}$ = 0x06
$CFG2_{PA}$ = 0x05
$CFG3_{PA}$ = 0x04
$CFG4_{PA}$ = 0x02
$CFG5_{PA}$ = 0x00
SegCtl0, bits 15:9 (PA field)
SegCtl0, bits 31:25 (PA field)
SegCtl1, bits 15:9 (PA field)
SegCtl1, bits 31:25 (PA field)
SegCtl2, bits 15:9 (PA field)
SegCtl2, bits 31:25 (PA field)

For example, to program the *xkseg0* region to a size of 3.0GB, the PA field of each register would be programmed as follows:

Table 3.8 Example of a 3.0GB Kernel Unmapped Segment

Register	CFGn Field	Bits	PA Field	Memory Segment	Virtual Address Range
SegCtl0[1]	CFG0	15:9	0x07	kseg2	0xE000_0000 - 0xFFFF_FFFF
	CFG1	31:25	0x06		0xC000_0000 - 0xDFFF_FFFF
SegCtl1	CFG2	15:9	0x05		0xA000_0000 - 0xBFFF_FFFF
	CFG3	31:25	0x04		0x8000_0000 - 0x9FFF_FFFF
SegCtl2	CFG4	15:9	0x02	xkseg0	0x4000_0000 - 0x7FFF_FFFF
	CFG5	31:25	0x00		0x0000_0000 - 0x3FFF_FFFF

1. In the 3GB xkseg0 example, the PA portion of the CFG0 and CFG1 fields are not used because they are associated with kernel mapped address spaces. In this case the PA fields are not required since the physical address is determined by the TLB. In the maximum configuration, xkseg0 can be extended to 3.5GB. In this case, the CFG1 field of the *SegCtl0* register would become part of the xkseg0 segment and the PA subfield would be used.

3.6.2.7 Enhanced Virtual Address (EVA) Instructions

By default, an implementation that supports EVA requires a number of new load/store instructions that are used when the enhanced virtual address scheme is enabled. These kernel-mode user load/store instructions allow the kernel mapped access to user address space as if it were in user mode.

For example, the kernel can copy data from user address space to kernel physical address space by using such instructions with user virtual addresses. Kernel system-calls from user space can be conveniently changed by replacing normal load/store instructions with these instructions. Switching modes (kernel to user) is an alternative but this is an issue if the same virtual address is being simultaneously used by the kernel. Further, there is a performance penalty in context-switching.

The opcode for these instructions is embedded into bits 2:0 of the instruction, known as the *Type* field. Note that some fields can have the same encoding depending whether the operation is a load or a store. The load/store designation is determined by the *AIU L/S* field, or bits 5:3 of the instruction. Table 3.9 lists the new kernel load/store instructions.

For a complete list of new instructions, refer to Section 17.7, "New Instructions for the proAptiv™ Core".

Table 3.9 Load/Store Instructions in Programmable Memory Segmentation Mode

Instruction Mnemonic	Instruction Name	Description
LBE	Load Byte Kernel	Load byte (as if user from) kernel extended virtual addressing load from user virtual memory while operating in kernel mode.
LBUE	Load Byte Unsigned Kernel	Load byte unsigned (as if user from) kernel.
LHE	Load Halfword Kernel	Load halfword (as if user from) kernel.
LHUE	Load Halfword Unsigned Kernel	Load halfword unsigned (as if user from) kernel.
LWE	Load Word Kernel	Load word (as if user from) kernel.
SBE	Store Byte Kernel	Store byte (as if user from) kernel extended virtual addressing load from user virtual memory while operating in kernel mode.
SHE	Store Halfword Kernel	Store halfword (as if user from) kernel.
SWE	Store Word Kernel	Store word (as if user from) kernel.

3.7 Boot Exception Vector Relocation in Kernel Mode

Historically in MIPS processors, the boot exception vector (BEV) has always been at the same location in both virtual and physical memory, being mapped from a virtual address of 0xBFC0_0000 to a physical address of 0x1FC0_0000. With the advent of memory segmentation in the proAptiv Multiprocessing System, the BEV vector may not always map to a physical address of 0x1FC0_0000. This can cause a scenario where the boot exception vector resides at two different physical addresses depending on the memory mode. To address this issue, the proAptiv core implements a boot exception vector overlay scheme that allows the BEV to be mapped to a single location in physical memory, regardless of the memory mode.

This section describes how to define the BEV overlay segment and the BEV relocation process for both the legacy setting and the Enhanced Virtual Address (EVA) setting, which is one element of the proAptiv memory segmentation scheme.

Note that boot exception vector relocation is performed only in Kernel mode. For more information on placing the proAptiv core in kernel mode, refer to Section 3.17.4, "Kernel Mode".

3.7.1 Boot Configurations

In kernel mode, the proAptiv Multiprocessing System can be powered up in one of two address settings:

- Legacy setting
- Enhanced Virtual Address (EVA) setting

Legacy Setting

The legacy setting is the traditional boot mode followed by all MIPS processor prior to proAptiv, where the boot exception vector (BEV) is located at 0xBFC0_0000 in virtual address space, and maps to 0x1FC0_0000 in physical address space. An example of legacy mode is described in Section 3.7.3, "Mapping of the Boot Exception Vector in the Legacy Configuration".

EVA Setting

In the EVA setting, the boot exception vector can be located anywhere in virtual address space and mapped to anywhere in physical address space. An example of an EVA configuration is described in Section 3.7.4, "Example Mapping of the Boot Exception Vector in the EVA Configuration".

For more information on configuring the proAptiv Multiprocessing System in the Legacy and EVA settings, refer to Section 3.7.5.1, "Setting the Type of Memory Addressing Mode".

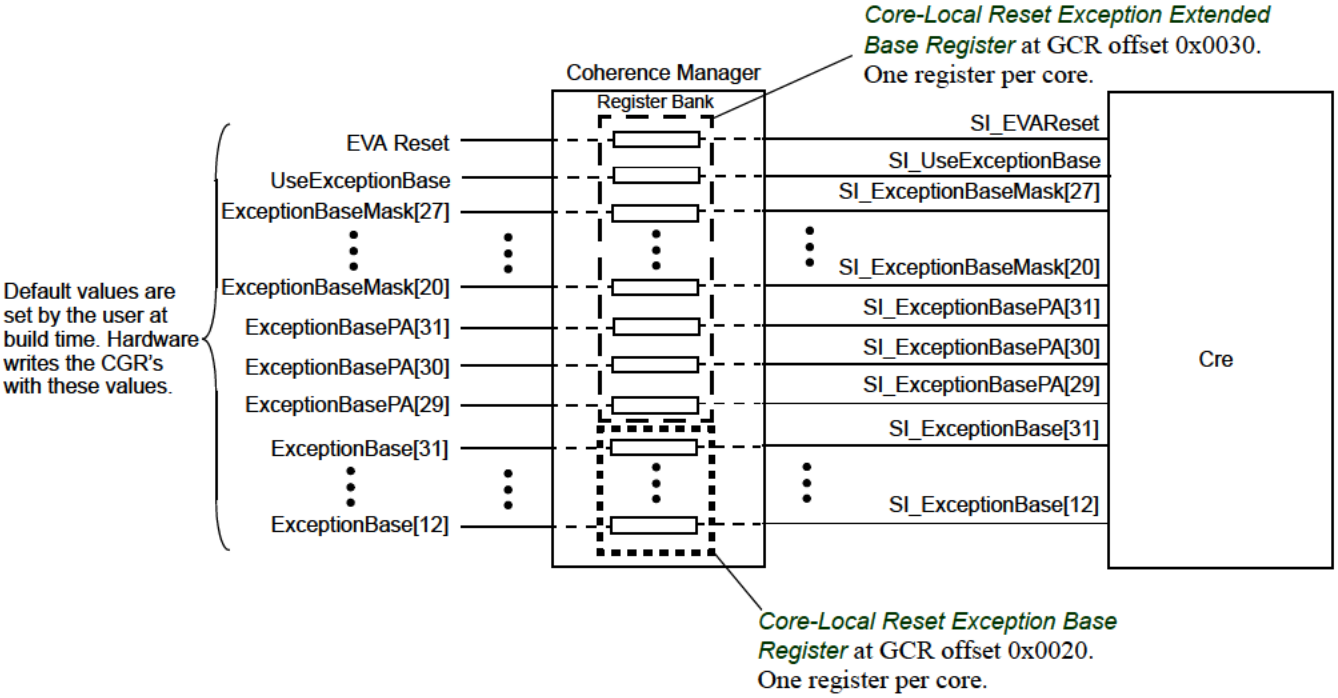
3.7.2 Pins Used to Support Boot Exception Vector Relocation

To facilitate the BEV overlay scheme, a number of pins were added to the proAptiv core that allow the user to select the boot overlay parameters at build time. The initial state of the default values selected by the user at build time are registered inside the Coherence Manager (CM2) block using two *Global Configuration Registers* (GCR). As shown in Figure 3.17, there are two GCR registers used per core. Each core has its own pair of GCR registers and its own set of BEV related pins. This allows each core to be programmed in a different manner and independently from one another.

The CM2 drives these values to the proAptiv cores at reset. Note that the two CGR registers are loaded only on a cold boot and are programmed with the values selected by the user at build time. Each of these pins is described in the following subsections.

Figure 3.17 shows the boot exception vector pins for a single proAptiv core. Each additional core would have an identical set of CM2 registers and set of BEV related pins shown in the figure.

Figure 3.17 Registered Boot Exception Vector Relocation Pins — One Core



As noted in the figure above, there is one pair of GCR registers for each core. This allows each proAptiv core to be powered up in a different memory mode and independently from one another.

The boot exception vector relocation pins are described in Table 3.10.

Table 3.10 proAptiv Boot Exception Vector Pins

Pin Name	Field Size in Bits	CM2 GCR Register Mapping	Description
SI_EVAReset	1	Bit 31 of the *Core-Local Reset Exception Extended Base Register* (offset = 0x0030)	If this pin is asserted at reset, the proAptiv core comes up in the EVA configuration. In this case the *CONFIG5.$_K$* bit becomes read-only with a fixed value of 1 to indicate EVA as the addressing scheme. In addition, the *SegCtl0 - SegCtl2* registers are configured with values that correspond to the EVA mapping. If this pin is not asserted at reset, the proAptiv core comes up in the legacy setting. In this case the *CONFIG5.$_K$* bit becomes read-write with an initial value of 0 to indicate legacy mode. This bit is modified by software when switching from legacy mode to EVA mode as described in Section 3.7.6, "Switching the Addressing Scheme from Legacy to EVA After Bootup". This pin is used in both the legacy and EVA settings. There is one *SI_EVAReset* pin per core.
SI_UseExceptionBase	1	Bit 30 of the *Core-Local Reset Exception Extended Base Register* (offset = 0x0030)	In the legacy configuration, if the *SI_UseExceptionBase* pin is not asserted, then the BEV location defaults to 0xBFC0_0000. If the *SI_UseExceptionBase* pin is asserted, address bits *SI_ExceptionBase[31:30]* are forced to a value of 2'b10 to force the BEV location into the KSEG0/ KSEG1 space. This pin is only used in the legacy configuration. There is one *SI_UseExceptionBase* pin per core.
SI_ExceptionBaseMask[27:20]	8	Bits 27:20 of the *Core-Local Reset Exception Extended Base Register* (offset = 0x0030)	Used to determine the size of the boot exception vector overlay region from 1 MB to 256 MB in powers of two. These pins are used in both the legacy and EVA configurations. There is one set of *SI_ExceptionBaseMask* pins per core.
SI_ExceptionBasePA[31:29]	3	Bits 3:1 of the *Core-Local Reset Exception Extended Base Register* (offset = 0x0030)	Upper physical address bits. The size of the overlay region defined by *SI_ExceptionBaseMask[27:20]* is remapped to a location in physical address space pointed to by the *SI_ExceptionBasePA[31:29]* pins. This allows the overlay region to be placed into one of the 512 MB segments in physical memory. These pins are used in both the legacy and EVA configurations. There is one set of *SI_ExceptionBasePA* pins per core.

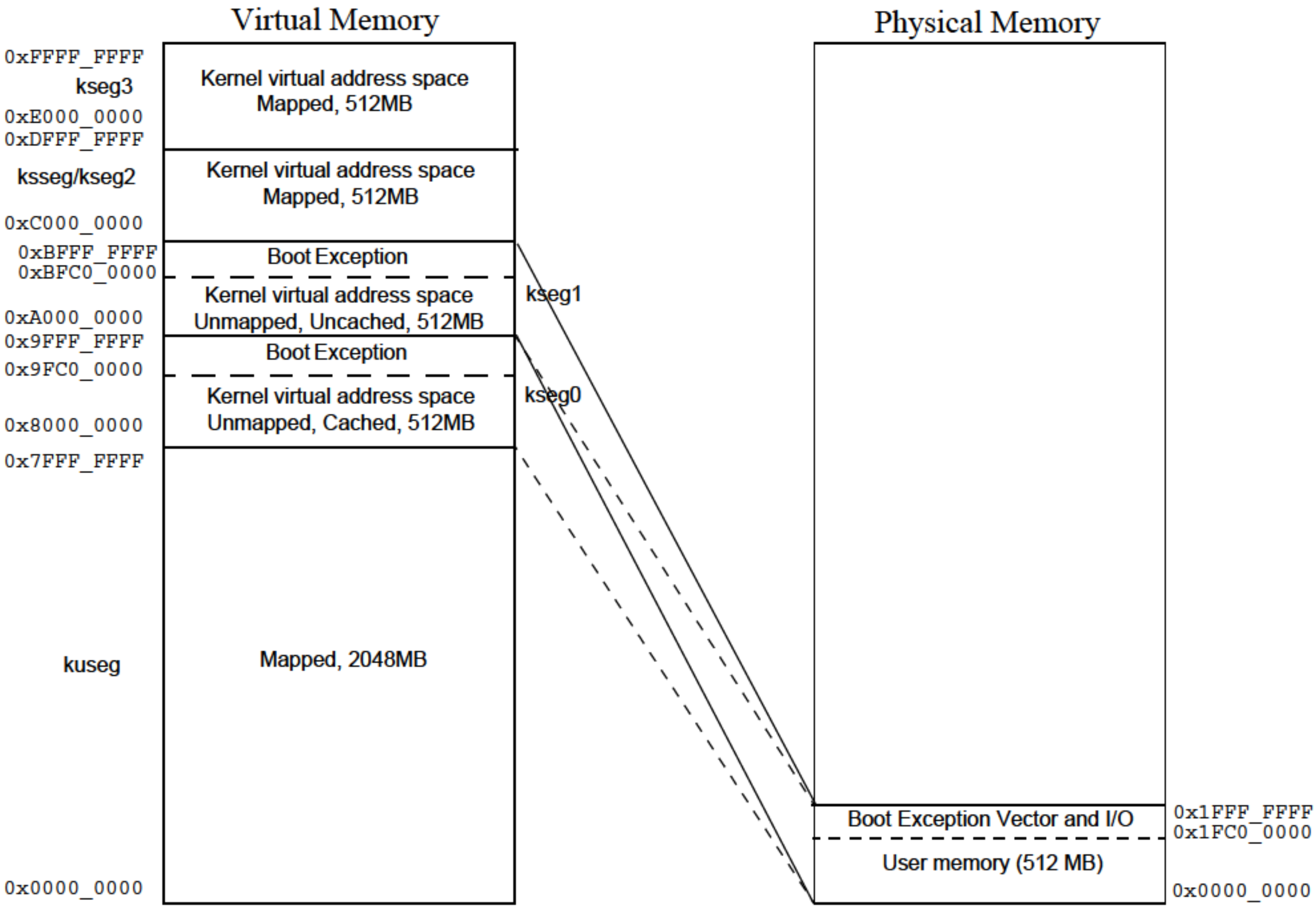
Table 3.10 proAptiv Boot Exception Vector Pins*(continued)*

Pin Name	Field Size in Bits	CM2 GCR Register Mapping	Description
SI_ExceptionBase[31:12]	20	Bits 31:12 of the *Core-Local Reset Exception Base Register* (offset = 0x0020)	The *SI_ExceptionBase[31:12]* pins define the boot address in virtual address space which is used to define the overlay region. These pins, along with the *SI_ExceptionBaseMask[27:20]* pins, determine the size and location of the BEV region within virtual address space. Note that the *CONFIG5.$_K$* CP0 register bit is used to determine which pins of the *SI_ExceptionBase[31:12]* address are used to calculate the overlay as described in Section 3.7.5.1. These pins are used in the EVA setting and can also be used in the legacy setting. There is one set of *SI_ExceptionBase* pins per core.

3.7.3 Mapping of the Boot Exception Vector in the Legacy Configuration

In all MIPS processors prior to proAptiv, the boot exception vector (BEV) was located at a virtual address of 0xBFC0_0000, and a corresponding physical address of 0x1FC0_0000. In addition, since both the Kernel Segment 1 (KSEG1) and Kernel Segment 0 (KSEG0) virtual memory spaces mapped to the same physical address space, the contents of the BEV were duplicated at a virtual address of 0x9FC0_0000. This concept is shown in Figure 3.18.

Figure 3.18 Mapping of the Boot Exception Vector in the Legacy Configuration



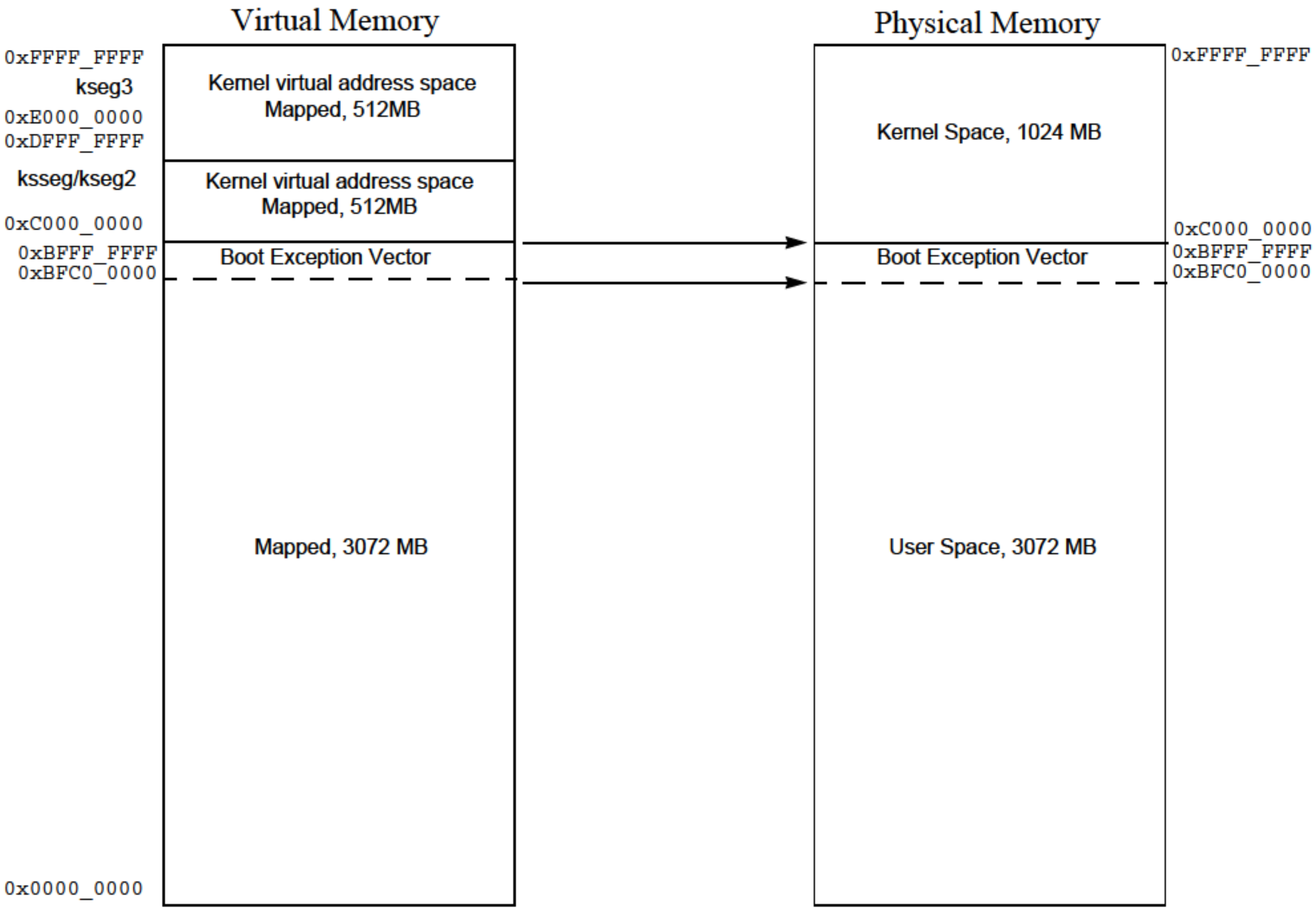
As shown in Figure 3.18 above, the default BEV location is at the top of the first 512 MB physical address space. This space typically includes not only the boot exception vector, but also the I/O memory. It is also used to store the debug exception and non-maskable interrupt (NMI) handlers. In this scheme, the top portion of memory is not available for general storage because of the existence of the boot ROM and I/O devices.

The BEV overlay scheme implemented in the proAptiv core not only ensures that the boot exception vector will be mapped to the same location regardless of the memory mode, but it also eliminates the issue of non-contiguous user memory because the BEV no longer need be in a fixed location of 0x1FC0_0000.

3.7.4 Example Mapping of the Boot Exception Vector in the EVA Configuration

In the proAptiv Multiprocessing System, physical memory sizes can be up to 3.5 GB. As described above, in the legacy configuration, the BEV is remapped from 0xBFC0_0000 in virtual memory to 0x1FC0_0000 in physical memory. However, in the EVA configuration, if the physical memory size is set to 3.0 GB, no remapping of the BEV is required. In this case, the BEV is remapped from 0xBFC0_0000 in virtual memory to 0xBFC0_0000 in physical memory. This relocation of the BEV between the two memory modes creates a conflict if software switches from the legacy configuration to the EVA configuration because the BEV will be mapped to two different locations in physical memory. The remapping of the BEV in the legacy configuration is shown in Figure 3.18 above. An example of mapping of the BEV in the EVA configuration with a 3.0 GB *xkseg0* space is shown in Figure 3.19.

Figure 3.19 Example of Mapping the Boot Exception Vector in the EVA Configuration



3.7.5 Defining the Boot Exception Vector Overlay Region

To solve the problem of having the boot exception vector residing at different physical address locations based on the memory mode, the proAptiv core defines a boot exception vector overlay region that can be programmed from 1 MB to 256 MB in powers of two using an 8-bit virtual address mask as described below.

- 1 MB, 2 MB, 4 MB, 8 MB, 16 MB, 32 MB, 64 MB, 128 MB, or 256 MB

This space is then mapped to a predetermined location in physical memory, regardless of the memory configuration (legacy or EVA).

The BEV overlay not only allows the location of the boot exception vector to be mapped to an area common to both the Legacy and EVA configuraitons, but also eliminates the non-contiguous chunk of memory that was created by having the boot exception vector at the top of the first 512M of physical memory space as in previous generation processors.

To set the boot exception overlay region, the following steps are taken. Each of these steps is described in the following subsections:

1. Determine whether the proAptiv core boots up in Legacy mode or EVA mode. This function is described in Section 3.7.5.1, "Setting the Type of Memory Addressing Mode".

2. Determine which virtual address bits will be used to calculate the boot exception vector base address. This function is described in Section 3.7.5.2, "Using the SI_UseExceptionBase Pin and CONFIG5.K to Determine How to Calculate the BEV Base Address".

3. Determine the size and location of the overlay region in virtual address space. This function is described in Section 3.7.5.3, "Determining the Size and Location of the Overlay Region in Virtual Address Space".

4. Determine the location of the overlay region in physical address space. This function is described in Section 3.7.5.4, "Determining the Location of the Overlay Region in Physical Memory".

3.7.5.1 Setting the Type of Memory Addressing Mode

The *SI_EVAReset* pin, along with the *CONFIG5.*$_K$ bit, determines whether the addressing scheme is set to legacy or EVA at reset.

Refer to Section 3.6.2.3, "Setting the Memory Addressing Scheme — SI_EVAReset and CONFIG5.K" for more information.

3.7.5.2 Using the SI_UseExceptionBase Pin and CONFIG5.K to Determine How to Calculate the BEV Base Address

The *SI_UseExceptionBase* pin and the *CONFIG5.*$_K$ register bit are also used to determine the addressing scheme and how the location of the boot exception vector will be calculated. The relationship between the *SI_UseExceptionBase* pin and the *CONFIG5.*$_K$ register is shown in Table 3.11. This table shows how to use the various address fields (*SI_ExceptionBaseMask[27:20]* and *SI_ExceptionBase[31:12]*) described in Section 3.7.5.3, "Determining the Size and Location of the Overlay Region in Virtual Address Space".

Table 3.11 *SI_UseExceptionBase* Pin and CONFIG5.K Encoding

CONFIG5.K Bit	*SI_UseExceptionBase* Pin	Condition	Action
0	0	Legacy Configuration *SI_ExceptionBase[31:12]* pins are not used.	Use default BEV location of 0xBFC0_0000.
0	1	Legacy Configuration Use only *SI_ExceptionBase[29:12]* for the BEV base location. Bits 31:30 are forced to a value of 2’b10 to put the BEV vector into KSEG0/KSEG1 virtual address space.	The BEV location is determined as follows: *SI_ExceptionBase[31:12]* = 2’b10, *SI_ExceptionBase[29:12]* pins, 12’b0 Bits 31:30 are forced to a value of 2’b10 to put the BEV vector into KSEG0/KSEG1 virtual address space.
1	Don’t care	EVA Configuration Use *SI_ExceptionBase[31:12]* pins.	The *SI_ExceptionBase[31:12]* pins are used directly to derive the BEV location. The *SI_UseExceptionBase* pin is ignored.

3.7.5.3 Determining the Size and Location of the Overlay Region in Virtual Address Space

The starting location of the overlay region in virtual address space is defined using either the *SI_ExceptionBase[31:12]* pins, or the *SI_ExceptionBase[29:12]* pins depending on the state of the *SI_UseExceptionBase* pin and *CONFIG5.$_K$* bit as described in Table 3.11 above. The size of the overlay region where the BEV is located is determined using the *SI_ExceptionBaseMask[27:20]* pins a shown in Table 3.12.

Table 3.12 Encoding of SI_ExceptionBaseMask[27:20]

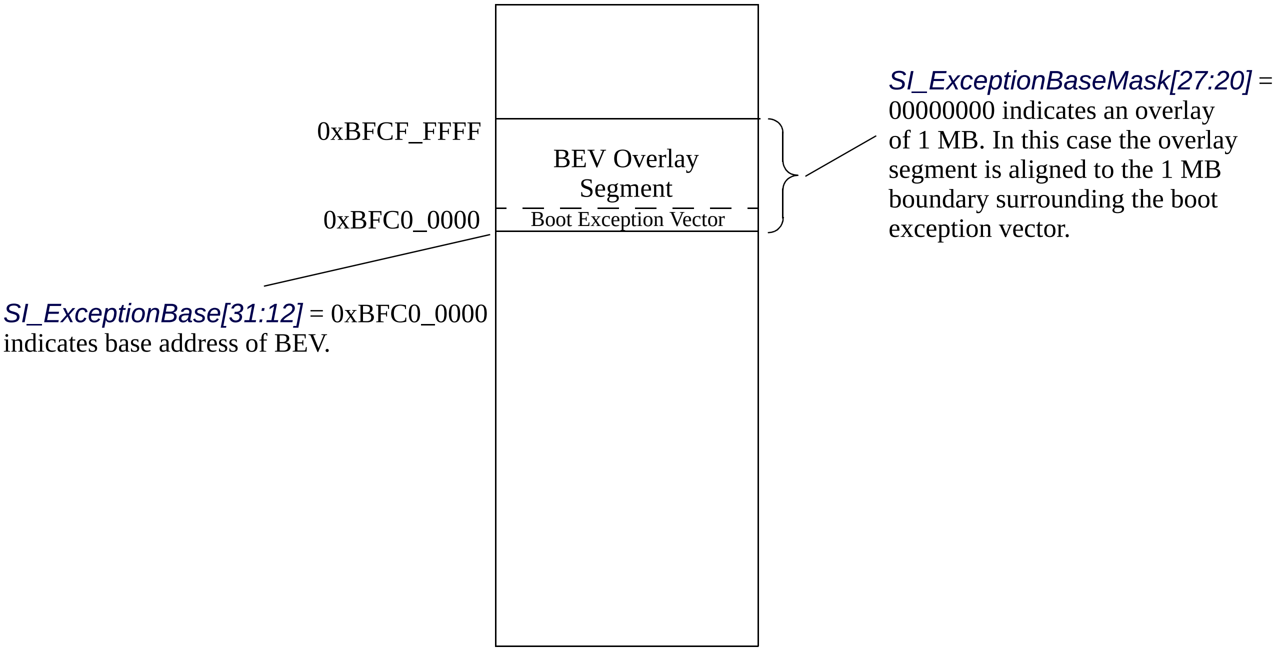
SI_ExceptionBaseMask[27:20]	Segment Size
00000000	1 MB
00000001	2 MB
00000011	4 MB
00000111	8 MB
00001111	16 MB
00011111	32 MB
00111111	64 MB
01111111	128 MB
11111111	256 MB

Consider the following example:

- The location of the BEV is at 0xBFC0_0000
- The overlay size is 1 MB (*SI_ExceptionBaseMask[27:20]* = 00000000)
- The *CONFIG5.$_K$* CP0 register bit is set

In this case the BEV segment would be located in virtual address space as shown in Figure 3.20.

Figure 3.20 Size and Location of Overlay Region in Virtual Address Space — 1 MB Example


0xBFCF_FFFF
BEV Overlay Segment
0xBFC0_0000
Boot Exception Vector
SI_ExceptionBaseMask[27:20] = 00000000 indicates an overlay of 1 MB. In this case the overlay segment is aligned to the 1 MB boundary surrounding the boot exception vector.
SI_ExceptionBase[31:12] = 0xBFC0_0000 indicates base address of BEV.

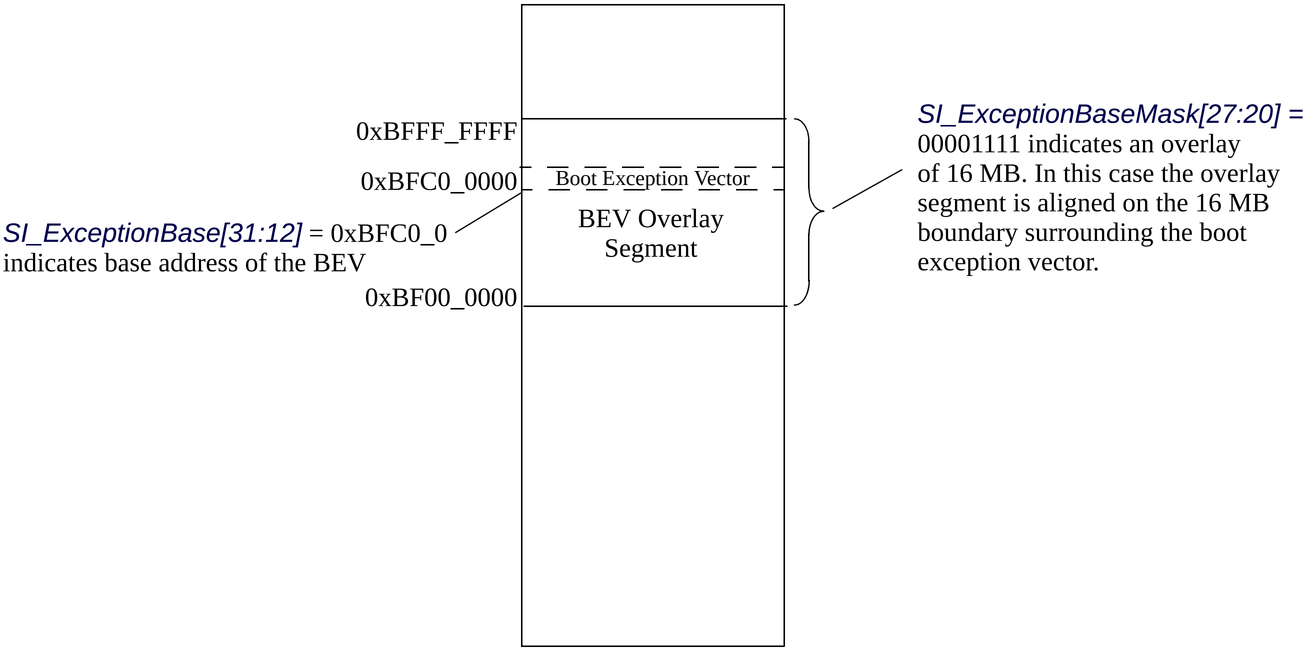
In the above example, the start of the BEV is aligned on a 1 MB boundary and therefore is at the start of the 1MB address space. This may not always be the case depending on the size of the overlay region as shown in Figure 3.21 below.

In another example:

- The location of the BEV is at 0xBFC0_0000
- The overlay size is 16 MB (*SI_ExceptionBaseMask[27:20]* = 00001111)
- The *CONFIG5.$_K$* CP0 register bit is set

In this case the BEV segment would be located in virtual address space as shown in Figure 3.21.

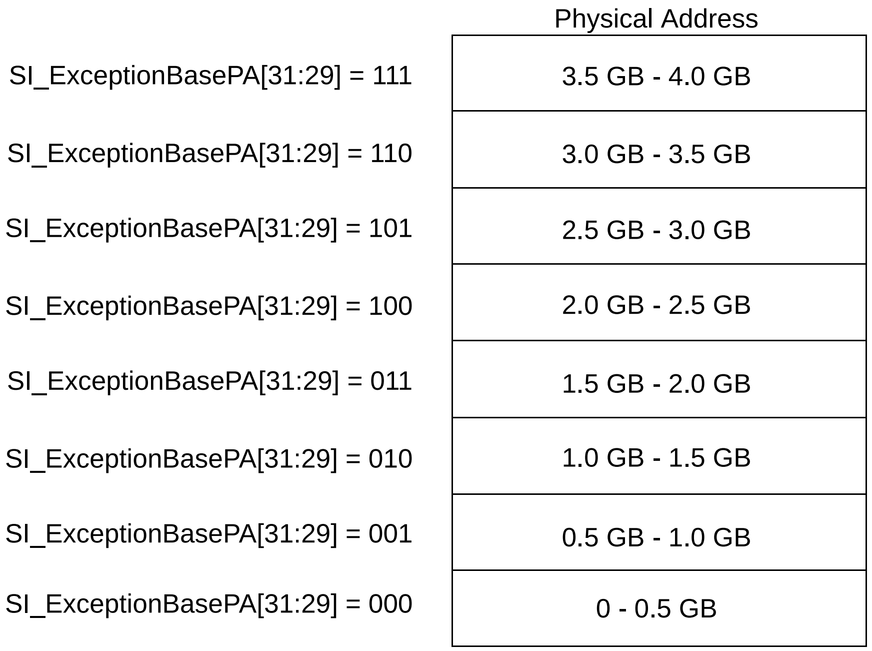
Figure 3.21 Size and Location of Overlay Region in Virtual Address Space — 16 MB Example


0xBFFF_FFFF
0xBFC0_0000
Boot Exception Vector
BEV Overlay Segment
0xBF00_0000
SI_ExceptionBase[31:12] = 0xBFC0_0 indicates base address of the BEV
SI_ExceptionBaseMask[27:20] = 00001111 indicates an overlay of 16 MB. In this case the overlay segment is aligned on the 16 MB boundary surrounding the boot exception vector.

3.7.5.4 Determining the Location of the Overlay Region in Physical Memory

As described in the previous subsections, the *SI_ExceptionBase[31:12]* and *SI_ExceptionBaseMask[27:20]* fields are used to determine the size and location of the overlay within virtual address space. This segment of virtual memory is then remapped to physical memory at a location determined by the *SI_ExceptionBasePA[31:29]* pins. These pins divide the physical address space into a number of 512 MByte segments. For example, in a 4 GB physical address space, the space can be divided into eight 512 MByte segments. This concept is shown in Figure 3.22.

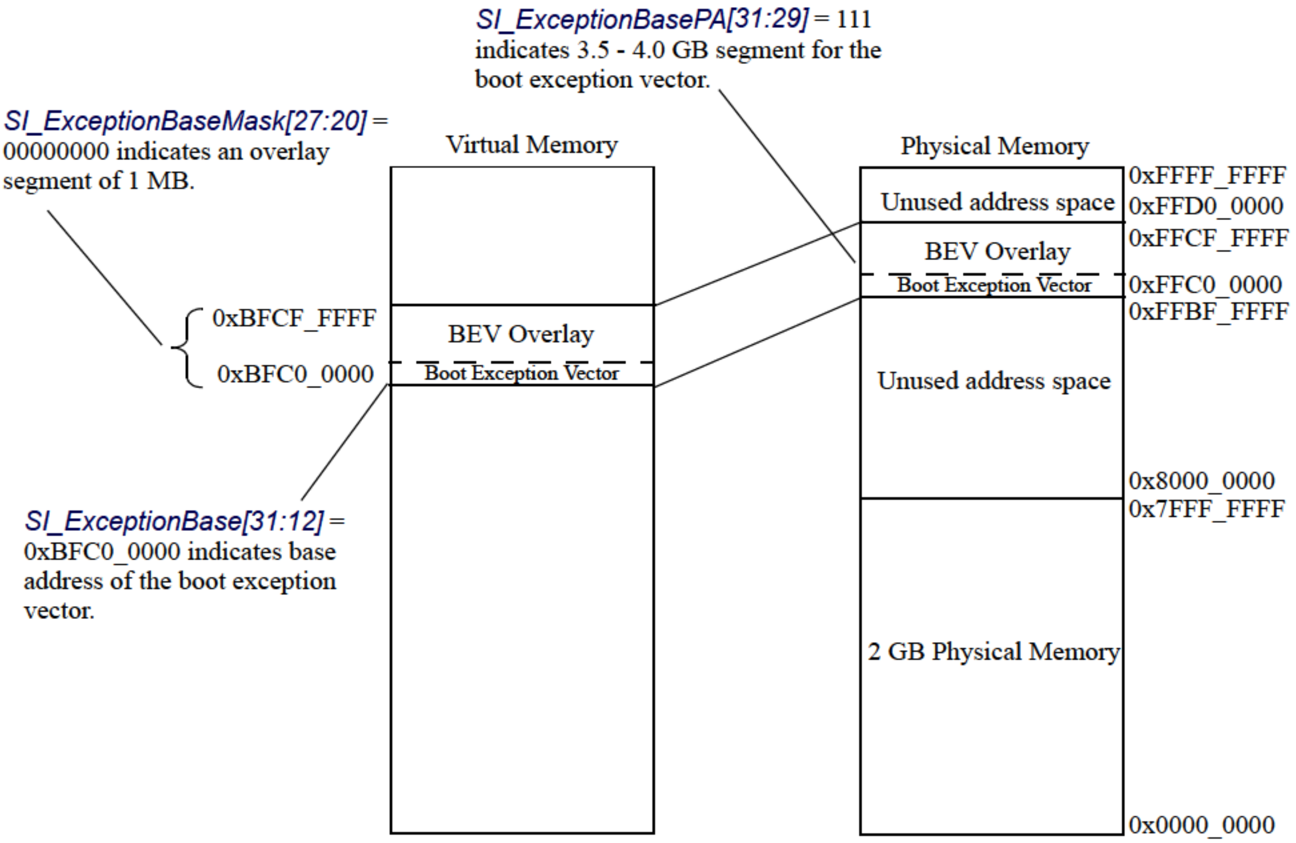
Figure 3.22 Physical Address Space Segmentation Using *SI_ExceptionBasePA[31:29]*


Physical Address
SI_ExceptionBasePA[31:29] = 111
3.5 GB - 4.0 GB
SI_ExceptionBasePA[31:29] = 110
3.0 GB - 3.5 GB
SI_ExceptionBasePA[31:29] = 101
2.5 GB - 3.0 GB
SI_ExceptionBasePA[31:29] = 100
2.0 GB - 2.5 GB
SI_ExceptionBasePA[31:29] = 011
1.5 GB - 2.0 GB
SI_ExceptionBasePA[31:29] = 010
1.0 GB - 1.5 GB
SI_ExceptionBasePA[31:29] = 001
0.5 GB - 1.0 GB
SI_ExceptionBasePA[31:29] = 000
0 - 0.5 GB

For example, assume that the boot exception vector resides at a virtual address of 0xBFC0_0000, and the size of the segment is 1 MB as determined by the *SI_ExceptionBaseMask[27:20]* pins. The physical memory size (amount of DRAM) is 2 GB, and the boot ROM that contains the BEV has been relocated to the top 512 MB of the 4 GB physical address space using the *SI_ExceptionBasePA[31:29]* pins, which selects the segment from 3.5 GB to 4.0 GB. The remapping of the boot exception vector would be as shown in Figure 3.23.

In this example, because the overlay region has been defined, the boot exception vector would be relocated to the same address space, regardless of whether the addressing scheme is legacy or EVA. In addition, the memory space that contains the BEV no longer need be shared with actual physical memory in the first 512 MB of memory space as with previous MIPS processors, thereby allowing for all of the memory to be contiguous and available to the user.

Figure 3.23 Example of Relocating the Boot Exception Vector



3.7.6 Switching the Addressing Scheme from Legacy to EVA After Boot-up

This section discusses a scenario where the processor is booted-up in the Legacy configuration, and then switches to the EVA configuration. To boot the proAptiv core in the Legacy configuration, the user selects the legacy boot configuration at build time. This causes the CM2 hardware to drive a logic ‘0’ onto the *SI_EVAReset* pin during boot time.

As shown in Figure 3.17, the default values selected by the user during build time are written into the GCR registers of the Coherence Manager (CM2). The CM2 then forwards these values to each proAptiv core in the system. In this case, the *SI_EVAReset* pin is a logic ‘0’ at boot time. This places the cores into the legacy configuration and writes a ‘0’ value into the $CONFIG5_K$ bit in CP0. The ‘registering’ of the BEV signals allows software to change the addressing scheme from legacy to EVA when necessary.

Along with clearing the $CONFIG5_K$ bit in CP0 and making the bit read/write, hardware also writes the *SegCtl0* - *SegCtl2* registers with values that mimic the legacy memory map shown in Figure 3.18. In this example, the *SegCtl0* - *SegCtl2* registers would be programmed with the following values shown in Table 3.13.

In the following tables, note that the PA field includes physical address bits 35:32. These bits are reserved for future expansion purposes and are not available in the proAptiv core. These upper four bits of each PA field are always 0 in proAptiv.

Table 3.13 SegCtl0 - SegCtl2 Register Settings in the Legacy Configuration

Register	Bits	Segment Size	Name	Definition	Reset value
SegCtl0 (CFG0)	[2:0]	0.5 GB (3.5 - 4.0 GB)	CCA	CFG0 Cache Coherency Attributes	Not defined since this is a mapped kernel region.
SegCtl0 (CFG0)	[3]		EU	CFG0 Error	1'b0: CP0 *Status*.$_{ERL}$ ignored
SegCtl0 (CFG0)	[6:4]		AM	CFG0 Region Type	3'b001: Mapped only kernel region. This is kseg3.
SegCtl0 (CFG0)	[15:9]		PA	CFG 0 Physical Address Bits [35:29]	Not defined since it is a mapped region
SegCtl0 (CFG1)	[18:16]	0.5 GB (3.0 - 3.5 GB)	CCA	CFG1 Cache Coherency Attributes	Not defined since this is a mapped kernel region.
SegCtl0 (CFG1)	[19]		EU	CFG1 Error	1'b0: CP0 *Status*.$_{ERL}$ ignored
SegCtl0 (CFG1)	[22:20]		AM	CFG1 Region Type	3'b010: Mapped Kernel/Supervisor region. This is kseg2/ksseg.
SegCtl0 (CFG1)	[31:25]		PA	CFG1 Physical Address Bits [35:29]	Not defined since it is a mapped region
SegCtl1 (CFG2)	[2:0]	0.5 GB (2.5 - 3.0 GB)	CCA	CFG2 Cache Coherency Attributes	0x2: Uncached
SegCtl1 (CFG2)	[3]		EU	CFG2 Error	1'b0: CP0 *Status*.$_{ERL}$ ignored.
SegCtl1 (CFG2)	[6:4]		AM	CFG2 Region Type	3'b000: Kernel unmapped region. This is kseg1.
SegCtl1 (CFG2)	[15:9]		PA	CFG2 Physical Address Bits [35:29]	0x0: Points to 0.0 - 0.5 GB physical address region.
SegCtl1 (CFG3)	[18:16]	0.5 GB (2.0 - 2.5 GB)	CCA	CFG3 Cache Coherency Attributes	0x3: Cacheable, noncoherent, write-back, write allocate
SegCtl1 (CFG3)	[19]		EU	CFG3 Error	1'b0: CP0 *Status*.$_{ERL}$ ignored
SegCtl1 (CFG3)	[22:20]		AM	CFG3 Region Type	3'b000: Kernel unmapped region. This is kseg0.
SegCtl1 (CFG3)	[31:25]		PA	CFG3 Physical Address Bits [35:29]	0x0: Points to 0.0 - 0.5 GB physical address region.
SegCtl2 (CFG4)	[2:0]	1.0 GB (1.0 - 2.0 GB)	CCA	CFG4 Cache Coherency Attributes	Not defined since it is a mapped region.
SegCtl2 (CFG4)	[3]		EU	CFG4 Error	1'b1: CP0 *Status*.$_{ERL}$ bit set.
SegCtl2 (CFG4)	[6:4]		AM	CFG4 Region Type	3'b011: Kernel/Supervisor/User mapped region. This is upper half of kuseg (0x4000_0000 - 0x7FFF_FFFF).
SegCtl2 (CFG4)	[15:9]		PA	CFG4 Physical Address Bits [35:29]	7’b000001x: Points to 1.0 - 2.0 GB region.

Table 3.13 SegCtl0 - SegCtl2 Register Settings in the Legacy Configuration

Register	Bits	Segment Size	Name	Definition	Reset value
SegCtl2 (CFG5)	[18:16]	1.0 GB (0.0 - 1.0 GB)	CCA	CFG5 Cache Coherency Attributes	Not defined since it is a mapped region.
SegCtl2 (CFG5)	[19]		EU	CFG5 Error	1'b1: CP0 *Status*.$_{ERL}$ bit set.
SegCtl2 (CFG5)	[22:20]		AM	CFG5 Region Type	3'b011: Kernel/Supervisor/User mapped region. This is the lower half of kuseg (0x0000_0000 - 0x3FFF_FFFF).
SegCtl2 (CFG5)	[31:25]		PA	CFG5 Physical Address Bits [35:29]	7'b000000x: Points to 0.0 - 1.0 GB region.

To make the transition from Legacy to EVA, software can execute the following steps. These steps are also called out in Figure 3.24.

1. Temporarily set bits 6:4 of the *SegCtl0* register (CP0 Register 5, Select 2) to a value of 0x0, and bits 2:0 to a value of 0x2 to set the CFG0 region as kernel unmapped, uncached. This is shown as #1 in Figure 3.24.

2. Temporarily set bits 15:9 of the *SegCtl0* register (CP0 Register 5, Select 2) to a value of 0x0 to map the CFG0 segment to a PA (physical address) value of 0x0.

3. Unpack the boot code and copy it to the CFG0 virtual address space (3.5 GB - 4.0 GB). This is shown as #3 in Figure 3.24.

4. Jump to the boot code unpacked in step 3 and continue executing the bring-up code using a PC in the range of 3.5 - 4.0 GB.

5. Initialize the caches.

6. Program all other segments other than segment 0 (since the code is currently executing out of the CFG0 segment) to the following. This is shown as #6 in Figure 3.24.

   - CFG1: AM = 0x1. Mapped Kernel (MK)
   - CFG2: AM = 0x4. Mapped User/Supervisor, Unmapped Kernel (MUSUK)
   - CFG3: AM = 0x4. Mapped User/Supervisor, Unmapped Kernel (MUSUK)
   - CFG4: AM = 0x4. Mapped User/Supervisor, Unmapped Kernel (MUSUK)
   - CFG5: AM = 0x4. Mapped User/Supervisor, Unmapped Kernel (MUSUK)

7. Set the *CONFIG5*.$_K$ bit to enable the EVA addressing scheme. Note that the Segment Control registers must be set for EVA as shown in step 6 above before the *CONFIG5*.$_K$ bit is set.

8. Unpack the kernel code in the low address space (0.0 - 1.0 GB). This is shown as #8 in Figure 3.24.

9. Jump to the kernel code that was extracted in the previous step.

10. Set the SegCtl0 register to the values shown in Table 3.14 below. This is shown as #10 in Figure 3.24.

When set for EVA, the *SegCtl0* - *SegCtl2* registers should be programmed as shown in Table 3.14 below.

Note that the exact location of the boot exception vector shown mapped to the upper addresses in EVA mode in Figure 3.24 is dependent on the state of the boot overlay pins as described in Section 3.7, "Boot Exception Vector Relocation in Kernel Mode" and all related subsections.

Figure 3.24 Mapping the Transition from Legacy Mode to EVA Mode

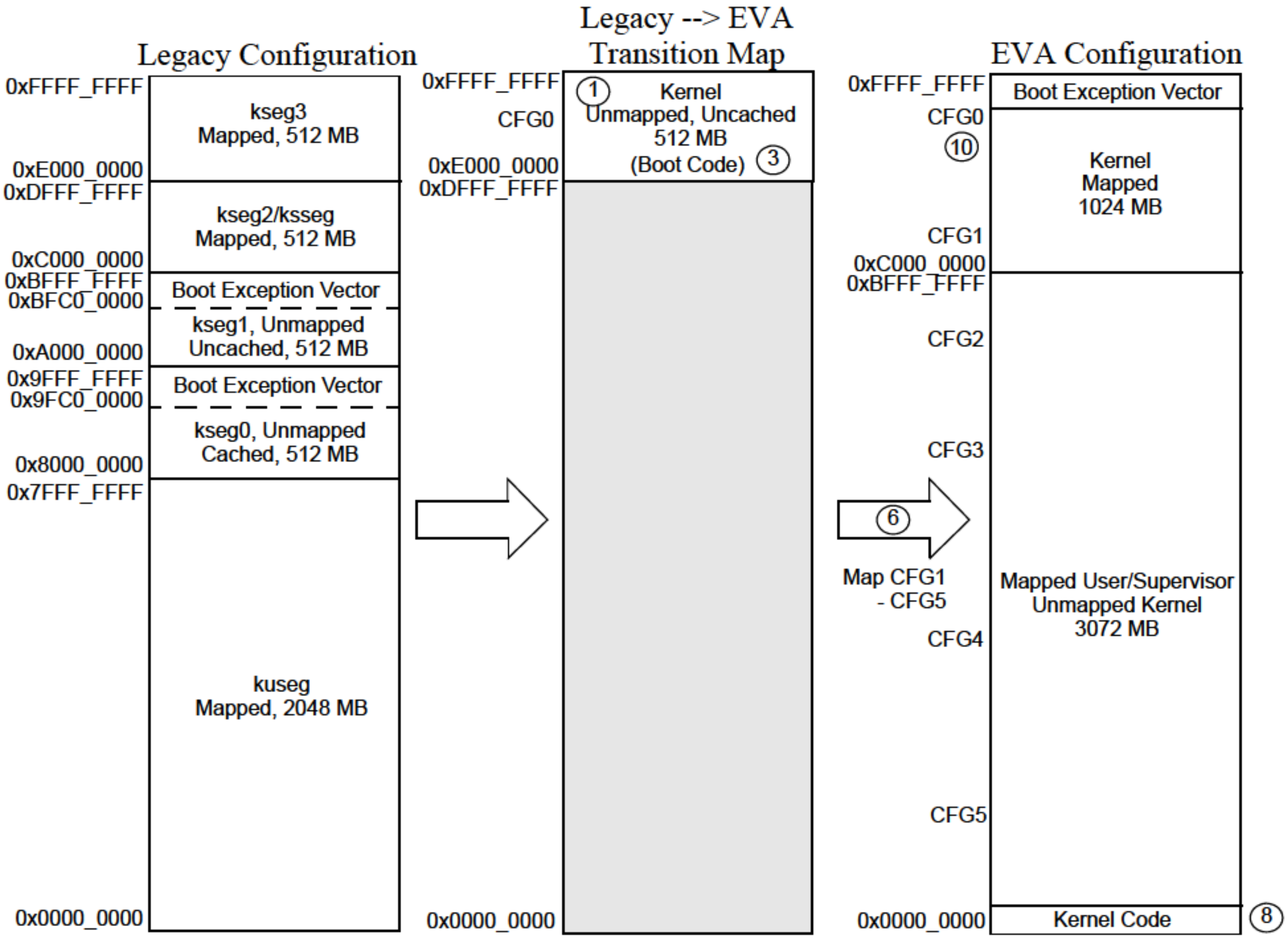


Table 3.14 New SegCtl0 - SegCtl2 Register Settings for the EVA Configuration

Register	Bits	Segment Size	Name	Definition	Reset value
SegCtl0 (CFG0)	[2:0]	0.5 GB (3.5 - 4.0 GB)	CCA	CFG0 Cache Coherency Attributes	3'bx - Not defined since it is a mapped region.
SegCtl0 (CFG0)	[3]		EU	CFG0 Error	1'b0
SegCtl0 (CFG0)	[6:4]		AM	CFG0 Region Type	3'h1 - Mapped kernel. This is the upper 512 MB of the 1024 MB mapped kernel space (3.5 - 4.0 GB).
SegCtl0 (CFG0)	[15:9]		PA	CFG 0 Physical Address Bits [35:29]	7'hx - Not defined since it is a mapped region
SegCtl0 (CFG1)	[18:16]	0.5 GB (3.0 - 3.5 GB)	CCA	CFG1 Cache Coherency Attributes	3'bx - Not defined since it is a mapped region
SegCtl0 (CFG1)	[19]		EU	CFG1 Error	1'b0
SegCtl0 (CFG1)	[22:20]		AM	CFG1 Region Type	3'h1 - Mapped kernel. This is the lower 512 MB of the 1024 MB mapped kernel space (3.0 - 3.5 GB).
SegCtl0 (CFG1)	[31:25]		PA	CFG1 Physical Address Bits [35:29]	7'hx - Not defined since it is a mapped region
SegCtl1 (CFG2)	[2:0]	0.5 GB (2.5 - 3.0 GB)	CCA	CFG2 Cache Coherency Attributes	3'b3 - WB
SegCtl1 (CFG2)	[3]		EU	CFG2 Error	1'b1
SegCtl1 (CFG2)	[6:4]		AM	CFG2 Region Type	3'b4 - MUSUK. Mapped user/supervisor, unmapped kernel. This is the upper 512 MB of the 3072 MB MUSUK space (2.5 - 3.0 GB).
SegCtl1 (CFG2)	[15:9]		PA	CFG2 Physical Address Bits [35:29]	7'h5 - 0xA000_0000 - 0xBFFF_FFFF (2.5 - 3.0 GB)
SegCtl1 (CFG3)	[18:16]	0.5 GB (2.0 - 2.5 GB)	CCA	CFG3 Cache Coherency Attributes	3'h3 - WB
SegCtl1 (CFG3)	[19]		EU	CFG3 Error	1'b1
SegCtl1 (CFG3)	[22:20]		AM	CFG3 Region Type	3'h4 - MUSUK. Mapped user/supervisor, unmapped kernel. This is the next 512 MB of the 3072 MB MUSUK space (2.0 - 2.5 GB).
SegCtl1 (CFG3)	[31:25]		PA	CFG3 Physical Address Bits [35:29]	7'h4 - 0x8000_0000 - 0x9FFF_FFFF (2.0 - 2.5 GB)

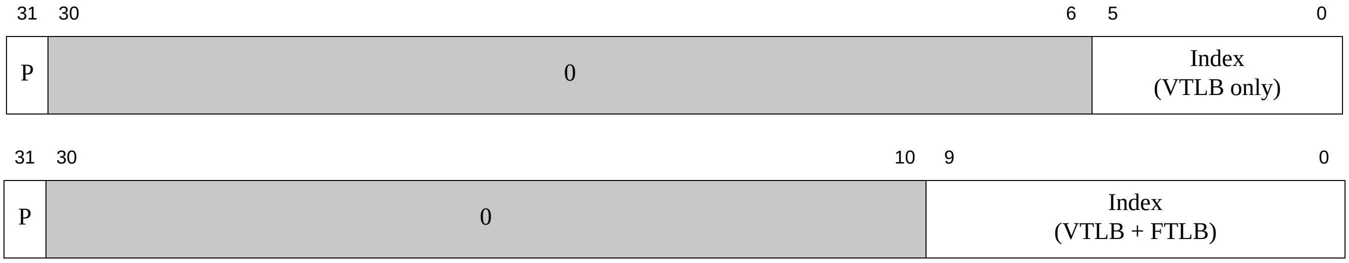
Table 3.14 New SegCtl0 - SegCtl2 Register Settings for the EVA Configuration*(continued)*

Register	Bits	Segment Size	Name	Definition	Reset value
SegCtl2 (CFG4)	[2:0]	1.0 GB (1.0 - 2.0 GB)	CCA	CFG4 Cache Coherency Attributes	3'h3 - WB
SegCtl2 (CFG4)	[3]		EU	CFG4 Error	1'b1
SegCtl2 (CFG4)	[6:4]		AM	CFG4 Region Type	3'h4 - MUSUK. Mapped user/supervisor, unmapped kernel. This is the next 1024 MB of the 3072 MB MUSUK space (1.0 - 2.0 GB).
SegCtl2 (CFG4)	[15:9]		PA	CFG4 Physical Address Bits [35:29]	7'h2 - 0x4000_0000 - 0x7FFF_FFFF (1.0 - 2.0 GB)
SegCtl2 (CFG5)	[18:16]	1.0 GB (0.0 - 1.0 GB)	CCA	CFG5 Cache Coherency Attributes	3'h3 - WB
SegCtl2 (CFG5)	[19]		EU	CFG5 Error	1'b1
SegCtl2 (CFG5)	[22:20]		AM	CFG5 Region Type	3'h4 - MUSUK. Mapped user/supervisor, unmapped kernel. This is the low-order 1024 MB of the 3072 MB MUSUK space (0.0 - 1.0 GB).
SegCtl2 (CFG5)	[31:25]		PA	CFG5 Physical Address Bits [35:29]	7'h0 - 0x0000_0000 - 0x3FFF_FFFF (0.0 - 1.0 GB)

3.8 Indexing the VTLB and FTLB

In the proAptiv core, the VTLB is 64 dual entries, and the FTLB is 512 dual entries. The FTLB is a build-time configuration option. If the FTLB is implemented and enabled, a 10-bit value is used to index all 576 dual entries of the VTLB and FTLB. If the FTLB is not implemented or not enabled, a 6-bit value is used to index the 64 dual entries of the VTLB. This is shown in Figure 3.25. This value is stored in the *Index* register (CP0 register 0, Select 0).

Figure 3.25 Index Register Format Depending on TLB Size



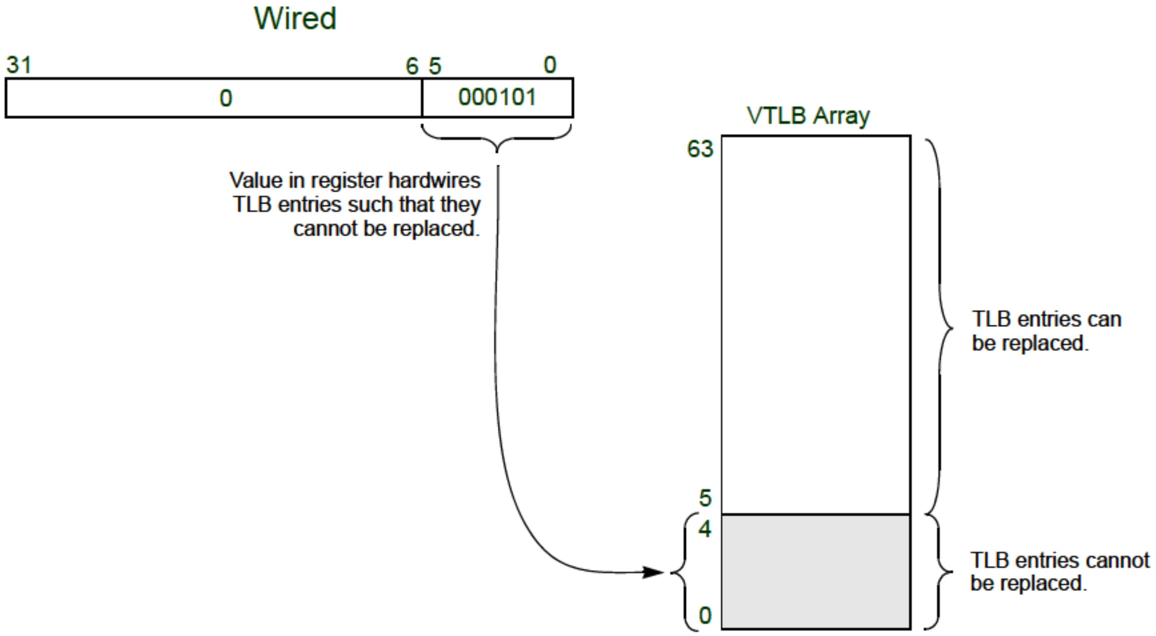
The *Index* register determines which TLB entry is accessed by a **TLBWI** instruction. This register is also used for the result of a **TLBP** instruction (used to determine whether a particular address was successfully translated by the CPU). Note that a **TLBP** instruction which fails to find a match for the specified virtual address sets bit 31 of *Index* register.

3.9 Hardwiring VTLB Entries

The proAptiv core allows up to 63 entries of the VTLB to be hardwired such that they cannot be replaced. This is accomplished using the *Wired* register (CP0 register 6, Select 0). The *Wired* register specifies the boundary between the wired and random entries in the VTLB. Wired entries are fixed, non-replaceable entries that cannot be overwritten by a **TLBWR** instruction. However, wired entries can be overwritten by a **TLBWI** instruction.

Note that wired entries in the VTLB must be contiguous and start from 0. For example, if the *Wired* field of this register contains a value of 5, this indicates that entries 4, 3, 2, 1, and 0 of the VTLB are wired. The *Wired* register is reset to zero by a Reset exception. Note that writing to the *Wired* register may cause the *Random* register to change state. Figure 3.26 shows an example of hardwiring the lower 5 entries of the VTLB. A value of 0x0 in the *Wired* register indicates that no entries are hardwired and that all entries are available for replacement.

Figure 3.26 Hardwiring Entries in the VTLB


Wired
31
6
5
0
0
000101
Value in register hardwires TLB entries such that they cannot be replaced.
VTLB Array
63
5
4
0
TLB entries can be replaced.
TLB entries cannot be replaced.

3.10 VTLB Random Replacement

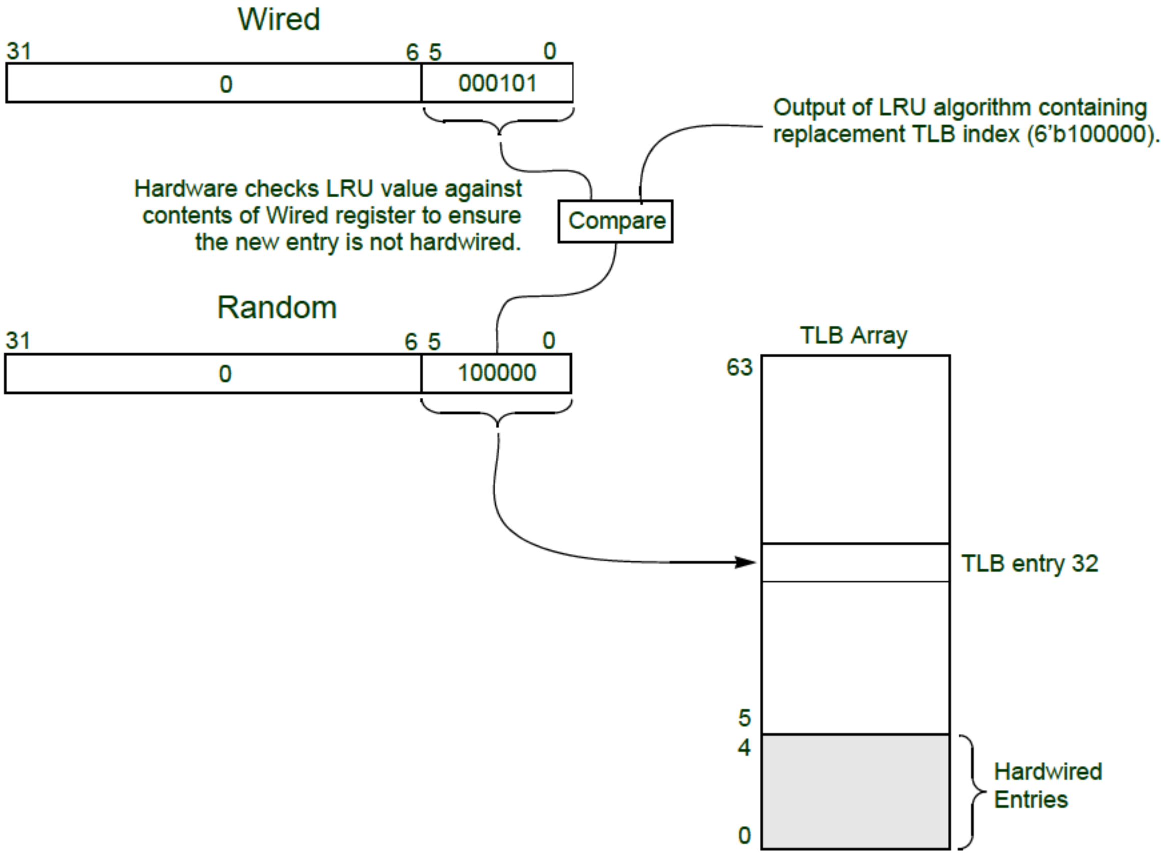
The proAptiv core performs random replacement within the 64 dual-entry VTLB using the CP0 *Random* register (CP0 register 1, Select 0). This read-only register is used to index the VTLB during a **TLBWR** instruction. It provides a quick way of replacing a VTLB entry at random.

The *Random* register employs a pseudo-random least-recently-used (LRU) algorithm which ensures that no wired entries are selected. Only those LRU entries that are not in the *Wired* register are targeted for replacement. The contents of the *Random* register are modified after a VTLB write, or on a write to the *Wired* register.

The processor initializes the *Random* register to the reflect the maximum number of entries (63) on a Reset exception. Note that the Random register is used only for VTLB accesses. It is not used on FTLB accesses as random replacement of FTLB entries is not supported.

Figure 3.27 shows an example of a random replacement to entry 32 of the VTLB with the lower five entries of the VTLB hardwired.

Figure 3.27 Random Replacement of a VTLB Entry


Wired
31
6 5
0
0
000101
Output of LRU algorithm containing replacement TLB index (6'b100000).
Hardware checks LRU value against contents of Wired register to ensure the new entry is not hardwired.
Compare
Random
31
6 5
0
0
100000
TLB Array
63
TLB entry 32
5
4
0
Hardwired Entries

3.11 FTLB Parity Errors

FTLB parity errors are reported using bits 31:28 of the CP0 *CacheErr* register (CP0, Register 27, Select 0). These read-only bits are set by hardware and are used to report errors within the L1 instruction and data caches, as well as the FTLB. An FTLB parity error can be reported for either the tag portion or the data portion of the array as shown in Table 3.15.

Table 3.15 FLTB Parity Error Reporting in the CacheErr Register

EREC (Bits 31:30)	ED (Bit 29)	ET (Bit 28)	Condition
2'b11	0	0	No FTLB errors
	0	1	FTLB Tag RAM error
	1	0	FTLB Data RAM error
	1	1	N/A[1]

[1] It is not possible to set both the ED and ET bits in the proAptiv core. Even if there are simultaneous errors in both arrays, the tag error takes precedence and the ET bit is set. In this case the data error is ignored.

Depending on the instruction being executed, hardware may or may not report a parity error for the tag and/or data array of the FTLB. Table 3.16 lists each TLB instruction and whether parity errors are logged for the data and tag arrays.

Table 3.16 FLTB Parity Error Reporting per Instruction

	Parity Error Checked?	
Instruction	**FTLB Data Array**	**FTLB Tag Array**
TLBINV	No	Yes
TLBINVF	No	No
TLBR	Yes	Yes
TLBWI	No *EntryHi*$_{EHINV}$ = 1	No *EntryHi*$_{EHINV}$ = 1
	No *EntryHi*$_{EHINV}$ = 0	Yes *EntryHi*$_{EHINV}$ = 0
TLBWR	No	Yes
TLBP	Yes	Yes
Lookup (ITLB or DTLB miss)	Yes	Yes

3.12 FTLB Hashing Scheme and the TLBWI Instruction

When a TLBWI instruction is executed, the following hashing scheme is used to calculate the FTLB index from the VPN2 field of the *EntryHi* register and the Index field of the *Index* register. This scheme is used only when the *EntryHi*$_{EHINV}$ bit is 0. When *EntryHi*$_{EHINV}$ = 1, hashing is ignored and the indexing of the FLTB is performed entirely in hardware.

When the *EntryHi*$_{EHINV}$ bit is 0, the VPN2 field in the *EntryHi* register must be consistent with the index value stored in the 10-bit *Index* field of the CP0 *Index* register. This field is used to index the total number of entries in the TLB, which equates to 64 entries in the VTLB and 512 entries in the FTLB for a total of 576 entries. To determine the size of the FTLB, hardware subtracts the VTLB size, which is always 64 entries, from the total number of entries (576) to derive an FTLB size of 512 entries. This number of entries is indexed by the lower 9 bits of the 10-bit Index field.

When the core is configured with an FTLB, the lower 9 bits of the *Index* field are organized as follows:

- Bits 6:0 = FTLB set
- Bits 8:7 = FTLB way

The FTLB set reflected in bits 6:0 of the Index field of the *Index* register (*Index*$_{Index}$) must be the same as the set number calculated from the VPN2 field of the *EntryHi* register (*EntryHi*$_{VPN2}$).

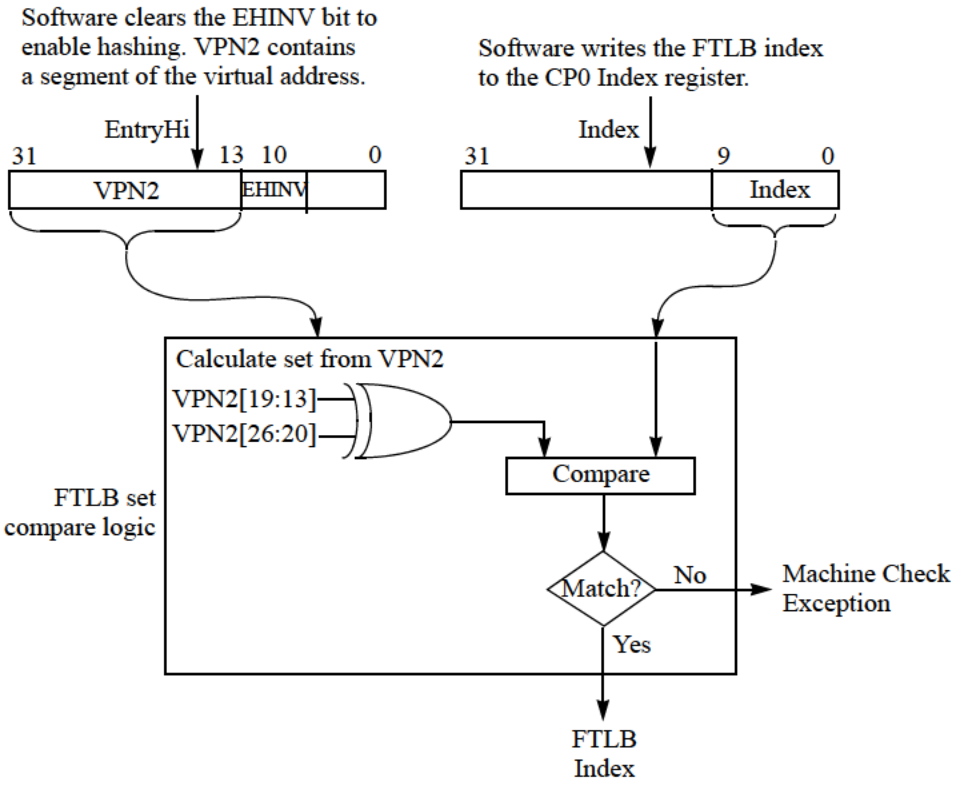
For a 4 KByte page size, the set number is calculated by performing an Exclusive OR (XOR) function of bits [26:20] and bits [19:13] of the *EntryHi*$_{VPN2}$ field.

For a 16 KByte page size, the set number is calculated by performing an Exclusive OR (XOR) function of bits [28:22] and bits [21:15] of the *EntryHi*$_{VPN2}$ field.

If the set number calculated from the *EntryHi*$_{VPN2}$ field as described above matches that stored in bits 6:0 of the *Index* register, the TLBWI instruction is allowed to continue and the FTLB is indexed. If the values do not match, a machine check exception is generated. Refer to Section 5.7.5 of the Exceptions chapter for more information on the machine check exception. Note that the TLBWR instruction does not use this hashing scheme because the indexing is performed exclusively in hardware.

The FTLB hashing scheme for a 4 KByte page size is shown in Figure 3.28. The 16 KByte page size would be identical, except for the range of VPN2 bits that are XOR'ed by hardware as described above. Note that only bits 6:0 of the Index field are compared with the calculated value. Bits 8:7 represent the FTLB way and bypass the compare operation.

Figure 3.28 FTLB Hashing Scheme During a TLB Index Write — 4 KByte Page Size



3.13 TLB Exception Handling

The proAptiv core allows for the following types of TLB exceptions.

- Address error (AdEL or AdES)
- TLB Refill
- TLB (TLBL, TLBS)
- TLB Read Inhibit (TLBRI)
- TLB Execute Inhibit (TLBXI)
- TLB Modified
- FTLB Parity

The *Address Error* exceptions (AdEL and AdES) are used in both user mode and supervisor mode.

- On a load in user mode, an *AdEL* exception is taken when the user does not have permission for the load address being accessed.
- On a store in user mode, an *AdES* exception is taken when the user does not have permission for the store address being accessed.
- On a load in supervisor mode, an *AdEL* exception is taken when the supervisor does not have permission for the load address being accessed.
- On a store in supervisor mode, an *AdES* exception is taken when the supervisor does not have permission for the store address being accessed.

The *TLB Refill* exception is taken on any TLB miss regardless of the operating mode.

The *TLB* exceptions (TLBL and TLBS) are taken under the following conditions.

- TLBL exception: On a load in any mode, there is a TLB hit, but the valid bit for that TLB entry is not set.
- TLBS exception: On a store in any mode, there is a TLB hit, but the valid bit for that TLB entry is not set.

The *TLB Read Inhibit* exception (TLBRI) is taken when there is a TLB hit during a read operation, the RI bit of the entry is set, and the $\mathit{PageGrain}_{\mathit{EIC}}$ bit is set. If the $\mathit{PageGrain}_{\mathit{EIC}}$ bit is cleared, a *TLBL* exception is taken.

The *TLB Execute Inhibit* exception (TLBXI) is taken when there is a TLB hit during an instruction fetch, the XI bit of the entry is set, and the $\mathit{PageGrain}_{\mathit{EIC}}$ bit is set. If the $\mathit{PageGrain}_{\mathit{EIC}}$ bit is cleared, a *TLBL* exception is taken.

A *TLB Modified* exception is taken whenever there is a TLB hit and the Dirty bit associated with that entry is not set.

A *FTLB Parity* exception is taken whenever a parity error occurs on an FTLB read. The FTLB parity exception is taken only when bit 31 of the CP0 *Error Control* register ($\mathit{ErrCtl}._{\mathit{PE}}$) is set. If this bit is cleared, FTLB parity errors are ignored.

Note that for the **`CacheOp`** and **`SyncI`** instructions, the TLBRI and TLBXI exceptions are not supported.

3.13.1 Overview of TLB Exception Handling Registers

The proAptiv core uses three CP0 registers to manage TLB exceptions. The exception flow in terms of these registers is described in Section 3.13.2, "TLB Exception Flow Examples".

- *Context* (CP0 register 4, Select 0): Contains the pointer to an entry in the page table entry (PTE) array.
- *ContextConfig* (CP0 register 4, Select 1): Defines the range of bits used by the *Context* register into which the high order bits of the virtual address causing the TLB exception will be written depending on the page size.
- *BadVAddr* (CP0 register 8, Select 0): Stores the virtual address that caused the exception.

3.13.1.1 Context Register

The *Context* register is a read/write register containing a pointer to an entry in the page table entry (PTE) array. When a TLB exception is taken, hardware performs the bit shifting and manipulation of the value stored in the *BadVAddr* register and places the result into the *BadVPN2* field of the *Context* register. This eliminates software from having to perform this function manually.

A TLB exception causes the virtual address to be written to a variable range of bits, defined as (X-1):Y of the *Context* register. This range corresponds to the contiguous range of set bits in the *ContextConfig* register. Bits 31:X, Y-1:0 are read/write to software and are unaffected by the exception. Software sets the *ContextConfig*$_{PTEBase}$ field to point to the base address of a page table in memory. The *ContextConfig*$_{BadVPN2}$ is derived from the virtual address associated with the exception.

Figure 3.29 shows the format of the *Context* register. Refer to Section 3.13.2, "TLB Exception Flow Examples" for more information on the usage of this register.

Figure 3.29 Context Register Format

31 … X	X-1 … Y	Y-1 … 0
PTEBase	BadVPN2	PTEBaseLow

3.13.1.2 ContextConfig Register

The *ContextConfig* register defines the bits of the *Context* register into which the high order bits of the virtual address causing a TLB exception will be written (*BadVPN2)*, and how many bits of that virtual address will be extracted. In the *Context* register, bits above the selected *BadVPN2* field are read/write to software and serve as the *PTEBase* field. Bits below the selected *BadVPN2* field serve as the *PTEBaseLow* field.

Software writes a set of contiguous ones to the *ContextConfig*$_{VirtualIndex}$ field. Hardware then determines which bits of this register are high and low. The highest order bit that is a logic '1' serves as the MSB of the *BadVPN2* field of the *Context* register. The lowest order bit that is a logic '1' serves as the LSB of the *BadVPN2* field of the *Context* register. A value of all zero's in the *VirtualIndex* field means that the full 32 bits of the *Context* register are R/W for software and are unaffected by TLB exceptions.

A value of all ones in the *ContextConfig*$_{VirtualIndex}$ field means that the full 21 bits of the faulting virtual address will be copied into the context register, making it duplicate the *BadVAddr* register. A value of all zeroes means that the full 32 bits of the *Context* register are R/W for software and unaffected by TLB exceptions.

Figure 3.30 shows the formats of the *ContextConfig* Register. Refer to Section 3.13.2, "TLB Exception Flow Examples" for more information on use of the this register.

Figure 3.30 ContextConfig Register Format

31 23	22 2	1 0
0	VirtualIndex	0

It is permissible to implement a subset of the *ContextConfig* register, in which some number of bits are read-only and set to one or zero as appropriate. It is possible for software to determine which bits are implemented by alternately writing all zeroes and all ones to the register, and reading back the resulting values. Table 3.17 describes some useful *ContextConfig* values. In this table, note that for a page table entry size of 32 bits per page, a total of 64 bits are copied from memory to support the dual-entry structure of the VTLB/FTLB. In this case, the lower 32 bits would be copied to entry 0 of the dual entry structure, and the upper 32 bits would be copied to entry 1 of the structure. The same is true for a page table with 64 bits per page. In this case, 128 bits would be fetched from memory.

Table 3.17 Example ContextConfig Values — Single Level Page Table Organization

Value	Page Table Organization	Page Size	Page Table Entry Size	Memory Structure
0x007F_FFF0	Single Level	4K	64 bits/page	128-bit
0x003F_FFF8	Single Level	4K	32 bits/page	64-bit

3.13.1.3 BadVAddr Register

The BadVAddr is a 32-bit read-only register which holds the virtual address which caused the last address-related exception. It is set for the exception types shown at the beginning of Section 3.13, "TLB Exception Handling".

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

Figure 3.31 BadVAddr Register Format

31 0
BadVAddr

3.13.2 TLB Exception Flow Examples

The following two examples show the flow of a TLB exception for the single level and dual level page table configurations.

3.13.2.1 Single Level Table Configuration

When a VTLB/FTLB error occurs, hardware writes the most recent virtual address that caused the error into bits 31:0 of the read-only *BadVAddr* register. The number of bits used by hardware to index the page table depends on the page size. For example, with a 4 KByte page size, hardware uses bits 31:13 of the *BadVAddr* register, along with the *PTEBase* field of the *Context* register, to determine the address that caused the exception.

Hardware assembles this information and places the result into the *Context* register. Use of the *Context* and *ContextConfig* registers eliminates software from having to derive the page table index manually. Depending on the page table architecture, software programs the *ContextConfig* register to indicate how many bits of the *BadVAddr* register are used by hardware to program the *Context* register. This determines the size of both the *Context*$_{BadVPN2}$ and *Context*$_{PTEBase}$ fields.

The example shown in Figure 3.32 is for a single level table configuration with a 4 KByte page size and 32 bits per page.

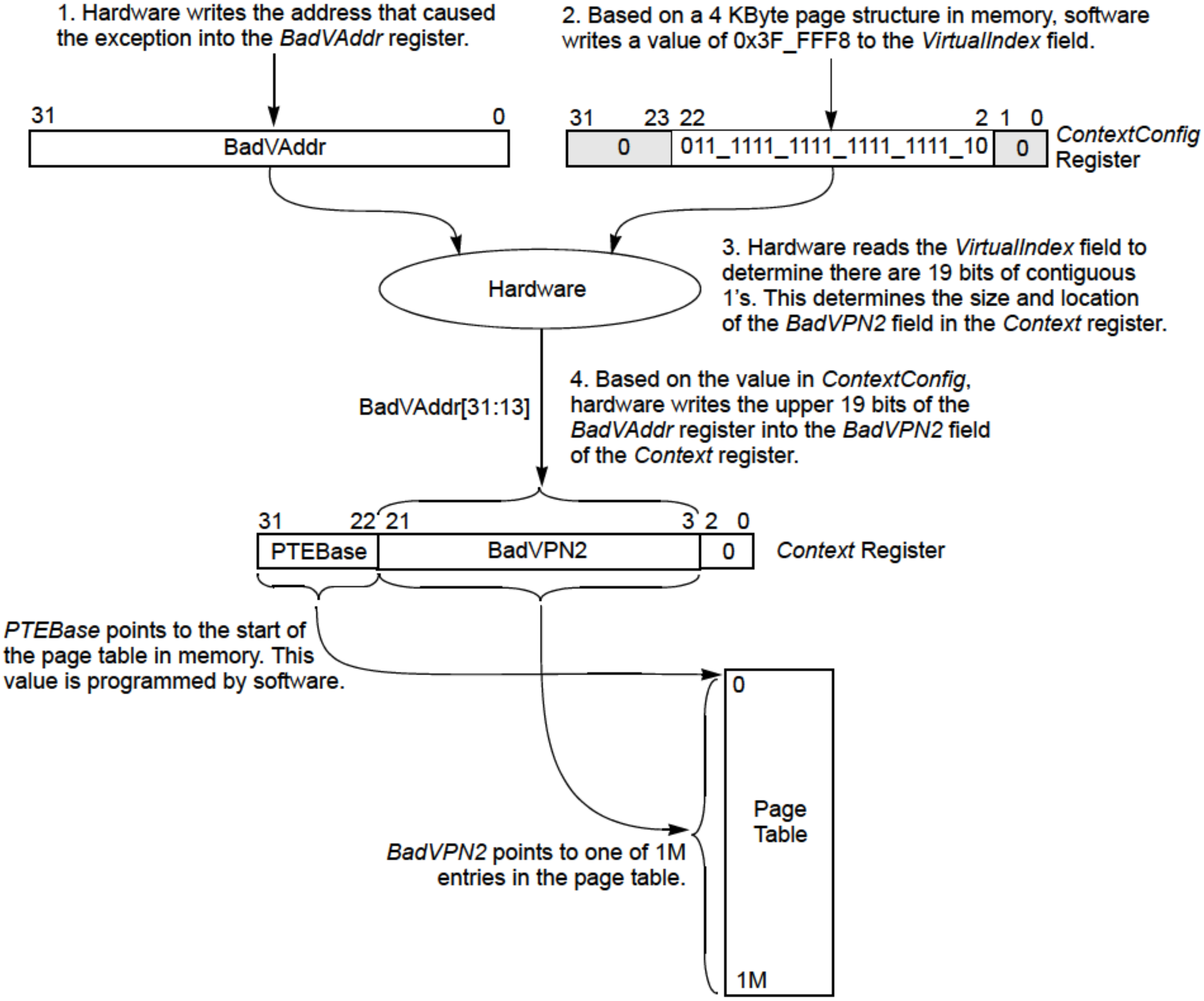
When an exception is taken, hardware writes the address that caused the exception into the *BadVAddr* register. Because the page table is single level and the page size is already known to be 4 KBytes, software programs a value of 0x3F_FFF8 into the *ContextConfig*$_{VirtualIndex}$ field. This value indicates the following information:

- The lower three bits of this value are 0, indicating that a 64-bit memory structure is being accessed. For this 64-bit value, the lower 32 bits are written to the entry 0 of the dual-entry TLB, and the upper 32 bits are written to entry 1 of the same TLB entry. Since the lower 3 bits of this field are zero, bit 3 (the first bit that is set) is used to define the low-order bit of the BadVPN2 field in the *Context* register.
- The highest-order bit that is 1 in this field is bit 21. This indicates that bit 21 is the last bit of the *BadVPN2* field in the *Context* register. As a result, the *PTEBase* field of the *Context* register occupies bits 31:22.

Based on this information, hardware assembles the value in the *Context* register as follows:

- *Context*$_{PTEBase}$ = bits 31:22. Indicates the base address of the page table in memory. This 10-bit value is a pointer to the start of the page table in memory.
- *Context*$_{BadVPN2}$ = bits 21:3. Hardware copies bits 31:13 of the *BadVAddr* register into this field. This 19-bit value is a pointer for up to 1M entries in each page table selected by the *Context*$_{PTEBase}$ field. Bits 12:0 of the *BadVAddr* register are not used in this case since the page size is 4 KBytes.
- *Context*$_{PTEBaseLow}$ = bits 2:0. Indicates access to a 64-bit memory location.

Figure 3.32 TLB Exception Flow Example — Single Level Table, 4 KB Page Size


1. Hardware writes the address that caused the exception into the *BadVAddr* register.
2. Based on a 4 KByte page structure in memory, software writes a value of 0x3F_FFF8 to the *VirtualIndex* field.
31
0
BadVAddr
31
23 22
2 1 0
0
011_1111_1111_1111_1111_10
0
ContextConfig Register
Hardware
3. Hardware reads the *VirtualIndex* field to determine there are 19 bits of contiguous 1's. This determines the size and location of the *BadVPN2* field in the *Context* register.
BadVAddr[31:13]
4. Based on the value in *ContextConfig*, hardware writes the upper 19 bits of the *BadVAddr* register into the *BadVPN2* field of the *Context* register.
31
22 21
3 2 0
PTEBase
BadVPN2
0
Context Register
PTEBase points to the start of the page table in memory. This value is programmed by software.
0
Page Table
BadVPN2 points to one of 1M entries in the page table.
1M

3.13.2.2 Dual Level Table Configuration

The TLB exception flow for a dual level page table structure is similar to that of a single level table described in Section 3.13.2.1, "Single Level Table Configuration". The upper bits of *PTEBase* are used to select the location of the first level table in memory. The *BadVPN2* field of the *Context* register is used to index the first level table and acts as a pointer to each of the second level tables in the page table array.

When a VTLB/FTLB error occurs, the most recent virtual address that caused the error is stored in bits 31:0 of the read-only *BadVAddr* register. The number of bits in the *BadVAddr* register used by hardware to index the page table depends on the page size and table organization.

Hardware assembles this information and places the result into the *Context* register. Use of the *Context* and *ContextConfig* registers eliminates software from having to derive the page table index manually. Depending on the page table architecture, software programs the *ContextConfig* register to indicate how many bits of the *BadVAddr* register are used by hardware to program the *Context* register. This determines the size of both the $Context_{BadVPN2}$ and $Context_{PTEBase}$ fields.

The example shown in Figure 3.33 is for a dual level table configuration with a 4 KByte page size and 32 bits per page.

When an exception is taken, hardware writes the address that caused the exception into the 32-bit *BadVAddr* register. Because each table in this example contains 1K entries, software programs a value of 0x00_0FFC into the $ContextConfig_{VirtualIndex}$ field. This value indicates the following information:

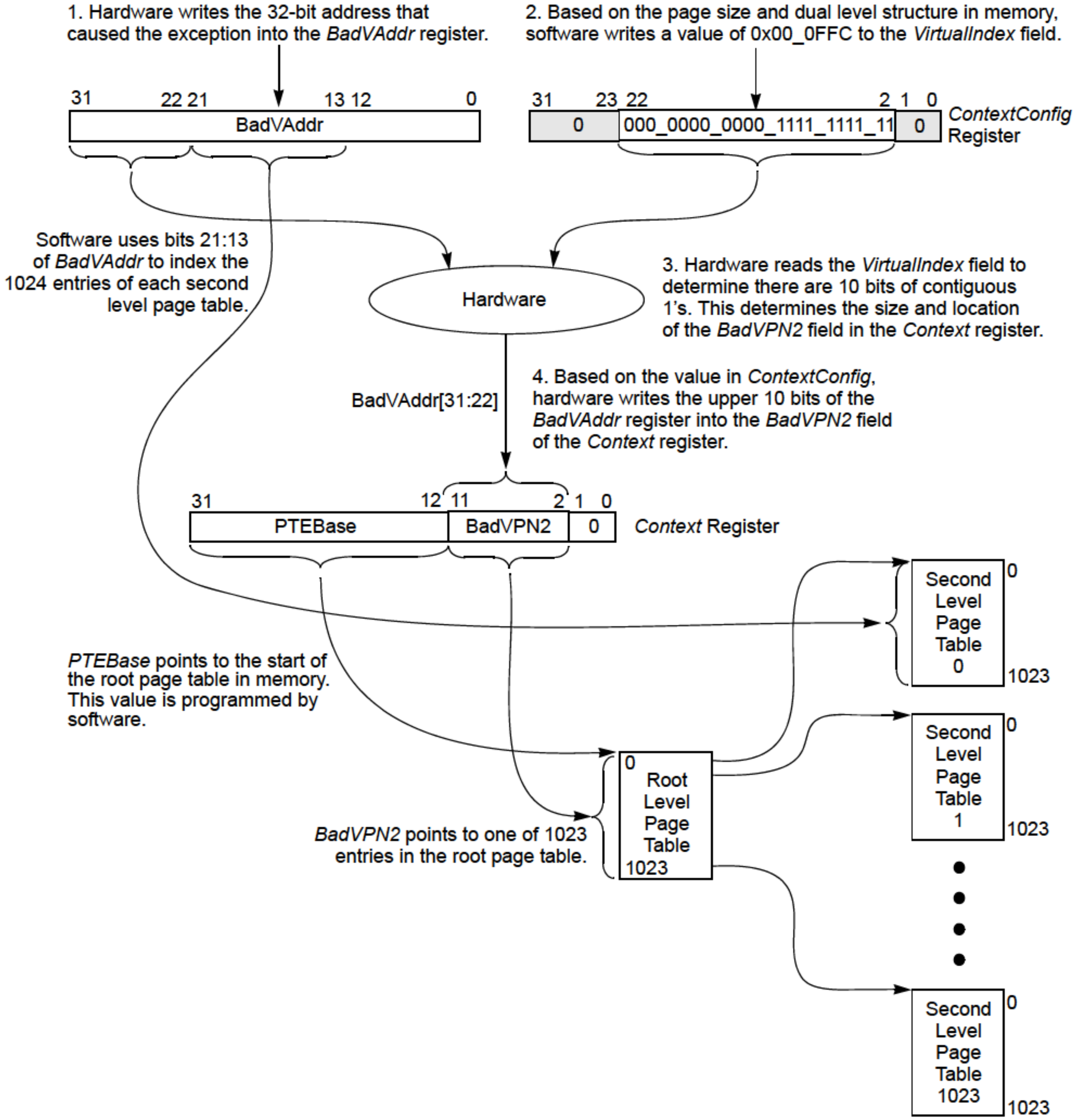
- The lower two bits of this value are 0, indicating that a 32-bit memory structure is being accessed. This also indicates that bit 2 will be the low-order bit for the $Context_{BadVPN2}$ field.
- The highest-order bit that is ‘1’ in the $ContextConfig_{VirtualIndex}$ field is bit 11. This indicates that bit 11 will be the highest-order bit of the $Context_{BadVPN2}$ field. As a result, the $Context_{PTEBase}$ field occupies bits 31:12. This field is used to access the location of the root level page table in memory.

Based on this information, hardware assembles the context register as follows:

- $Context_{PTEBase}$ = bits 31:12. Indicates the base address of the page table in memory. This 20-bit value is a pointer to the root page table in memory.
- $Context_{BadVPN2}$ = bits 11:2. Based on the state of the $ContextConfig_{VirtualIndex}$ field in this example, hardware copies bits 31:22 of the *BadVAddr* register into this field. This 10-bit value is a pointer to the 1024 entries in the root page table selected by the $Context_{PTEBase}$ field. Bits 12:0 of the *BadVAddr* register are not used in this case since the page size is 4 KBytes.
- $Context_{PTEBaseLow}$ = bits 1:0. Indicates access to a 32-bit memory location.

As stated above, bits 31:22 of the *BadVAddr* register are copied into the *BadVPN2* field of the *Context* register and are used to select one of 1024 entries in the root page table. Each of these entries acts as a pointer to one of the 1024 second level tables. Software uses bits 21:13 of the *BadVAddr* register to index one of 1024 entries in each second level page table.This concept is shown in Figure 3.33.

Figure 3.33 TLB Exception Flow Example — Dual Level Table, 4 KB Page Size



1. Hardware writes the 32-bit address that caused the exception into the *BadVAddr* register.

2. Based on the page size and dual level structure in memory, software writes a value of 0x00_0FFC to the *VirtualIndex* field.

Software uses bits 21:13 of *BadVAddr* to index the 1024 entries of each second level page table.

3. Hardware reads the *VirtualIndex* field to determine there are 10 bits of contiguous 1's. This determines the size and location of the *BadVPN2* field in the *Context* register.

4. Based on the value in *ContextConfig*, hardware writes the upper 10 bits of the *BadVAddr* register into the *BadVPN2* field of the *Context* register.

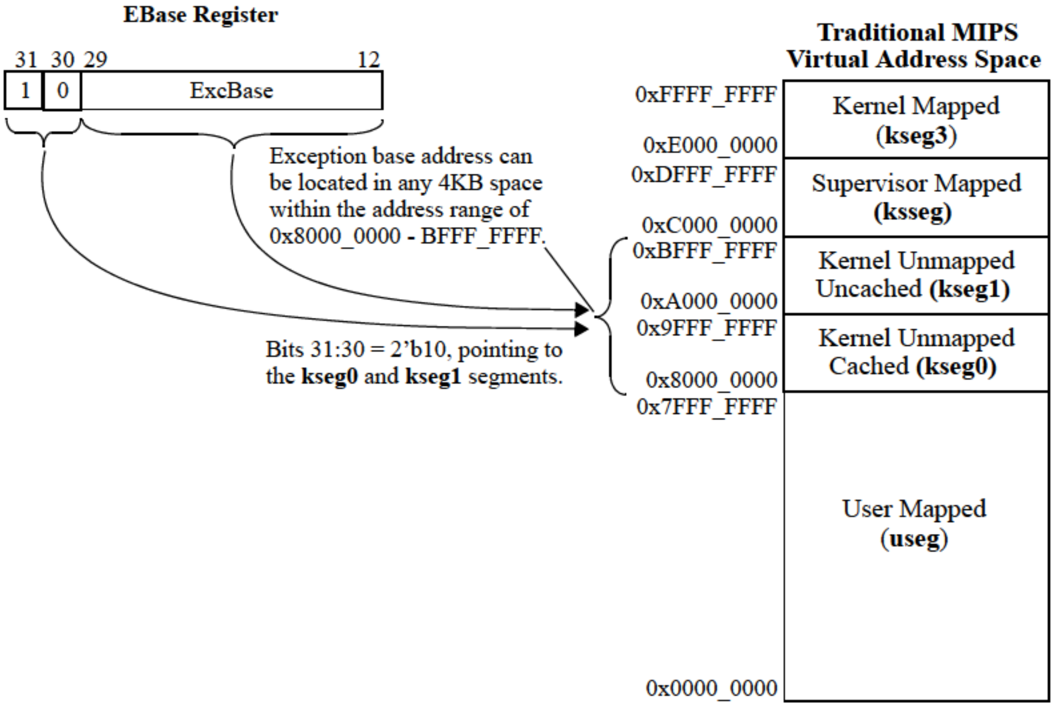
PTEBase points to the start of the root page table in memory. This value is programmed by software.

BadVPN2 points to one of 1023 entries in the root page table.

3.14 Exception Base Address Relocation

The proAptiv architecture allows the base address of an exception vector to be relocated when programmable memory segmentation is enabled. The base address of the exception is stored in the CP0 *EBase* register. In previous generation MIPS32 processors, bits 31:30 of the *EBase* Register were not writeable and had a fixed value of 2'b10 so that the exception handler would be executed from the *kseg0* or *kseg1* segments. This concept is shown in Figure 3.34.

Figure 3.34 Location of Exception Vector Base Address in Traditional MIPS Virtual Address Space



EBase Register
31 30 29 12
1
0
ExcBase
Exception base address can be located in any 4KB space within the address range of 0x8000_0000 - BFFF_FFFF.
Bits 31:30 = 2'b10, pointing to the **kseg0** and **kseg1** segments.
Traditional MIPS Virtual Address Space
0xFFFF_FFFF
0xE000_0000
0xDFFF_FFFF
0xC000_0000
0xBFFF_FFFF
0xA000_0000
0x9FFF_FFFF
0x8000_0000
0x7FFF_FFFF
0x0000_0000
Kernel Mapped (kseg3)
Supervisor Mapped (ksseg)
Kernel Unmapped Uncached (kseg1)
Kernel Unmapped Cached (kseg0)
User Mapped (useg)

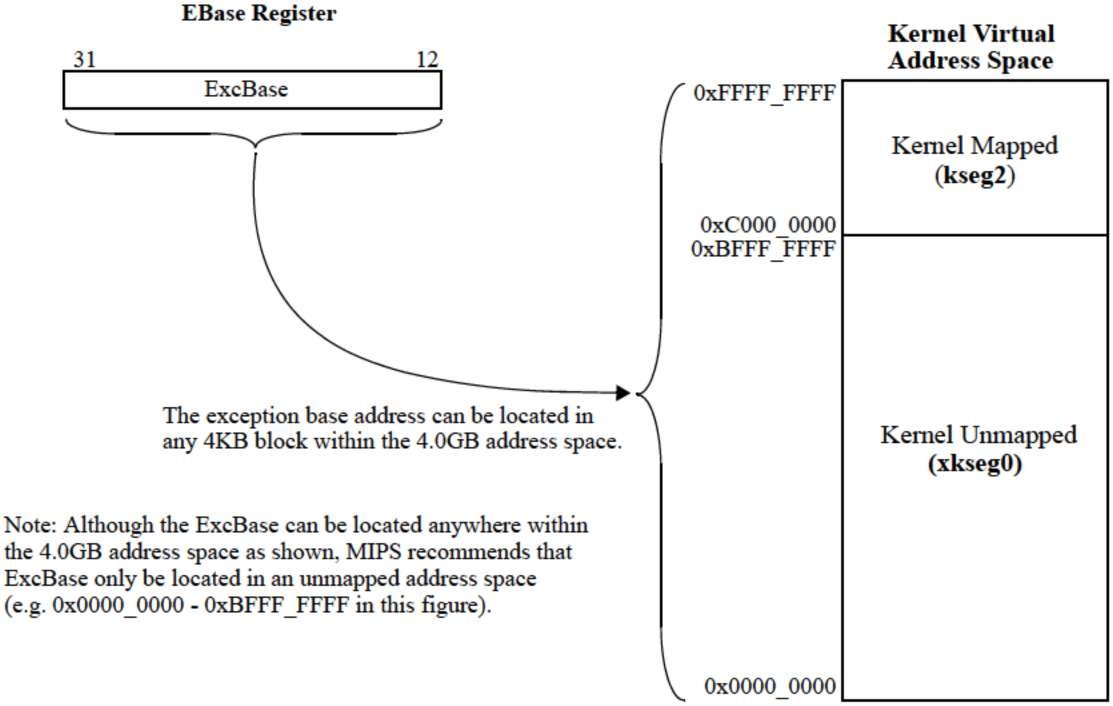
When programmable memory segmentation is enabled, the size of the exception base address is determined by the state of the *WG* bit in the CP0 *EBase* register (CP0 register 15, Select 1). At reset, the *WG* bit is cleared by default and bits 31:30 of the *EBase* Register are forced to a value of 2'b10 by hardware as described above. This is shown in Figure 3.34 above.

When the *WG* bit is set, bits 31:30 of the *ExcBase* field become writeable and are used to relocate the exception base address to other segments after they have been setup using the *SegCtl0* through *SegCtl2* registers. This is shown in Figure 3.35.

Note that if the *WG* bit is set by software (allowing bits 31:30 to become part of the *ExcBase* field) and then cleared, bits 31:30 can no longer be written by software and the state of these bits remains unchanged for any writes after *WG* was cleared. Therefore, it is the responsibility of software to write a value of 2'b10 to bits 31:30 of the *EBase* register prior to clearing the *WG* bit if it wants to ensure that future exceptions will be executed from the *kseg0* or *kseg1* segments.

Note that the *WG* bit is different from the *CV* bit in the *Config5* register. Although their functions are similar, the *CV* bit applies only to cache error exceptions, whereas the *WG* bit applies to all exceptions.

Figure 3.35 Location of Exception Vector Base Address Using the Enhanced Virtual Addressing Scheme



The exception base address can be located in any 4KB block within the 4.0GB address space.

Note: Although the ExcBase can be located anywhere within the 4.0GB address space as shown, MIPS recommends that ExcBase only be located in an unmapped address space (e.g. 0x0000_0000 - 0xBFFF_FFFF in this figure).

3.15 VTLB and FTLB Initialization

This section describes the procedure for VJTLB and FTLB initialization.

3.15.1 TLB Initialization Sequence

The following steps are used to initialize the TLB's.

1. Read the 3-bit *Config*$_{MT}$ field to determine if an FTLB is implemented. If this field is 3'b001, the FTLB is not implemented and address translation is performed only in the VTLB. If this field is 3'b100, both the VTLB and the FTLB are implemented. Refer to the *Config* register in the chapter entitled *CP0 Registers of the proAptiv Core* for more information.

2. Read the 6-bit *Config1*$_{MMUSIZE}$ field to determine the VTLB size. This field has a default of 0x3F, indicating a VTLB size of 64 entries. Refer to the *Config1* register in the chapter entitled *CP0 Registers of the proAptiv Core* for more information.

3. Read the 2-bit *Config4*$_{MMUEXDEF}$ field to determine the FTLB organization and how software should interpret bits 12:0 of this register. If this field is 2'b01, then the FTLB is not implemented, and bits 7:0 of the *Config4* register store VTLB related information. If this field is 2'b11, then the FTLB is implemented, and bits 12:0 of the *Config4* register store information relating to FTLB organization. Bits 3:0 indicate the number of FTLB ways, bits 7:4 indicate the number of FTLB sets, and bits 12:8 indicate the FTLB page size. Refer to the *Config4* register in the chapter entitled *CP0 Registers of the proAptiv Core* for more information.

4. Set the $\mathit{EntryHi}_{EHINV}$ bit to indicate that **TLBWI** invalidate is enabled. When this bit is set, the **TLBWI** instruction acts as a TLB invalidate operation, setting the hardware valid bit associated with the TLB entry to the invalid state. This bit is ignored on a **TLBWR** instruction. Refer to the *EntryHi* register in the chapter entitled *CP0 Registers of the proAptiv Core* for more information.

5. Write all zero's to the *EntryLo0* and *EntryLo1* registers to initialize them. Refer to the *EntryLo0 and EntryLo1* registers in the chapter entitled *CP0 Registers of the proAptiv Core* for more information.

6. Write the appropriate TLB size to the Index_{INDEX} field. The value written depends on whether or not an FTLB is present. If the FTLB is not present, a value of 0x3F is programmed into the lower 6 bits of this register. If the FTLB is present, a value of 0x1FF is programmed into the lower 10 bits of this register and indicates a total of 576 entries (64 VTLB + 512 FTLB). Refer to the *Index* register in the chapter entitled *CP0 Registers of the proAptiv Core* for more information.

3.15.2 TLB Initialization Code

The following code snippet can be used to initialize the VTLB and FTLB.

```
****************************************************************

/* ... at this point, t0 = index of highest tlb entry in jtlb or ftlb if present */

/*    initialize EntryHi.EHINV=1 */

        li      t1, M_EntryHiEHINV
        mtc0    t1, C0_EntryHi          # set EntryHi.EHINV=1

/* initialize EntryLo0/1 to avoid x's in simulation */

        mtc0     zero, C0_EntryLo0
        mtc0     zero, C0_EntryLo1

/*    invalidate each entry */

10:     mtc0      t0, C0_Index          # Store new index in register
        tlbwi                         # Initialize the TLB entry
        bne    t0, zero, 10b          # Loop if more to do
        addi    t0, t0, -1            # Subtract one from index field

/* clear out EHINV bit again */

    mtc0   zero,C0_EntryHi

****************************************************************
```

3.16 TLB Duplicate Entries

The VTLB/FTLB entries come up in a random state on power-up and must be initialized by hardware before use. Typically, bootstrap software initializes each entry in the TLB. Since the VTLB is a fully-associative array and entries are written by index, it is possible to load duplicate entries, where two or more entries match the same virtual address/ ASID.

If duplicate entries are detected on a TLB write, no machine check is generated and the older entries are just invalidated. The new entry gets written. When writing to the TLB, all ways of a single set in the FTLB and all the entries of the VTLB are searched for duplicates. If a large page is written to the VTLB and multiple duplicates exist for that larger page in the FTLB (multiple sets in the FTLB), then not all the duplicates are detected (and invalidated).

3.17 Modes of Operation

The MMU's virtual-to-physical address translation is determined by the mode in which the processor is operating. The proAptiv 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.

Table 3.18 Selecting the Addressing Mode

	Status			Debug	
Mode	**EXL**	**ERL**	**KSU**	**DM**	**Description**
User	0	0	2'b2	0	User addressing mode. In this mode, a TLB miss goes to the TLB Refill Handler.
Supervisor	0	0	2'b1	0	Supervisor addressing mode. In this mode, a TLB miss goes to the TLB Refill Handler.
Kernel	x	x	2'b0	0	Kernel addressing mode. In this mode, a TLB miss goes to the TLB Refill Handler.
	x	1	x	0	Kernel addressing mode. In this mode, a TLB miss goes to the TLB Refill Handler.
	1	x	x	0	Kernel addressing mode. In this mode, a TLB miss goes to the general exception handler as opposed to the TLB Refill handler.
Debug	x	x	x	1	Debug mode.

3.17.1 Virtual Memory Segments

The proAptiv architecture supports the following virtual memory schemes.

- Traditional MIPS32 virtual address space, which contains fixed address ranges for the various user and kernel segments.
- Enhanced Virtual Address (EVA) mode that allows the kernel and user address spaces to be programmed to different sizes depending on the needs of the application.

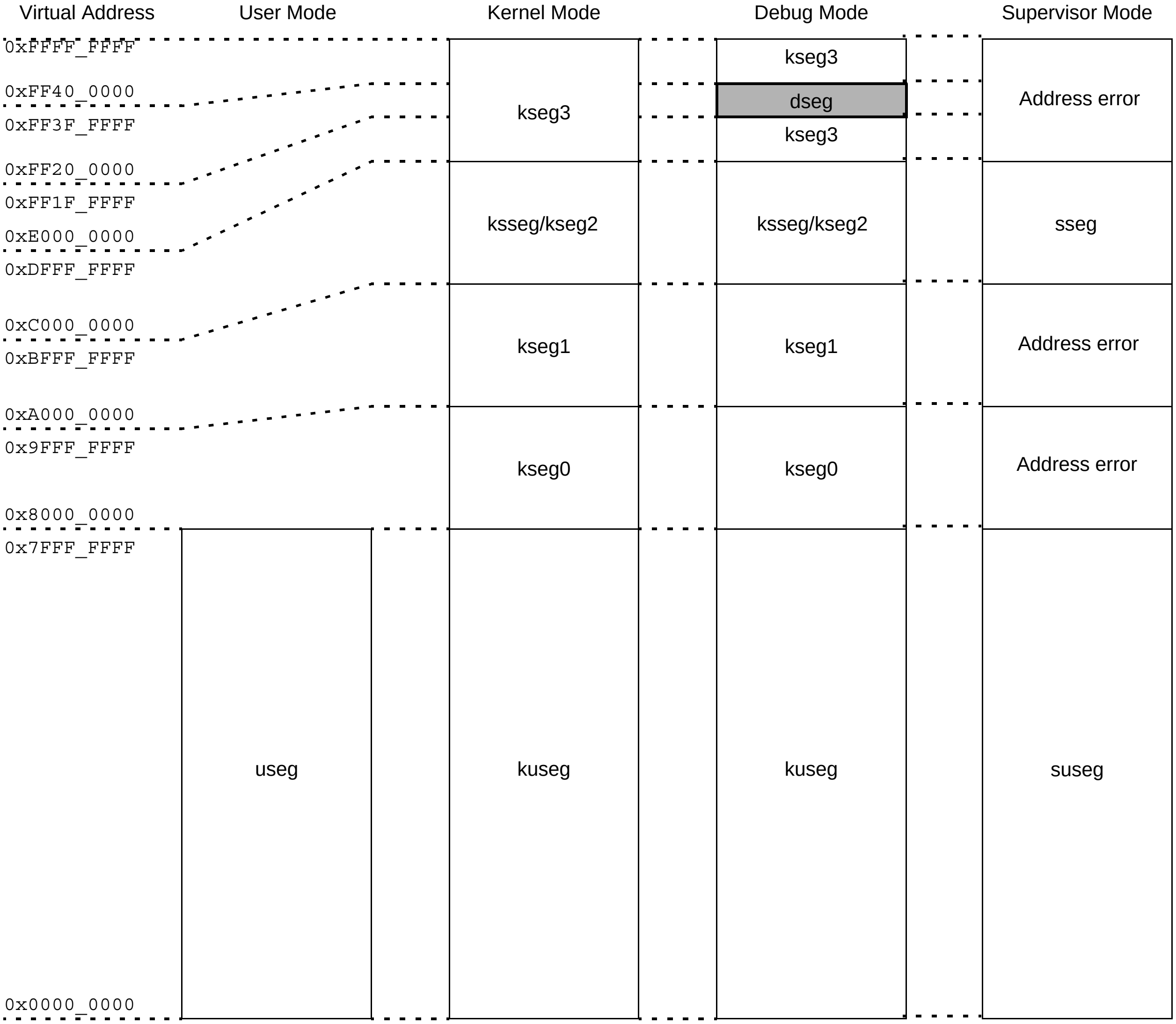
MIPS32 Virtual Address Space — Legacy Addressing Scheme

In the legacy mode, 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 3.36 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 CPU 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 3.36 Virtual Memory Map — Legacy Mode

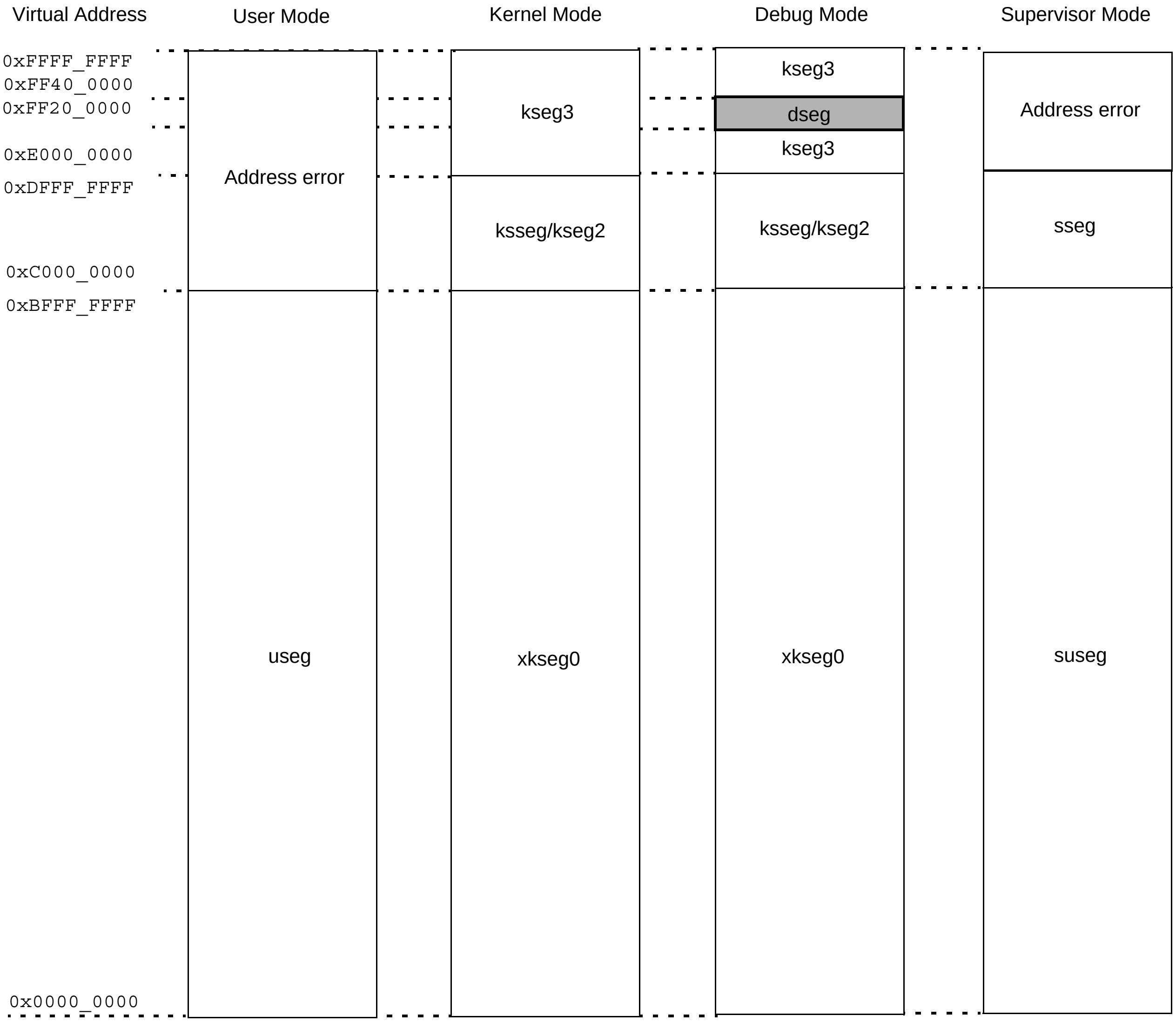


MIPS32 Virtual Address Space — EVA Addressing Scheme

In the EVA addressing scheme, the MIPS32 architecture supports a 4 GByte virtual address space that is partitioned into a number of programmable segments using the SegCtl0 through SegCtl2 registers. The EVA scheme is described in Section 3.6, "Enhanced Virtual Address". The virtual memory segments are different depending on the mode of operation and the programming of these registers.

Figure 3.37 shows an example segmentation for the 4 GByte (2^{32} bytes) virtual memory space, with the kernel and user segments being defined as 3 GB in size. Note that is only and example and the sizes of these memory segments can be increased or decreased depending on the needs of the application.

Figure 3.37 Virtual Memory Map — EVA Mode, 3GB xkseg0 Example



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

3.17.1.1 Unmapped Segments

An unmapped segment does not use the TLB 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 simple translation from virtual to physical address.

Except for kseg0, unmapped segments are always uncached. The cacheability of kseg0 is set in the K0 field of the CP0 *Config* register.

3.17.1.2 Mapped Segments

A mapped segment uses the TLB to translate from virtual to physical addresses. The translation of mapped segments are 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.

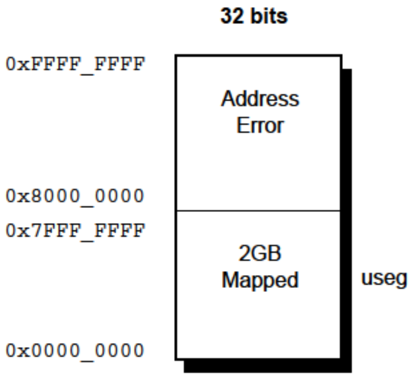
3.17.2 User Mode

In user mode, a single uniform virtual address space, called the user segment (useg), is available. The size of the user segment depends on the virtual addressing mode used.

3.17.2.1 User Mode Legacy Configuration

In the legacy mode, the user segment occupies the lower 2 GB of 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. This is shown in Figure 3.38.

Figure 3.38 User Mode Virtual Address Space — Legacy Configuration



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

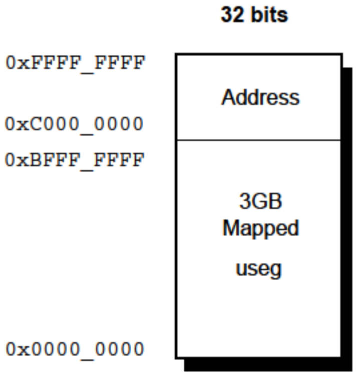
- *KSU* = 0b10
- *EXL* = 0
- *ERL* = 0

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

3.17.2.2 User Mode EVA Configuration

In EVA mode, the user segment occupies up to the lower 3.0 GB of virtual address space. The user segment starts at address 0x0000_0000 and ends at address 0xBFFF_FFFF. Accesses to all other addresses cause an address error exception. This is shown in Figure 3.39.

Figure 3.39 User Mode Virtual Address Space — EVA Mode



All references to useg are mapped through the 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.

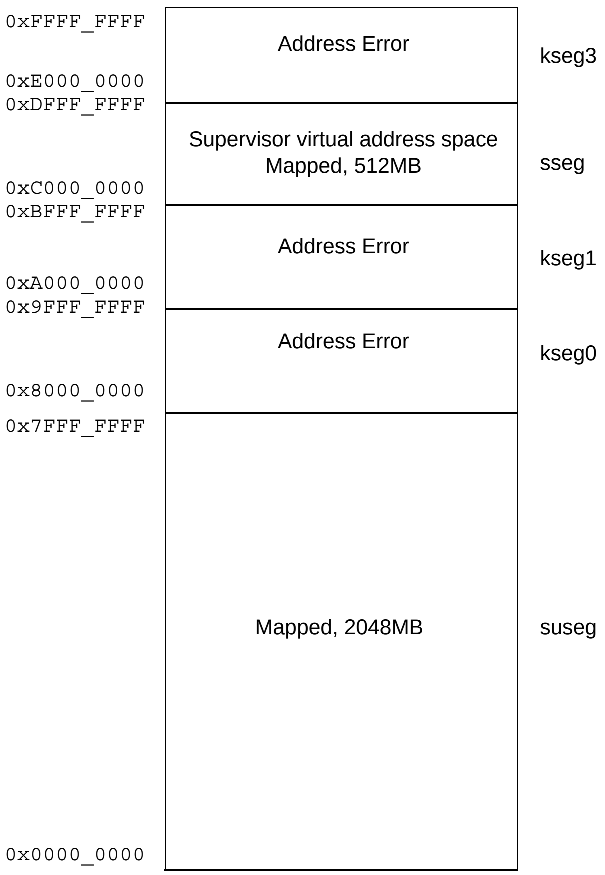
3.17.3 Supervisor Mode

In supervisor mode, two uniform virtual address spaces are available, legacy and EVA. The size of these spaces depends on the addressing mode used.

3.17.3.1 Supervisor Mode — Legacy Configuration

In the supervisor mode - legacy configuration, the 2GB 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 3.40.

Figure 3.40 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:

- *KSU* = 2’b01
- *EXL* = 0
- *ERL* = 0

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

Table 3.19 lists the characteristics of the Supervisor mode segments in the legacy.

Table 3.19 Supervisor Mode Segments — Legacy Configuration

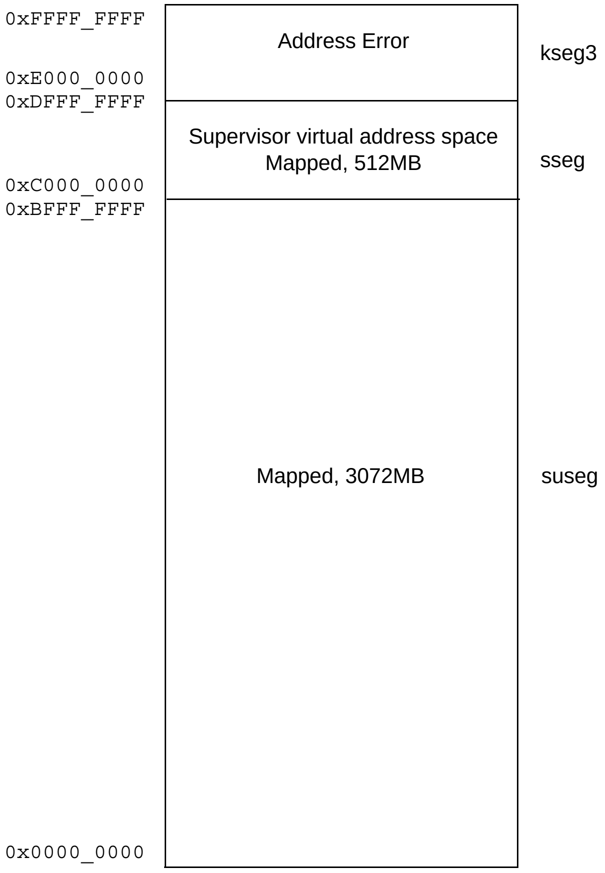
Address-Bit Value	Status Register				Segment Name	Address Range	Segment Size
	Bit Value						
	EXL	ERL	UM	SM			
32-bit A(31) = 0	0	0	0	1	suseg	0x0000_0000 --> 0x7FFF_FFFF	2 GByte (2^{31} bytes)
32-bit A(31:29) = 3’b110	0	0	0	1	sseg	0xC000_0000 -> 0xDFFF_FFFF	512MB (2^{29} bytes)

The system maps all references to s*useg and sseg* through the TLB. 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.

3.17.3.2 Supervisor Mode — EVA Configuration

In the supervisor mode - EVA configuration, the virtual address spaces are called the supervisor user segment (suseg), and the supervisor segment (sseg). The size of each field depends on the programming of the segment control registers. Figure 3.41 shows an example of a 3GB supervisor user segment (suseg), and a 512MB supervisor segment (sseg).

Figure 3.41 Supervisor Mode Virtual Address Space — EVA Configuration



The supervisor user segment begins at address 0x0000_0000 and ends at address 0xBFFF_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.

Note that the sseg segment is programmed when the bits 17:14 of the *SegCtl0* register contains a value of 0x2. This causes the address range of 0xC000_0000 to 0xDFFF_FFFF to be mapped in supervisor space. However, while in supervisor mode, where 0x0000_0000 - 0xBFFF_FFFF is defined as the suseg segment, the 0xC000_0000 to 0xDFFF_FFFF address range can be configured as kernel mapped. This occurs when 17:14 of the *SegCtl0* register contains a value of 0x1. Refer to Table 3.6 and Table 3.7 for more information.

3.17.4 Kernel Mode

In kernel mode, two uniform virtual address spaces are available, legacy and EVA. The size of these spaces depends on the addressing mode used as described below.

3.17.4.1 Kernel Mode — Legacy Configuration

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

- *KSU* = 2’b00
- *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 3.42. The characteristics of kernel-mode segments are listed in Table 3.20.

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

Figure 3.42 Kernel Mode Virtual Address Space — Legacy Configuration

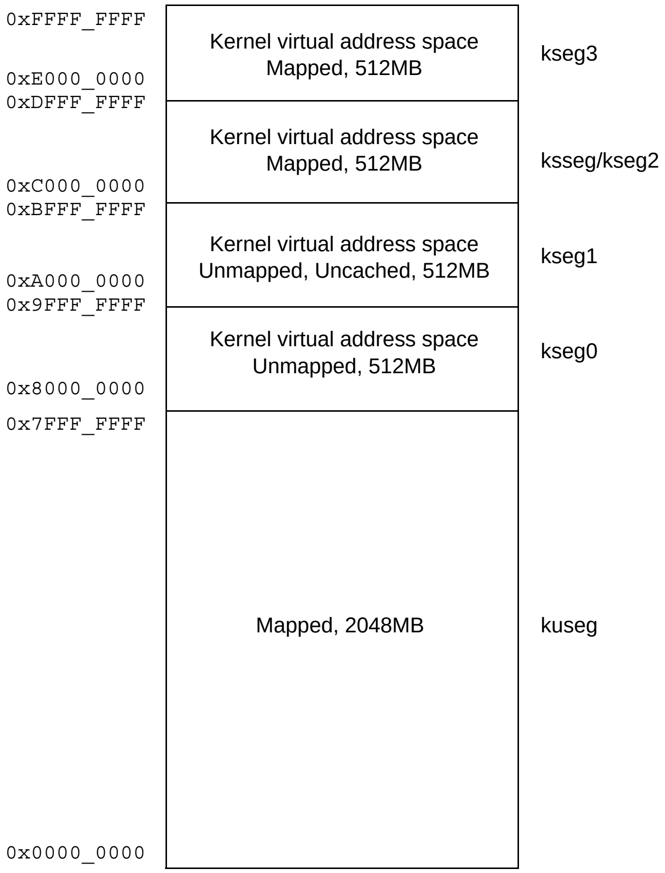


Table 3.20 Kernel Mode Segments

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) = 3’b100				kseg0	0x8000_0000 through 0x9FFF_FFFF	512 MBytes (2^{29} bytes)
A(31:29) = 3’b101				kseg1	0xA000_0000 through 0xBFFF_FFFF	512 MBytes (2^{29} bytes)
A(31:29) = 3’b110				ksseg/kseg2	0xC000_0000 through 0xDFFF_FFFF	512 MBytes (2^{29} bytes)
A(31:29) = 3’b111				kseg3	0xE000_0000 through 0xFFFF_FFFF	512 MBytes (2^{29} bytes)

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.

Kernel Mode, Kernel Space 0 (kseg0)

In Kernel mode, when the most-significant three bits of the virtual address are 3'b100, 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.

Kernel Mode, Kernel Space 1 (kseg1)

In Kernel mode, when the most-significant three bits of the 32-bit virtual address are 3'b101, 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.

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

In Kernel mode, when *KSU* = 2'b00, *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 3'b110, 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.

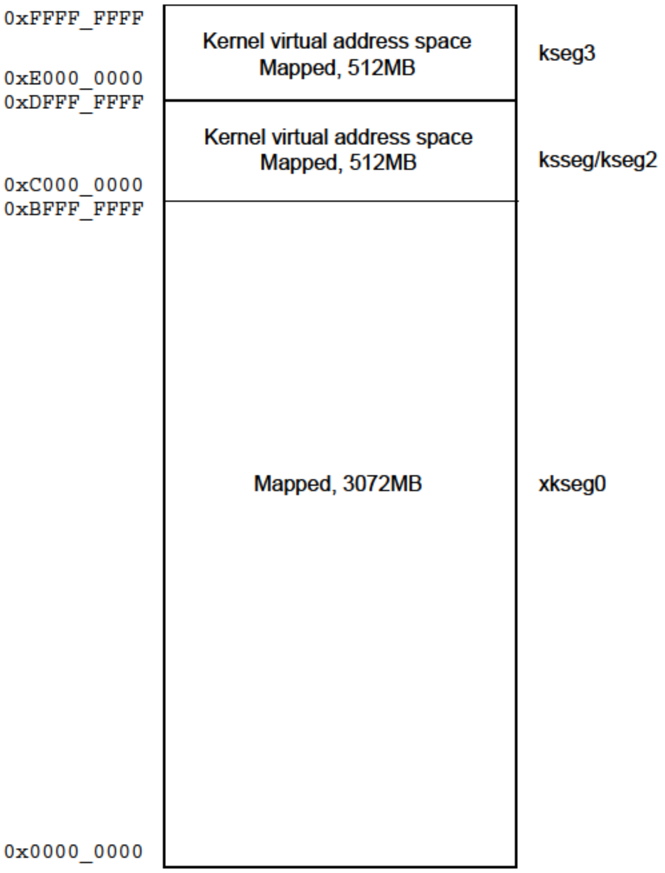
Kernel Mode, Kernel Space 3 (kseg3)

In Kernel mode, when the most-significant three bits of the 32-bit virtual address are 3'b111, 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.

3.17.4.2 Kernel Mode — EVA Configuration

In the kernel mode - EVA configuration, the size of each kernel virtual address segment depends on the programming of the segment control registers. Figure 3.41 shows an example of a 3GB xkseg0 segment (suseg), a 512MB kernel supervisor segment (ksseg), and a 512MB kernel segment 3 (kseg3).

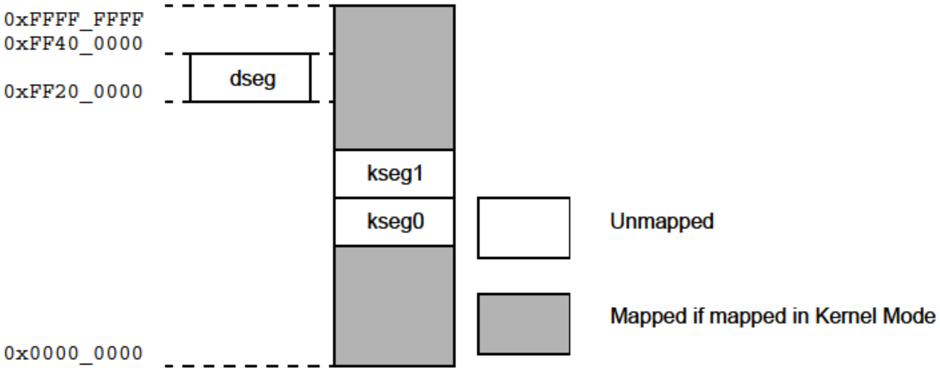
Figure 3.43 Kernel Mode Virtual Address Space — EVA Mode, 3GB Example



3.17.5 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 3.44.

Figure 3.44 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 3.21.

Accesses to memory that would normally cause an exception in kernel mode cause the CPU 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 3.21 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. drseg maps to the breakpoint registers 0x0_0000 - 0xF_FFFF	Uncached
	drseg	0xFF30_0000 through 0xFF3F_FFFF		

3.17.5.1 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 3.22

Table 3.22 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 14, “EJTAG Debug Support”.

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

3.17.5.2 Debug Mode, Memory (dmseg)

The conditions for CPU accesses to the dmseg address range (0xFF20_0000 to 0xFF2F_FFFF) are shown in Table 3.23.

Table 3.23 CPU Access to dmseg

Transaction	ProbEn Bit in DCR Register[1]	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

[1] The NoDCR bit in the CP0 Debug register indicates if the dmseg and drseg address spaces and associated DCR register exists in memory mapped space. The NoDCR bit must be cleared, this DCR register exists. If the bit is set, the register does not exist.

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.

3.18 TLB Instructions

Table 3.24 lists the TLB-related instructions implemented in the proAptiv core. .

Table 3.24 TLB Instructions

Mnemonic	Instruction	Description
TLBP	Translation Lookaside Buffer Probe	Used to determine whether a particular address was successfully translated. When a TLBP instruction is executed and fails to find a match for the specified virtual address, hardware sets bit 31 of the *Index* register.
TLBR	Translation Lookaside Buffer Read	
TLBWI	Translation Lookaside Buffer Write Index	TLB write extended to support invalidation of individual TLB entries.
TLBWR	Translation Lookaside Buffer Write Random	
TLBINV	Translation Lookaside Buffer Invalidate	Added to support set level invalidation of TLB entries.
TLBINVF	Translation Lookaside Buffer Invalidate Flush	Added to support VTLB flush based invalidation of TLB entries.

Refer to the Instructions chapter for more information on the TLB instructions.

Chapter 4

Primary and Secondary Caches

This chapter describes the caches present in an proAptiv core and contains the following sections:

- Section 4.1 “Cache Configurations”
- Section 4.2 “L1 Instruction Cache”
- Section 4.3 “L1 Data Cache”
- Section 4.4 “L1 Instruction and Data Cache Software Testing”
- Section 4.5 “L2 Cache”
- Section 4.6 “The CACHE Instruction”
- Section 4.7 “Write Back Buffer”

4.1 Cache Configurations

The proAptiv Multiprocessing System contains three caches; L1 instruction, L1 data, and shared L2. All of these caches are non-optional in the proAptiv architecture and are always present. The size of each cache can be configured as shown in Table 4.1.

Table 4.1 proAptiv Cache Configurations

Attribute	L1 Instruction Cache	L1 Data Cache	L2 Cache
Size[1]	32 KB or 64 KB	32 KB or 64 KB	256 KB, 512 KB 1 MB, 2 MB, 4 MB, or 8 MB
Line Size	32-byte	32-byte	32-byte or 64-byte
Number of Cache Sets	256 or 512	256 or 512	512, 1024, 2048, 4096, 8192, 16384, or 32768
Associativity	4 way	4 way	8 way

[1] For Linux-based applications, MIPS recommends an optimum cache size of 64 KB, and a minimum cache size of 32 KB.

The L1 instruction cache is attached to the Instruction Fetch Unit (IFU) via four 128-bit data paths, allowing for up to four instruction fetches per cycle.The L1 data cache contains four 128-bit data paths, allowing for up to four data read/write operations per cycle. The L2 cache is embedded within the Coherence Manager (CM2) and communicates with external memory via a configurable 64-bit or 256-bit OCP interface.

For more information on the L1 instruction cache, refer to Section 4.2 “L1 Instruction Cache”.

For more information on the L1 data cache, refer to Section 4.3 “L1 Data Cache”.

For more information on the L2 cache, refer to Section 4.5 “L2 Cache”.

4.1.1 Cacheability Attributes

The proAptiv core supports the following cacheability attributes:

- *Uncached (code #2)*: 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.
- *Non-coherent Writeback With Write Allocation (code #3)*: 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 'dirty' 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.
- *Coherent Write-back With Write Allocation, Exclusive (code #4)*: This attribute is similar to code #5 described below, except that load misses bring data into the cache in the exclusive state rather than the shared state. This can be used if data is not shared and will eventually be written. This can reduce bus traffic, because the line does not have to be refetched in an exclusive state when a store is done.
- *Coherent Write-back With Write Allocation, Exclusive on Write (code #5)*: Use coherent data. Load misses will bring the data into the cache in a shared state. Multiple caches can contain data in the shared state. Stores will bring data into the cache in an exclusive state - no other caches can contain that same line. If a store hits on a shared line in the cache, the line will be invalidated and brought back into the cache in an exclusive state.
- *Uncached Accelerated (code #7):* Uncached stores are gathered together for more efficient bus utilization.

4.2 L1 Instruction Cache

The L1 instruction cache contains three arrays: tag, data, and way-select. The L1 instruction cache is 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.

A tag entry consists of the upper bits of the physical address (bits [31:12]) for instruction 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.

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 instruction cache only supports reads, hence only LRU entries are stored in the instruction way-select array.

Table 4.2 shows the key characteristics of the L1 instruction cache. Figure 4.1 shows the format of an entry in the three arrays comprising the instruction cache: data, tag, and way-select.

Table 4.2 L1 Instruction Cache Attributes

Attribute	With Parity
Size[1]	32 KB or 64 KB
Line Size	32-byte
Number of Cache Sets	256 or 512
Associativity	4-way
Replacement	LRU
Cache Locking	per line
Data Array	
Read Unit	144b x 4
Write Unit	144b
Tag Array	
Read Unit	55b x 4
Write Unit	55b
Way-Select Array	
Read Unit	6b
Write Unit	1-6b

[1] For Linux based applications, MIPS recommends a 64 KB L1 instruction cache size, with a minimum size of 32 KB.

Figure 4.1 L1 Instruction Cache Organization

	5	1	1	20	7	7	7	7
Tag (per way): **(55 bits total)**	Parity	Valid	Lock	PA[31:12]	Precode_67	Precode_45	Precode_23	Precode_01

	16	64	64	16	64	64
Data (per way)[1]: **(288 bits total)**	Parity	dword3	dword2	Parity	dword1	dword0

	6
Way-Select: **(6 bits total)**	LRU

[1] Parity bits in data array will be interleaved with precode and data bytes.

4.2.1 L1 Instruction Cache Virtual Aliasing

The instruction cache on the proAptiv 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 16 KB.

In the proAptiv core, the **Config7**$_{IAR}$ bit is always set to indicate the existence of instruction cache virtual aliasing hardware. The core allows a physical address to reside at multiple indices if accessed with different virtual addresses. When an invalidate request is made due to the CACHE or SYNCI instructions, the core will serially check each possible alias location for the given physical address.

The hardware can be enabled and disabled using the **Config7**$_{IVAD}$ bit. When this bit is cleared, the hardware used to remove instruction cache virtual aliasing is enabled. In this case the virtual aliasing is managed in hardware. No software interaction is required. When the **Config7**$_{IVAD}$ bit is set, the virtual aliasing hardware is disabled. This can be done when software ensures that no cache aliases are possible, for example when using a minimum TLB page size of 16KB. In cases where the TLB page size is less than 116 KB, it is up to software to manage virtual aliasing within the instruction cache.

4.2.2 L1 Instruction Cache 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.

4.2.3 L1 Instruction Cache Parity

The instruction cache contains 16 parity bits — one for each byte of the 128 bits of 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. Instruction cache parity is always present in the instruction cache and cannot be disabled.

4.2.4 L1 Instruction Cache Replacement Policy

The L1 instruction cache 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 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:** 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.

4.2.5 L1 Instruction Cache Line Locking

The proAptiv 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.

Locking lines in the caches is somewhat counter to the idea of coherence. If a line is locked into a particular cache, it is expected that any processes utilizing that data will be locked to that processor and coherence is not needed. Based on this usage model, locking coherent lines into the cache is not recommended. However, should this occur, the CPUadheres to the following rules:

- SYNCI instructions are user-mode instructions. Since locking is a kernel mode feature (requires the CACHE instruction), SYNCI is not allowed to unlock cache lines. This applies to both local and globalized SYNCI instructions.
- Locking overrides coherence. Intervention requests from other CPUs and I/O devices that match on a locked line are treated as misses.
- Self-intervention requests for globalized CACHE instructions are allowed to affect a locked line. This is done primarily for handling lock and unlock requests for kseg0 addresses when kseg0 is being treated coherently.
- The CPU 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, subsequent cache misses at that cache index will displace one of the locked lines.

4.2.6 L1 Instruction Cache Memory Coherence Issues

The proAptiv core supports cache coherency in a multi-CPU cluster using Cache Coherence Attributes (CCAs) specified on a per cache-line basis and an Intervention Port containing coherent requests by all CPUs in the system. Each proAptiv core monitors its Intervention Port and updates the state of its cache lines (valid, lock, and dirty tag bits) accordingly.

The L1 instruction caches utilizes a modified MESI protocol. Each cache line will be in one of the following states:

Invalid: The line is not present in this cache.

Exclusive: This cache has a copy of the line with the right to modify. The line is not present in other L1 data caches. The line is still clean and is consistent with the value in L2 cache or memory.

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

In the proAptiv core, the hardware does not automatically keep the instruction caches coherent with the data caches. Doing so requires many additional cache lookups and would likely require the instruction cache tag array to be duplicated as well. For many types of code, this would be of small benefit, and the added area and power costs would not make sense. Further, the existing non-coherent cores from MIPS do not keep the I-Cache coherent with the D-Cache, so the code already exists for software I-Cache coherence where it is required.

Globalized CACHE and SYNCI instructions ease the task of software I-Cache coherence. Existing, single-CPU routines that push dirty data out of the data cache and invalidate stale instruction cache lines using hit-type CACHE or SYNCI instructions can be globalized, and the coherence can be handled for all of the instruction caches in parallel.

4.2.7 Software I-Cache Coherence (JVM, Self-modifying Code)

The CPU does not support hardware I-Cache coherence, so code that modifies the instruction stream must clean up the instruction cache. This is equivalent to what is currently required on uniprocessor systems that also do not have a coherent I-Cache. The recommended SYNCI sequence shown below will also work for coherent addresses:

```
SW instn_address
SYNCI instn_address
SYNC
JR.HB instn_address
```

```
NOP
```

4.2.8 L1 Instruction Software Cache Management

The L1 instruction cache is not fully “coherent” and requires OS intervention at times. The CACHE instruction is the building block of such OS interventions, and is required for correct handling of DMA data and for cache initialization. Historically, the CACHE instruction also had a role when writing instructions. Unless the programmer takes the appropriate action, those instructions may only be in the D-cache and would need them to be fetched through the I-cache at the appropriate time. Wherever possible, use the SYNCI instruction for this purpose, as described in Section 4.2.11 “Cache Management When Writing Instructions - the “SYNCI” Instruction”.

A cache operation instruction is written **`cache op,addr`** where **`addr`** is just an address format, written as for a load/store instruction. Cache operations are privileged and can only run in kernel mode (SYNCI works in user mode, though).

31 26	25 21	20 18	17 16	15 0
cache	base	op		offset
47	register	what to do	which cache	

Figure 4.2 Fields in the Encoding of a CACHE Instruction

The **`op`** field packs together a 5-bit field. The lower 2 bits of this field (17:16) select which cache to work on:

00	L1 I-cache
01	L1 D-cache
10	reserved
11	L2 cache

The upper 3-bits of the OP field encodes a command to be carried out on the line the instruction selects.

The CACHE instruction come in three varieties which differ in how they pick the cache entry (the “cache line”) they will work on:

- *Hit-type cache operation*: presents an address (just like a load/store), which is looked up in the cache. If this location is in the cache (it “hits”) the cache operation is carried out on the enclosing line. If this location is not in the cache, nothing happens.

- *Address-type cache operation*: presents an address of some memory data, which is processed just like a cached access - if the cache was previously invalid the data is fetched from memory.

- *Index-type cache operation*: as many low bits of the address as are required are used to select the byte within the cache line, then the cache line address inside one of the four cache ways, and then the way. The size of the cache (contained within the *Config1* register) to know exactly where the field boundaries are located. The address is used as follows:

31		5	4 0
Unused	Way1-0	Index	byte-within-line

Note that the MIPS32 specification allows the CPU designer to select whether to derive the index from the virtual or physical address. For index-type operations, MIPS recommends using a kseg0 address, so that the virtual and physical address are the same. This also avoids a potential of cache aliasing.

4.2.9 L1 Instruction Cache CP0 Register Interface

The proAptiv core uses different CP0 registers for instruction cache operations.

Table 4.1 Instruction Cache CP0 Register Interface

CP0 Registers	CP0 number
Config1	*16.1*
CacheErr	*27.0*
ITagLo	*28.0*
ITagHi	*29.0*
IDataLo	*28.1*
IDataHi	*29.1*
L23TagLo[1] *L23DataLo* *L23DataHi*	*28.4* *28.5* *29.5*

[1] In past versions of this manual *L23TagLo* was known as “STagLo”, and so on. But this name is more mnemonic.

4.2.9.1 Config1 Register (CP0 register 16, Select 1)

The *Config1*.$_{IS}$ field (bits 24:22) indicates the number of sets per way in the instruction cache. The proAptiv L1 instruction cache supports 256 sets per way, which is used to configure a 32 KB cache, or 512 sets per way, which is used to configure a 64 KB cache.

The *Config1*.$_{IL}$ field (bits 21:19) indicates the line size for the instruction cache. The proAptiv L1 instruction cache supports a fixed line size of 32 bytes as indicated by a default value of 4 for this field.

The *Config1*.$_{IA}$ field (bits 18:16) indicates the set associativity for the instruction cache. The proAptiv L1 instruction cache is fixed at 4-way set associative as indicated by a default value of 3 for this field.

For more information, refer to Section 2.3.1.2, "Device Configuration 1 — Config1 (CP0 Register 16, Select 1)".

4.2.9.2 CacheErr Register (CP0 register 27, Select 0)

The *CacheErr* register is a read-only register used to determine the status of a cache error. The upper two bits of this register (*CacheErr*.$_{EREC}$) indicate whether the contents of the register pertain to an L1 instruction cache error, an L1 data cache error, a TLB error, or an external error. This register provides information such as:

- L1 data versus L2 data cache error
- Tag RAM versus Data RAM error
- External snoop request indication in multi-core systems
- Indicates coherent L1 cache error in another CPU in a multi-core system
- Fatal/non-fatal error indication

- Indicates if the error affects the Scratchpad RAM
- Indicates the cache index or Scratchpad RAM index of the double word entry where the error occurred

For more information, refer to Section 2.3.6.11, "Cache Error — CacheErr (CP0 Register 27, Select 0)".

4.2.9.3 L1 Instruction Cache TagLo Register (CP0 register 28, Select 0)

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 of the $ErrCtl_{WST}$ and $ErrCtl_{SPR}$ *bits.*

- 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)

For more information, refer to Section 2.3.6.1, "Level 1 Instruction Cache Tag Low — ITagLo (CP0 Register 28, Select 0)".

4.2.9.4 L1 Instruction Cache TagHi Register (CP0 register 29, Select 0)

This register represents the I-cache pre-decode bits and is intended for diagnostic use only.

For more information, refer to Section 2.3.6.2, "Level 1 Instruction Cache Tag High — ITagHi (CP0 Register 29, Select 0)".

4.2.9.5 L1 Instruction Cache DataLo Register (CP0 register 28, Select 1)

Staging registers for special **`cache`** instruction which loads or stores data from or to the cache line. Two registers (*IDataHi*, *IDataLo*) are needed, because the proAptiv core loads I-cache data at least 64 bits at a time. This register stores the lower 32 bits of the load data.

For more information, refer to Section 2.3.6.3, "Level 1 Instruction Cache Data Low — IDataLo (CP0 Register 28, Select 1)".

4.2.9.6 L1 Instruction Cache DataHi Register (CP0 register 29, Select 1)

Staging registers for special **`cache`** instruction which loads or stores data from or to the cache line. Two registers (*IDataHi*, *IDataLo*) are needed, because the proAptiv core loads I-cache data at least 64 bits at a time. This register stores the upper 32 bits of the load data.

For more information, refer to Section 2.3.6.4, "Level 1 Instruction Cache Data High — IDataHi (CP0 Register 29, Select 1)".

4.2.10 L1 Instruction Cache Initialization

The L1 instruction cache must be initialized during power-up or reset in order to place the lines of the cache in a known state. This is accomplished via the cache initialization routine, which is normally part of the boot code. For experienced user’s, a sample boot code is shown in the following subsection.

4.2.10.1 L1 Instruction Cache Initialization Routine

The following assembly provides an example initialization routine for the instruction cache.

```
/**************************************************************************************
init_icache invalidates all Instruction cache entries
**************************************************************************************/

LEAF(init_icache)

      // For this Core there is always an instruction cache
      // The IS field determines how many sets there are:
      // IS = 2 there are 256 sets
      // IS = 3 there are 512 sets
      // $11 set to line size, will be used to increment through the cache tags

      li    $11, 32           # Line size is always 32 bytes.
      mfc0  $10, $16, 1       # Read C0_Config1
      ext    $12, $10, 22, 3  # Extract IS
      li    $14, 2            # Used to test against
      beq   $14, $12, Isets_done# if  IS = 2
      li    $12, 256          # sets = 256
      li    $12, 512          # else sets = 512 Skipped if branch taken
Isets_done:
      lui   $14, 0x8000       # Get a KSeg0 address for cacheops
      // clear the lock bit, valid bit, and the LRF bit
      mtc0    $0, $28         # Clear C0_ITagLo to invalidate entry

next_icache_tag:
      cache   0x8, 0($14)     # Index Store tag Cache opt
      add       $12, -1       # Decrement set counter
      bne     $12, $0, next_icache_tag # Done yet?
      add     $14, $11        # Increment line address by line size
done_icache:

      ins     r31_return_addr, $0, 29, 1
      jr      r31_return_addr
      nop
END(init_icache)
```

4.2.10.2 L1 Instruction Cache Initialization Routine Details

This section provides a detailed description of each line of code in the L1 instruction cache initialization routine described above. Note that this code represents an example of an implementation specific cache initialization. The core is code is used in specifies specific cache sizes of 32K or 64K, is always part of a CPS and will always have the L2 cache present. The code example is written with those parameters in mind.

Before use, the cache must be initialized to a known state; that is, all cache entries must be invalidated. This code example initializes the cache, finds the total number of cache sets, then loops through the cache sets using the cache instruction to invalidate each cache set.

```
LEAF (init_icache)

// For this Core there is always an L1 nstuction cache
// The IS field determines how many sets there are
// IS = 2 there are 256 sets
// IS = 3 there are 512 sets
// $11 set to line size, will be used to increment through the cache tags

      li    $11, 32           # Line size is always 32 bytes.
```

This instruction cache always has a line size of 32 bytes, 4 ways and can have a size of either 32 KB or 64 KB. The IS field (sets per way) of the *Config1* register will be use to determine the size of the cache. This field can have one of two values. A value of 0x2 indicates a 32 KB cache and a value of 0x3 indicates a 64 KB cache.

```
      mfc0  $10, $16, 1       # Read C0_Config1
      ext   $12, $10, 22, 3   # Extract IS
      li    $14, 2            # Used to test against
```

If the check is true, the code uses the branch delay slot (which is always executed) to set the set iteration value to 256 for a 32 KB cache and then branches ahead to **Isets_done**. If the check is false, the code assumes that the size of the cache is 64 KB. At this point, the code still sets the iteration value to 256 in the branch delay slot, but then falls through and sets it again to 512 for a 64 KB cache.

```
      beq   $14, $12, Isets_done # if  IS = 2
      li    $12, 256          # sets = 256
      li    $12, 512          # else sets = 512 Skipped if branch taken
```

```
Isets_done:
```

GPR 14 will be used as an index into the cache. It will be set to a virtual address, and then translated to a physical address. Since the address 0x8000_0000 is in kseg0, the CPU will ignore the top bit, so virtual 0x8000_0000 will become physical address 0x0000_0000. Since the cache is physically indexed, the first time through the loop, the cache instruction will write the tag to way 0 index line 0.

The **lui** instruction loads 0x8000 into the upper 16 bits and clears the lower 16 bits of the GPR14 register.

```
      lui   $14, 0x8000       # Get a KSeg0 address for cacheops
```

Clearing the tag registers performs two important functions: it sets the Physical Tag address called PTagLo to 0, which ensures the upper physical address bits are zeroed out, and it also clears the valid bit for the set, which ensures that the set is free and may be filled as needed.

The code uses the Move to Coprocessor zero instruction to move the general purpose register zero, which always contains a zero, to the tag register.

```
// clear the lock bit, valid bit, and the LRF bit

      mtc0  $0, $28     # Clear C0_ITagLo to invalidate entry
```

The **`Cache`** instruction uses the **`Index Store Tag`** operation on the Level 1 instruction cache so the op field is coded with a value of 0x8. The first two bits are 2’b00 for the L1 instruction cache, and the operation code for **`Index Store tag`** is encoded as 3’b010 in bits two, three and four.

`next_icache_tag:`

```
      cache 0x8, 0($14) # Index Store tag Cache op
```

The index type of operation can be used to address a byte in the cache in a specific way of the cache. This is done by breaking down the virtual address argument stored in the base register of the **`Cache`** instruction into several fields.

Bits 14:0 of the Cache Instruction

14 13	12 5	4 0
Way	Page Index	Byte Index

The size of the index field varies according to the size of a cache way. The larger the way, the larger the index. In the table above, the combined byte and page index is 13 bits because each way of the cache is 8K. The way number is always the next two bits following the index.

The code does not explicitly set the way bits. Instead it just increments the virtual address by the cache lines size so the next time through the loop the **`Cache`** instruction will initialize the next set in the cache. Eventually this increment has the effect of setting the cache to index 0 of the next way in the cache because it overflows into the way bits.

At this point all the code needs to do is loop maintenance. First decrement the loop counter (12/t4).

```
      add   $12, -1                 # Decrement set counter
```

Then test it to see if it has gotten to zero and if it has not branch back to label one.

```
      bne   $12, $0, next_icache_tag # Done yet?
```

The instruction in the branch delay slot, which is always executed, is used to increment the virtual address (14/t6) to the next set in the cache. (11/t3) holds the line size in bytes.

```
      add   $14, $11          # Increment line address by line size
```

From this point on, the code can be executed from a cached address. This is easily done by changing the return address from a KSEG1 address to a KSEG0 address by simply inserting a 0 into bit 29 of the address. However, during debugging, this operation will confuse the debugger and you will no longer be able to do source-level debugging. That is why it is commented out here. Once the code has been debugged, the "ins" line can be uncommented.

`done_icache:`

```
      // Modify return address to kseg0 which is cacheable
      // (for code linked in kseg1.)
      // However it makes it easier to debug if this is not done. So while
      // debugging, this should be commented out.
```

```
        ins      r31_return_addr, $0, 29, 1
        jr       r31_return_addr
        nop

END (init_icache)
```

4.2.11 Cache Management When Writing Instructions - the “SYNCI” Instruction

The **`synci`** instruction (new to the MIPS32 Release 2 update) provides a mechanism available to user-level code for ensuring that previously written instructions are correctly presented for execution (it combines a D-cache writeback with an I-cache invalidate). Use of the **`synci`** instruction is preferred to the traditional alternative of a D-cache writeback followed by an I-cache invalidate.

4.3 L1 Data Cache

The L1 data cache is similar to the instruction cache, with a few key differences;

- In addition to the three arrays (tag, data, and way-select), the L1 data cache also contains a separate dirty array to hold the dirty bits of cache lines.
- The data cache does not contain any precode information.
- To handle store bytes, the data array is byte-accessible, and the data parity is 1 bit per byte.
- The way-select array for the data cache holds the lock bits (and 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.

Like the L1 instruction cache, the L1 data cache is virtually indexed, since a virtual address is used to select the appropriate line within each of the arrays. The cache is 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.

A tag entry consists of the upper bits of the physical address (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, word, or doubleword 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.

Table 4.3 shows the key characteristics of the data cache. Figure 4.3 shows the format of an entry in the arrays comprising the data cache: tag, data, way-select, and dirty.

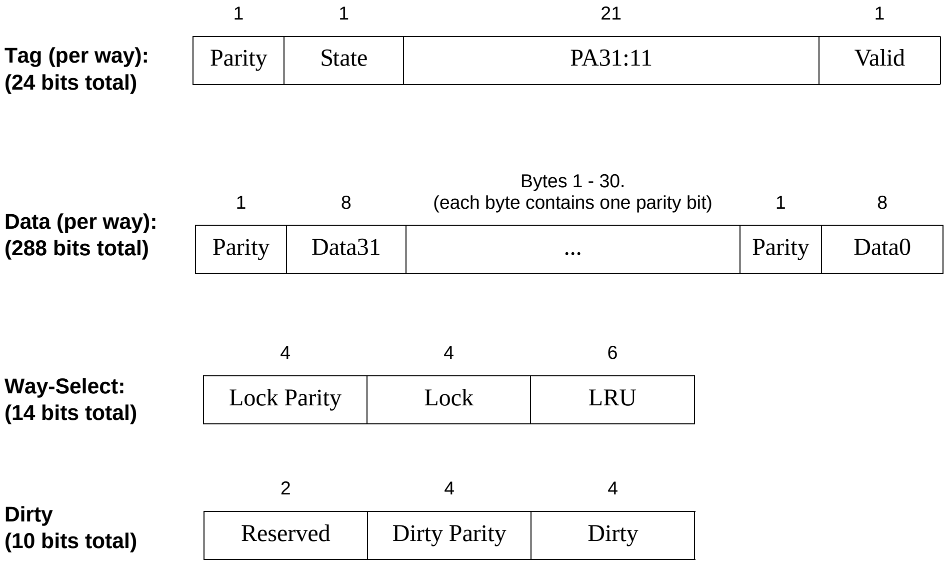
Table 4.3 L1 Data Cache Organization

Attribute	With Parity
Size	32 or 64KB
Line Size	32-byte
Number of Cache Sets	256 or 512
Associativity	4-way
Replacement	LRU
Cache Locking	per line
Data Array	
Read Unit	72b x 4
Write Unit	9b
Tag Array	
Read Unit	24b x 4
Write Unit	24b

Table 4.3 L1 Data Cache Organization*(continued)*

Attribute	With Parity
Way-Select Array	
Read Unit	14b
Write Unit	1-14b
Dirty Array	
Read Unit	10b
Write Unit	1-10b

Figure 4.3 L1 Data Cache Organization



4.3.1 L1 Data Cache Virtual Aliasing

The data cache on the proAptiv 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.

The following table indicates the conditions under which virtual aliasing can occur.

Table 4.1 L1 Data Cache Virtual Aliasing Conditions

Cache Size	MMU Page Size	Way Size	Aliasing Can Occur	Hardware Aliasing Fix Required
32 KB	4 KB	8 K	Yes	Yes
64 KB	4 KB	16 K	Yes	Yes
32 KB	>= 16 KB	8 K	No	No

Table 4.1 L1 Data Cache Virtual Aliasing Conditions*(continued)*

Cache Size	MMU Page Size	Way Size	Aliasing Can Occur	Hardware Aliasing Fix Required
64 KB	>= 16 KB	16 K	No	No

In the proAptiv core, the read-only **`Config7`**.*AR* bit determines whether the data cache virtual aliasing hardware is enabled based on the build-time configuration. Note that for some of the configuration options in the table above, the hardware aliasing fix (HWAF) is required. As such, it is incumbent upon the designer to select the HWAF option at build time. The selection of this option causes hardware to set the **`Config7`**.*AR* bit.

4.3.2 L1 Data Cache Parity

The data array requires a parity bit for each byte, corresponding to the minimum number of bytes 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. Instruction cache parity is always present in the instruction cache and cannot be disabled.

4.3.3 L1 Data Cache 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.
  - **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.

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.

4.3.4 L1 Data Cache Line Locking

The mechanism for line locking in the L1 data cache is identical to that of the L1 instruction cache. For more information, refer to Section 4.2.5, "L1 Instruction Cache Line Locking".

4.3.5 L1 Data Cache Memory Coherence Protocol

The proAptiv core supports cache coherency in a multi-CPU cluster using Cache Coherence Attributes (CCAs) specified on a per cache-line basis and an Intervention Port containing coherent requests by all CPUs in the system. Each proAptiv core monitors its Intervention Port and updates the state of its cache lines (valid, lock, and dirty tag bits) accordingly.

The L1 data caches utilize a standard MESI protocol. Each cache line will be in one of the following four states:

Invalid: The line is not present in this cache.

Shared: This cache has a read-only copy of the line. The line may be present in other L1 data caches, also in a Shared state. The line will have the same value as it does in the L2 cache or memory.

Exclusive: This cache has a copy of the line with the right to modify. The line is not present in other L1 data caches. The line is still clean - consistent with the value in L2 cache or memory.

Modified: This cache has a dirty copy of the line. The line is not present in other L1 data caches. This is the only up-to-date copy of the data in the system (the value in the L2 cache or memory is stale).

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

Some of the basic characteristics of the coherence protocol are summarized below. Coherence can occur on the data cache.

- Writeback cache - Uses a writeback cache to ensure high performance
- Cache-line based - Coherence and ownership is maintained per 32-byte cache line
- Snoopy protocol - Each CPU snoops the stream of transactions and updates its cache state accordingly
- Invalidate - A line is invalidated from the cache (possibly with a writeback to memory) when a store from another processor is seen.

4.3.6 L1 Data Cache Initialization

The L1 data cache must be initialized during power-up or reset in order to place the lines of the cache in a known state. This is accomplished via the cache initialization routine, which is normally part of the boot code. For experienced user's, a sample boot code is shown in the following subsection.

4.3.6.1 L1 Data Cache Initialization Routine

The following assembly provides an example initialization routine for the data cache.

```
/***************************************************************************
init_dcache invalidates all data cache entries
****************************************************************************/

LEAF (init_dcache)

      // For the proAptiv MPS there is always an L1 data cache
      // The ID field determines how many sets there are
      // DS = 2 there are 256 sets
      // DS = 3 there are 512 sets
      // $11 set to line size, will be used to increment through the cache tags

      li    $11, 32           # Line size is always 32 bytes
      mfc0  $10, $16, 1       # Read C0_Config1
      ext    $12, $10, 13, 3  # Extract DS
      li    $14, 2            # Used to test against
      beq   $14, $12, Dsets_done # if  DS = 2
      li    $12, 256          # sets = 256
      li    $12, 512          # else sets = 512, skipped if branch taken

Dsets_done:

      lui     $14, 0x8000     # Get a KSeg0 address for cacheops
      // clear the lock bit, valid bit, and the LRF bit
      mtc0    $0, $28, 2      # Clear C0_DTagLo to invalidate entry

next_dcache_tag:

      cache 0x9, 0($14)       # Index Store tag Cache opt
      add    $12, -1          # Decrement set counter
      bne   $12, $0, next_dcache_tag # Done yet?
      add    $14, $11         # Increment line address by line size

done_dcache:

       jr      r31_return_addr
      nop

END (init_dcache)
```

4.3.6.2 L1 Data Cache Initialization Routine Details

This section provides a detailed description of each line of code in the initialization routine. The L1 data cache initialization routine is very similar to the L1 instruction cache initialization routine.

```
LEAF(init_dcache)

      // For the proAptiv CPS there is always a L1 data cache
      // The DS field determines how many sets there are
      // DS = 2 there are 256 sets
      // DS = 3 there are 512 sets
      // $11 set to line size, will be used to increment through the cache tags

      li    $11, 32           # Line size is always 32 bytes.
```

The data cache always has a line size of 32 bytes and 4 ways, and can have a size of either 32 KB or 64 KB. The DS field (sets per way) of the *Config1* register is used to determine the size of the cache. This field can have one of two values. A value of 0x2 indicates a 32 KB cache and a value of 0x3 indicates a 64 KB cache.

```
      mfc0  $10, $16, 1       # Read C0_Config1
      ext   $12, $10, 13, 3   # Extract DS
      li    $14, 2            # Used to test against
```

If the check is true, the code uses the branch delay slot (which is always executed) to set the set iteration value to 256 for a 32 KB cache and then branches ahead to **Dsets_done**. If the check is false, the code assumes that the size of the cache is 64 KB. At this point, the code still sets the iteration value to 256 in the branch delay slot, but then falls through and sets it again to 512 for a 64 KB cache.

```
      beq   $14, $12, Dsets_done # if  DS = 2
      li    $12, 256          # sets = 256
      li    $12, 512          # else sets = 512 Skipped if branch taken
```

```
Dsets_done:
```

GPR 14 will be used as an index into the data cache. It is set to a virtual address and then translated to a physical address. Since the address 0x8000_0000 is in kseg0, the CPU will ignore the top bit, so virtual 0x8000_0000 will become physical address 0x0000_0000. Since the cache is physically indexed, the first time through the loop, the cache instruction will write the tag to way 0 index line 0.

The **lui** instruction loads 0x8000 into the upper 16 bits and clears the lower 16 bits of the GPR14 register.

```
      lui   $14, 0x8000       # Get a KSeg0 address for cacheops
```

Clearing the tag registers performs two important functions: it sets the Physical Tag address called PTagLo to 0, which ensures the upper physical address bits are zeroed out, and it also clears the valid bit for the set, which ensures that the set is free and may be filled as needed.

The code uses the Move to Coprocessor zero instruction to move the general purpose register zero, which always contains a zero, to the tag register.

```
      // clear the lock bit, valid bit, and the LRF bit
      mtc0    $0, $28, 2      # Clear C0_DTagLo to invalidate entry
```

The **`Cache`** instruction uses the **`Index Store Tag`** operation on the Level 1 data cache so the op field is coded with a value of 0x9. The first two bits are 2’b01 for the L1 data cache, and the operation code for **`Index Store tag`** is encoded as 3’b010 in bits two, three and four.

```
next_dcache_tag:

      cache 0x9, 0($14) # Index Store tag Cache opt
```

The index type of operation can be used to address a byte in the cache in a specific way of the cache. This is done by breaking down the virtual address argument stored in the base register of the **`Cache`** instruction into several fields.

Bits 14:0 of the Cache Instruction

14 13	12 5	4 0
Way	Page Index	Byte Index

The size of the index field varies according to the size of a cache way. The larger the way, the larger the index. In the table above, the combined byte and page index is 13 bits because each way of the cache is 8K. The way number is always the next two bits following the index.

The code does not explicitly set the way bits. Instead it just increments the virtual address by the cache line size so the next time through the loop the **`Cache`** instruction will initialize the next set in the cache. Eventually this increment has the effect of setting the cache to index 0 of the next way in the cache because it overflows into the way bits.

At this point all the code needs to do is loop maintenance. First decrement the loop counter (12/t4).

```
      add   $12, -1           # Decrement set counter
```

Then test it to see if it has gotten to zero and if not branch back to label one.

```
      bne   $12, $0, next_dcache_tag # Done yet?
```

The instruction in the branch delay slot, which is always executed, is used to increment the virtual address (14/t6) to the next set in the cache. (11/t3) holds the line size in bytes

```
      add   $14, $11          # Increment line address by line size
```

At this point the Dcache initialization is done.

```
done_dcache:

      jr      r31_return_addr
      nop

END (init_dcache)
```

4.3.7 Data Cache CP0 Register Interface

The proAptiv core uses different CP0 registers for data cache operations.

Table 4.4 Data Cache CP0 Register Interface

CP0 Registers	CP0 number
Config1	*16.1*
CacheErr	*27.0*
DTagLo	*28.2*
DDataLo	*28.3*
L23TagLo[1]	*28.4*
L23DataLo	*28.5*
L23DataHi	*29.5*

[1] In past versions of this manual *L23TagLo* was known as “STagLo”, and so on. But this name is more mnemonic.

4.3.7.1 Config1 Register (CP0 register 16, Select 1)

The *Config1.*$_{DS}$ field (bits 15:13) indicates the number of sets per way in the data cache. The proAptiv L1 data cache supports 256 sets per way, which is used to configure a 32 KB cache, or 512 sets per way, which is used to configure a 64 KB cache.

The *Config1.*$_{DL}$ field (bits 12:10) indicates the line size for the data cache. The proAptiv L1 data cache supports a fixed line size of 32 bytes as indicated by a default value of 4 for this field.

The *Config1.*$_{DA}$ field (bits 9:7) indicates the set associativity for the data cache. The proAptiv L1 data cache is fixed at 4-way set associative as indicated by a default value of 3 for this field.

For more information, refer to Section 2.3.1.2, "Device Configuration 1 — Config1 (CP0 Register 16, Select 1)".

4.3.7.2 CacheErr Register (CP0 register 27, Select 0)

The *CacheErr* register is a read-only register used to determine the status of a cache error. The upper two bits of this register (*CacheErr.*$_{EREC}$) indicate whether the contents of the register pertain to an L1 instruction cache error, an L1 data cache error, a TLB error, or an external error.

For more information, refer to Section 2.3.6.11, "Cache Error — CacheErr (CP0 Register 27, Select 0)".

4.3.7.3 L1 Data Cache TagLo Register (CP0 register 28, Select 2)

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

In a multi-core system, the D-cache has five logical memory arrays associated with this *DTagLo* register. The tag RAM stores tags and other state bits with special attention to the needs of the CPU. The duplicate tag RAM also stores tags and state, but is optimized for the needs of interventions. Both of these arrays are set-associative (4-way). The Dirty RAM and duplicate Dirty RAM store the dirty bits (indicating modified data) for CPU and intervention uses, and each combine their ways together in a single entry per set. The WS RAM also combines the lock and LRU data in a single entry per set. Accessing these arrays for index cache loads and stores is controlled by using three bits in the *ErrCtl* register to create modes that allow the correct access to these arrays.

Note that the proAptiv core does not implement the *DTagHi* register.

The interpretation of this register changes depending on the settings of $ErrCtl_{WST,}$ $ErrCtl_{DYT,}$ and $ErrCtl_{SPR.}$

For more information, refer to Section 2.3.6.5, "Level 1 Data Cache Tag Low — DTagLo (CP0 Register 28, Select 2)".

4.3.7.4 L1 Data Cache DataLo Register (CP0 register 28, Select 3)

In the proAptiv core, 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.

Note that the proAptiv core does not implement the *DDataHi* register.

For more information, refer to Section 2.3.6.6, "Level 1 Data Cache Data Low — DDataLo (CP0 Register 28, Select 3)".

4.4 L1 Instruction and Data Cache Software 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.

4.4.1 L1 Instruction Cache Tag Arrays

The L1 instruction cache tag array can be tested via the **`Index Load Tag`** and **`Index Store Tag`** varieties of the **`CACHE`** instruction. An **`Index Store Tag`** writes the contents of the *ITagLo and ITagHi* registers into the selected tag entry. An **`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.

4.4.2 L1 Data Cache Tag Arrays

The L1 data cache tag array can be tested via the **`Index Load Tag`** and **`Index Store Tag`** varieties of the **`CACHE`** instruction. An **`Index Store Tag`** writes the contents of the *DTagLo* register into the selected tag entry. An **`Index Load Tag`** will read the selected tag entry into the *DTagLo* register.

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 *DTagLo* as the parity values.

4.4.3 Duplicate D-cache Tag Array

This array can be tested via the Index Load Tag and Index Store Tag varieties of the CACHE instruction. In order to access the duplicate tags, the WST and SPR bits of ErrCtl should both be set. Index Store Tag will write the contents of the TagLo register into the selected tag entry. Index Load Tag will read the selected tag entry into the TagLo. In normal mode, with WST and SPR cleared, *IndexStoreTags* will write into both the primary and duplicate tags, while *IndexLoadTags* will read the primary tag.

If parity is implemented, the parity bit can be tested as a normal bit by setting the PO bit in the ErrCtl register. This will override the parity calculation and write P bit in TagLo as the parity value.

4.4.4 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 executing 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.

4.4.5 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.

4.4.6 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.

4.4.7 D-Cache WS Array

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

4.4.8 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.

4.5 L2 Cache

The L2 cache processes transactions that are not serviced by the L1 cache. L2 is generally larger than the L1 cache, but slower, due to the use of higher-density memories.

The L2 communicates with external memory via an Open Core Protocol (OCP) interface. Because the L2 cache is integrated into the Coherence Manager (CM2) in the proAptiv architecture, no OCP interface between the two is required, reducing both latency and complexity.

The L2 also communicates with the CPU(s) through the performance counter interface, error reporting interface, and other side band signals. In addition to these interfaces, the L2 has the clock, reset, and bypass signals as well as some static input signals which can be used to configure it for different operating modes.

4.5.1 L2 Cache General Features

- 7-stage pipeline. (Optional 8th stage[1] for pipelined memory arrays.)
- 32-bit address paths and 256-bit internal data paths
- Associativity: 8-way
- Cache size: 256 KB, 512 KB, 1 MB, 2 MB, 4 MB, 8 MB
- Latency: The hit latency is 9-1-1-1 in the *cm_clk* domain (9-0-0-0 if a wide system bus). If pipelined memory arrays are used, the latency will increase to 10 cycles. If parity is enabled, the latency is 10 cycles. If both pipelined and parity, it is 11 cycles. The L2 read miss request latency is 8 cycles from the cycle L2 receives a processor request to the cycle L2 sends a request to the system. The L2 read miss response latency is 2 cycles from receiving the system response to L2 driving the response to the processor. Therefore, the aggregate read miss latency is 10 cycles plus the system latency. Of the 8 cycle read miss latency, 5 cycles are required to determine hit/miss and 3 cycles are required from the detected miss to the time the request goes out on the system bus. Uncached read latency is 7 (five request plus 2 response) cycles plus system latency.
- Line Size: 32 or 64 bytes (4 or 8 doublewords)
- Locking Support: Yes
- Replacement Algorithm: Pseudo LRU for 8-way
- Write policy: Write Back and Write through
- Write miss allocation policy: No-Write-Allocate and Write-Allocate.
- Error Checking and Correction (ECC): Optional 2-bit error detection and 1-bit error correction covering the tag and data arrays. 1-bit error detection covering the WS array.
- Maximum read misses outstanding: 8, 12 or 15. Build-time configuration option.
- Out-Of-Order processing (OOO): Yes
- Coherency: Non-coherent

[1] Build time option. The customer must choose this option if they are using pipelined RAM's in the wrappers instead of standard RAM cells (that are not pipelined in this way).

- 256-bit or 64-bit OCP SData/MData width on memory-side OCP interface.
- OCP Burst Size on the memory interface: 64-byte line size: 8 beats of 64-bit data or 2 beats of 256-bit data
- Bypass Mode Support: In bypass mode, all processor requests are routed to the system. This mode is used only for debug purposes and should not be used during normal operation.
- Multi-cycle Data Rams: 0, 1, 2, or 3 stalls can set Data RAM access times to 1, 2, 3, or 4 clocks.
- Multi-cycle Tag Rams: 0, 1, 2, or 3 stalls can set Tag RAM access times to 1, 2, 3, or 4 clocks.
- Multi-cycle Way-Select Rams: 0, 1, 2, or 3 stalls can set the Way-Select RAM access times to 1, 2, 3, or 4 clocks.
- Endianness: Independent of endianness

Table 4.5 L2 Cache Attributes

Attribute	With Parity
Size	256 KB, 512 KB, 1 MB, 2 MB, 4 MB, or 8 MB
Line Size	32-byte or 64-byte
Number of Cache Sets	512, 1024, 2048, 4096, 8192, 16384 of 32768
Associativity	8 way

In the table above, the associativity of the L2 cache is fixed at 8 ways. As a result, changes to the number of sets per way and the line size determine the overall size of the L2 cache. The proAptiv Multiprocessing Systemonly supports the cache sizes shown in Table 4.5 above. As a result, some of the options for line size and sets per way are invalid as they would result in cache sizes being either smaller or larger than those listed above. Table 4.6 shows the list of possible configurations and which ones are valid or invalid. The invalid configurations are shaded in the table.

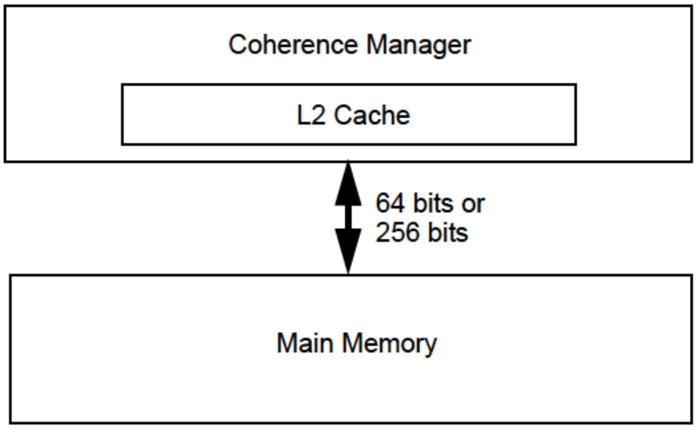
Table 4.6 Valid and Invalid Cache Configurations

Line Size	Sets per Way	Number of Ways	Total L2 Cache Size	Valid L2 Cache Configuration	Notes
32 bytes	1024	8	256 KBytes	Yes	
32 bytes	2048	8	512 KBytes	Yes	
32 bytes	4096	8	1 MByte	Yes	
32 bytes	8192	8	2 MByte	Yes	
32 bytes	16384	8	4 MByte	Yes	
32 bytes	32768	8	8 MByte	Yes	
64 bytes	512	8	256 KBytes	Yes	
64 bytes	1024	8	512 KBytes	Yes	
64 bytes	2048	8	1 MByte	Yes	
64 bytes	4096	8	2 MByte	Yes	
64 bytes	8192	8	4 MByte	Yes	
64 bytes	16384	8	8 MByte	Yes	
64 bytes	32768	8	16 MByte	No	32768 sets/way valid only with 32 byte line size

4.5.2 OCP Interface

In the proAptiv Multiprocessing System, the L2 cache is integrated into the CM2. This integration improves performance by eliminating the OCP interface that originally connected the L2 cache to the CM, or the L2 cache to the CPU depending on configuration. The OCP interface between the CM2 and the memory is programmable for widths of either 64-bit or 256-bit and has a fixed 64-byte line size. This is shown in Figure 4.4.

Figure 4.4 .OCP Interface Between CM2 and Memory



4.5.3 L2 Replacement Policy

The proAptiv architecture uses a pseudo-LRU replacement algorithm. The system memory configuration does not affect the replacement policy.

4.5.4 L2 Allocation Policy

The L2 cache controller can change its allocation policy based on the state of the *PB_MReqInfo[4]* pin. This feature is only supported with 32-byte cache line size. With 64-byte cache line, *PB_MReqInfo[4]* is ignored and the L2 cache always defaults to the behavior as if *PB_MReqInfo[4]* is zero - no allocate on write miss and allocate on read miss.

4.5.4.1 Write

The L2 will process the cached full-line writes in different ways depending on the allocation policy attribute on *PB_MReqInfo*. Table 4.7 shows the operations of the L2 according to the different allocation types.

When a write hits in the L2 cache, the data in the cache is always updated with the new data from the OCP master.

Table 4.7 L2 writes - Full Line AND Write-Back Cacheable (cca 3)

Allocation policy (*PB_MReqInfo[4]*)	L2 hit/miss	What L2 does[1]
No allocate (0)	hit	overwrite, mark dirty
	miss	write out to the main memory, no-allocate
Allocate (1)	hit	overwrite, mark dirty
	miss	write-allocate, mark dirty

[1] If not mentioned, L2 does not generate a main memory write.

When a write misses in the L2 cache, the data is written into either the L2 cache or the main memory depending on *PB_MReqInfo[4]*.

Note that the L2 never allocate on miss if the write is write-through type or is writing a partial data. Table 4.8 shows the L2 behaviors.

Table 4.8 L2 writes - partial data OR write-through cacheable (cca 0)

Allocation policy (*PB_MReqInfo[4]*)	L2 hit/miss	What L2 does
Don’t care	hit	update the cache. write out to memory if it is write-through type
	miss	no-allocate, just write out to the main memory

4.5.4.2 Read

MReqInfo[4] field also controls whether or not the L2 allocates read data after a miss in the cache array.

As shown in Table 4.9, on a read hit, there are no differences; the line will be returned to the core. However, when the read misses, depending on the value of *PB_MReqInfo[4],* the line that has been brought in from the main memory will end up residing in the L2 cache in the end of the operation.

Table 4.9 L2 reads - cacheable (cca 0/3)

L2 hit/miss	Allocation policy (*MReqInfo[4]*)	What L2 does
hit	Don’t care	Return the data. Keep the line in the L2.
miss	Allocate (0)	Get the data from memory. Return the data to the core. Allocate into the L2.
	No allocate (1)	Get the data from memory. Return the data to the core. Don’t allocate into the L2.

4.5.5 Write-Through vs. Write-Back

Write-through and write-back operations are both supported. The L2 decodes *MReqInfo[2:0]* fields and determines which way to handle the write data.

When a write hits in the L2 cache, the data is written into the L2 cache, and also sent to the main memory when it was write-through type (*MReqInfo[2:0]* = 0).

When a write misses, the no-write-allocation policy is employed in most cases. That is, the write data is forwarded to the main memory without updating the L2 cache contents. However, for the write-back type write with full line data, usually resulting from the L1 D-cache eviction, the L2 supports write-allocate on miss as well as the normal no-allocate policy. This is controlled by the value on *MReqInfo[4]* that is set by the OCP requester. Please refer to the Section 4.5.4 "L2 Allocation Policy" for more details.

4.5.6 Cacheable vs. Uncacheable vs. Uncached Accelerated

The L2 cache supports cacheable and uncacheable accesses. This information also is conveyed on the *MReqInfo[2:0]* field. Cacheable operations access the cache memories, whereas an uncached access bypasses the L2 cache arrays and is sent directly to the main memory.

Uncached accelerated accesses are treated the same way as non-accelerated uncached accesses. This CCA enables uncached transactions to better utilize bus bandwidth via burst transactions. L2 supports single-beat as well as 4-beat burst uncacheable transactions for both read and write operations.

4.5.7 Cache Aliases

The L2 cache is physically addressed and physically tagged. It is not subject to virtual aliasing.

4.5.8 Performance Counters

The L2 tracks and reports to core the number of the following events.

- the number of cached accesses
- the number of misses
- the number of write backs
- the amount of cycles the L2 is held due to misses
- the number of single bit errors that were corrected
- L2 pipeline utilization — Counts the number of starts into the TA stage of the L2 pipeline
- L2 hit qualifier — Counts different types of L2 cache hits and misses, crossed with the instruction being requested

4.5.9 Sleep Modes

The L2 cache contains two basic sleep modes:

- Instruction controlled sleep mode using the WAIT instruction
- Internal dynamic sleep mode

4.5.9.1 Sleep Mode Using the WAIT Instruction

In addition to slowing down or stopping the primary *cm_clk* input, software may initiate low-power Sleep Mode via the execution of the WAIT instruction in the processor.

When the processor enters into Sleep Mode, it will assert *SI_Sleep*. The *SI_Sleep* drives the *SI_L2_Sleep* input to the L2. The L2 then enters a low-power state and asserts the *L2_Sleep* output once all outstanding bus activity has completed. Most clocks in the L2 will be stopped, but a handful of flops will remain active to sense the wake up call from the processor, which is the deassertion of *SI_L2_Sleep*.

Power is reduced since the global clock goes to the vast majority of flops within the L2, which are held idle during this period. There is no bus activity while the L2 is in sleep mode, so the system bus logic which interfaces to the L2 could be placed into a low power state as well.

When the L2 samples *SI_L2_Sleep* asserted and there is no activity in the L2, the L2 will assert *L2_Sleep* two *cm_clks* later. Any activity in the L2 will delay the start of *L2_Sleep* assertion.

When *SI_L2_Sleep* is deasserted, the L2 will deassert *L2_Sleep* and assert *PB_SCmdAccept* two clocks later. If there is a valid *PB_MCmd* waiting at the L2 pins at the *cm_clk*, then the following *cm_clk* will have a coincident internal *l2_clk* edge (clocks are now enabled) and the command that was accepted is launched into the pipeline as indicated by *inst_ta*. The following clock after that will have an *l2_tram_clk* that initiates the tag ram access for that command. Thus, there is a four *cm_clk* latency from *SI_L2_Sleep* deassertion to the start of a tag ram access.

4.5.9.2 Internal Dynamic Sleep Mode

When there is no activity at the input pins of the L2 cache and all pending transactions from the CPU are completed, the L2 cache will eventually empty. When this occurs, the L2 cache will turn off the l2_clk signal after some small delay. Only data of value in the CMOS SRAM's retains state.

Beside the WAIT instruction induced sleep mode, the L2 is also equipped with the dynamic global clock gating. When there are no pending transactions in the L2 cache, the L2 shuts down the majority of internal clocks to save power. While the most part of the L2 cache can be turned off, the minimum required logic on the core-side OCP interface remain active. Thus, the L2 cache can accept a new OCP request from core at any time, and this will wake up the whole L2 cache controller.

4.5.10 Bypass Mode

Note: Bypass mode is strictly a debug feature and is not intended to be a normal mode of operation. It was not intended for active switching during normal operation.

Bypass mode is a test/bringup feature that causes the L2 cache to forward all requests received from either the core or the Coherency Manager to the OCP system interface to main memory. Entering or exiting from Bypass Mode other than at reset requires flushing of the L2 cache while running from uncached memory to restore the L2 cache state to a stable state. In bypass mode, all requests are forwarded to the system as received including L2 CACHE instructions and SYNCs.

4.5.11 L2 Cache Initialization

The L2 cache controller contains minimal hardware initialization logic. It normally relies on software to fully initialize the L2 arrays. The registers used to support cache initialization are described in Section 4.5.12, "L2 Cache CP0 Interface". For additional information, refer to the *CP0 Registers* chapter of this manual.

The L1 data cache must be initialized during power-up or reset in order to place the lines of the cache in a known state. This is accomplished via the cache initialization routine, which is normally part of the boot code. For experienced user's, a sample boot code is shown in the following subsection.

4.5.11.1 init_l2u Cache Initialization Routine

The following assembly provides an example initialization routine for the L2 cache.

```
LEAF(init_l2u)
      # Use CCA Override to allow cached execution of L2 init.
      # Check for CCA_Override_Enable by writing a one.
      lw r4_temp_data, 0x0008(r22_gcr_addr) # Read GCR_BASE register
      li r7_temp_mark, 0x50 # CM_DEFAULT_TARGET Memory
      # CCA Override Uncached enabled
      ins r4_temp_data, r7_temp_mark, 0, 8
      sw r4_temp_data, 0x0008(r22_gcr_addr)
      lw r4_temp_data, 0x0008(r22_gcr_addr) # GCR_BASE
      ext r4_temp_data, r4_temp_data, 4, 1 # Extract CCA_Override_Enable
      bnez r4_temp_data, done_l2 # Skip if CCA Override is implemented.
      nop
      b init_l2u
      nop
END(init_l2u)
```

4.5.11.2 init_l2c Cache Initialization Routine

The code in this function will be called from start.S after the L1 caches have been initialized. It will check to see if the core implements CCA Override. If it does, it will call the code to initialize the L2 cache.

```
LEAF(init_l2c)

      # Skip cached execution if CCA Override is not implemented.
      # If CCA override is not implemented the L2 cache would have already
      # been initialized when init_l2u was called.

      lw r4_temp_data, 0x0008(r22_gcr_addr) # Read GCR_BASE
      bnez r16_core_num, done_l2 # Only done from core 0.
      ext r4_temp_data, r4_temp_data, 4, 1 # CCA_Override_Enable
      beqz r4_temp_data, done_l2
      nop

END(init_l2c)
```

4.5.11.3 init_L2u Initialization Routine Details

This section provides a detailed description of each line of code in the init_l2u initialization routine.

The L2 cache is a system resource used by all cores in the proAptiv Multiprocessing System. Initialization of the L2 cache is done only by Core 0 in a CPS, because it only needs to be done once. The initialization of the L2 cache can be time consuming depending on its size. For example, a 256 KByte cache initializes quicker than an 8 MB cache.

The L2 cache initialization code executes faster if it is being run out of the instruction cache, so ideally the L2 initialization should be done after the L1 instruction cache in core 0 has been initialized. The instruction cache is a per-core resource and not initialized in the system initialization section of the code. Therefore, to be efficient and run the L2 cache initialization code out of the I-cache, the boot code tries to put off L2 cache initialization until the core 0 resources have been initialized. This can only be done if the L2 cache can be disabled before other cores are released to run this boot code. Otherwise there is a danger that other cores will use the L2 cache before it has been initialized by core 0.

The CCA override feature controls the cache attributes for the L2 cache. It allows for the disabling of the L2 cache by enabling the CCA override and setting the CCA to uncached. The CCA override works along with the L2 cache implementation.

The init_l2u function tries to enable the CCA override and set the L2 cache to uncached in the GCR_BASE register, thus disabling it. On systems that do not support CCA override, writes to the CCA override field have no effect, and reading back the GCR_BASE register will not show the CCA override being set.

The code reads the GCR Base register.

```
      lw r4_temp_data, 0x0008(r22_gcr_addr) # GCR_BASE
```

The next 3 lines of code are used to enable CCA Override and set the L2 cache CCA to uncached.

```
      li r7_temp_mark, 0x50 # CM_DEFAULT_TARGET Memory
      # CCA Override Uncached enabled
      ins r4_temp_data, r7_temp_mark, 0, 8
      sw r4_temp_data, 0x0008(r22_gcr_addr)
```

Now the code reads back the GCR_BASE register. If the CCA override bit is set, it means the code above worked, and the L2 cache is set to uncached. In this case, the code skips the initialization for now. The routine will be recalled later once the code is executing out of the L1 instruction cache. If not, the code branches to the init_l2 function, which initializes the L2 cache.

```
      lw r4_temp_data, 0x0008(r22_gcr_addr) # GCR_BASE
      ext r4_temp_data, r4_temp_data, 4, 1 # CCA_Override_Enable
      bnez r4_temp_data, done_l23 # Skip if CCA Override is implemented.
      nop
      b init_l2
      nop

END(init_l2u)
```

4.5.11.4 init_L2c Initialization Routine Details

This section provides a detailed description of each line of code in the init_l2c initialization routine. The code in this function is called from the start.S function after the L1 caches have been initialized. It checks to see if the core implements CCA Override. If it does, it calls the code to initialize the L2 cache.

In Section 4.5.11.3 the code also checks to see if CCA override was implemented, If it was not, then it initialized the L2 cache while the code was executing in uncached mode, so there is no need to do it again here.

```
LEAF(init_l2c)

      # Skip cached execution if CCA Override is not implemented.
      # If CCA override is not implemented the L2 cache
      # would have already been initialized when init_l2u was called.

      lw r4_temp_data, 0x0008(r22_gcr_addr)      # GCR_BASE
      bnez r16_core_num, done_l2                 # Only done from core 0
      ext r4_temp_data, r4_temp_data, 4, 1       # CCA_Override_Enable
      beqz r4_temp_data, done_l23 nop

END(init_l2c)
```

4.5.12 L2 Cache CP0 Interface

The proAptiv core uses different CP0 registers for L2 cache operations.

Table 4.1 L2 Cache CP0 Register Interface

CP0 Registers	CP0 number
Config2	*16.2*
ErrCtl	*26.0*
CacheErr	*27.0*
L23TagLo	*28.4*
L23DataLo	*28.5*
L23DataHi	*29.5*

This section describes the base processor core CP0 registers that support the L2 cache. A complete description and bit assignments for each register listed is described in Chapter 2, “CP0 Registers”.

4.5.12.1 Config2 Register (CP0 register 16, Select 2)

Asserting *Config2*.$_{L2B}$ (bit 12) enables the bypass-mode of the L2 cache. This bit is reflected on the *L2_Bypass* output from the core. When L2 goes into bypass-mode, L2 responds by asserting *L2_Bypassed* output, and the value or *L2_Bypassed* is returned when *Config2*.$_{L2B}$ is read by software. Thus, reading this *Config2*.$_{L2B}$ bit does not read back what was written: it reflects the value of a signal sent back from the L2. The feedback signal, *L2_Bypassed*, will reflect the previously written value with some implementation and clock ratio dependent delay.

Changing the value of *Config2*.$_{L2B}$ field in the middle of the normal operation may cause an unwanted loss of an OCP transaction in the L2 cache. For the safe transition into the L2 bypass-mode, an externalized SYNC before the MTC0 *Config2*.$_{L2B}$ is necessary to make sure all the pending transactions in L2 are completed. And, these instructions should run from the uncached space. It might be also a good idea to check if L2 is really in bypass-mode by reading the *Config2*.$_{L2B}$ field before moving onto the next instructions.

The *Config2.$_{SS}$* field (bits 11:8) indicates the number of sets per way in the data cache. The proAptiv L2 cache supports from 512 up to 32768 sets per way, which is used to configure cache sizes from 256 KBytes to 8 MBytes.

The *Config2.$_{SL}$* field (bits 7:4) indicates the line size for the L2 cache. The proAptiv L2 cache can be configured for a 32-byte or 64 byte line size.

The *Config2.$_{SA}$* field (bits 3:0) indicates the set associativity for the L2 cache. The proAptiv L2 cache is fixed at 8-way set associative as indicated by a default value of 4 for this field.

For more information, refer to Section 2.3.1.3, "Device Configuration 2 — Config2 (CP0 Register 16, Select 2)".

4.5.12.2 Error Control Register (CP0 register 26, Select 0)

ErrorControl.L2P (bit 23) is used to enable L2 ECC checking and correction. This bit is read-only if the L2 has not been built with ECC/Parity support. Specific parity support is enabled using both L2P and ErrorControl.PE (bit 31) as described in Table 4.10. L2P is also reflected on the *L2_ECCEnable* output from the core.

These encodings were chosen such that legacy code which is unaware of L2P, will by default enable L2 ECC logic when it enables L1 parity. For more information, refer to Section 2.3.6.10, "ErrCtl (CP0 Register 26, Select 0)"

Table 4.10 L2_ECC_Enable

PE	L2P	L2_ECCEnable
1	0	1
1	1	0
0	0	0
0	1	1

4.5.12.3 Cache Error Register (CP0 register 27, Select 0)

When the L2 detects an uncorrectable error, CacheError.EC is set, identifying the exception as an L2 error. The Cache Error register stores information such as the cache way where the error was detected, the cache index of the double word in which the error was detected, the cache level at which the error was detected, if the tag RAM was involved, etc.

For more information, refer to Section 2.3.6.11, "Cache Error — CacheErr (CP0 Register 27, Select 0)".

4.5.12.4 L23TagLo Register (CP0 register 28, Select 4)

The L23TagLo register contains the contents of the L2 tag array at the location accessed by the L2 Index Load Tag cache-op. It is also used as the source register for the L2 Index Store Tag cache-op.

For more information, refer to Section 2.3.6.7, "Level 2/3 Cache Tag Low — L23TagLo (CP0 Register 28, Select 4)".

4.5.12.5 L23DataHi Register(CP0 register 29, Select 5) / L23DataLo Register(CP0 register 28, Select 5)

For the L2 Index Load Tag cache-op, L23DataHi and L23DataLo hold the contents of the doubleword from the L2 data array at the indexed location. (L23DataHi holds the most-significant word and L23DataLo holds the least-significant word). For the L2 Index Load WS cache-op, L23DataHi and L23DataLo each hold the ECC parity of the doubleword from the L2 data array at the indexed location.

These registers are also used for the source data for the Index Store Data cache-op. Finally, L23DataLo is used as the data source for the ECC to be written by the Index Store ECC cache-ops. For more details on the data returned by the L2 on a Index Load Tag/Data cache-op, please refer to Section 4.6 "The CACHE Instruction".

For more information on the L23DataLo register, refer to Section 2.3.6.8, "Level 2/3 Cache Data Low — L23DataLo (CP0 Register 28, Select 5)". For more information on the L23DataHi register, refer to Section 2.3.6.9, "Level 2/3 Cache Data High — L23DataHi (CP0 Register 29, Select 5)".

4.5.13 L2 Cache Operations

Cache-ops are used for control operations such as initialization, invalidation, eviction, etc. A brief description of the cache-ops implemented by the L2 are given below:

Index Writeback Invalidate: If the state of the cache line at the specified index is valid and dirty, the line is written back to the memory address specified by the cache tag. After that operation is completed, the state of the cache line is set to invalid. If the line is valid but not dirty, the state of the line is set to invalid.

Index Load Tag: The tag, valid, lock, dirty, parity and LRU bits for the cache line at the specified index are read. The doubleword indexed in the data RAM is also read.

Index Load WS: The LRU, dirty, and dirty parity bits for the cache line at the specified index are read. ECC for the doubleword indexed in the data RAM is also read.

Hit Invalidate: If the cache contains the specified address, the state of that cache line is set to invalid.

Hit Writeback Inv: If the cache contains the specified address and it is valid and dirty, the contents of that line are written back to main memory. After that operation is completed, the state of the cache line is set to invalid. If the line is valid but not dirty, the state of the line is set to invalid.

Hit Writeback: If the cache contains the specified address and it is valid and dirty, the contents of that line are written back to main memory. After the operation is completed, the state of the line is left valid, but the dirty state is cleared.

Index Store Tag: Write the tag for the cache line at the specified index.

Index Store WS: Write the WS array for the cache line at the specified index.

Fetch And Lock: If the cache contains the specified address, lock the line. If the cache does not contain the specified address, refill the line from main memory and then lock the line.

Index Store Data: Write the data and ECC for the cache line at the specified index. Proper ECC is generated for the written data and written into the ECC field.

Index Store ECC: Write the ECC for the cache line at the specified index.

Most CP0 instructions are used rarely, in code which is not timing-critical. But an OS which has to manage caches around I/O operations or otherwise may have to sit in a tight loop issuing hundreds of `cache` operations at a time, so performance can be important.

4.5.13.1 Bus Transaction Equivalence

When the base processor executes an L2 CACHE instruction, the operands and as well as data to be written to CP0 registers is transferred to and from L2. Index Load Tag and Index Load WS generate burst read transactions. All other L2 cache-ops generate single write transactions.

For 64 byte line configurations, bit 5 (the LSB of the Index field) is the selector to which 32 byte half of the 64 byte line is targeted (essentially it becomes an additional DW bit). For *tag* and *ws* type cache-ops, this bit is disregarded and cache-ops with either value of bit 5 impact the exact same tag or ws entry. For data type cache-ops, bit 5 selects which half of the 64 byte cache line is being accessed.

Figure 4.5 Index Encoding for PB_MAddr (1MB, 8-way)

31 23	22 20	19 5	4 3	2 0
Unused	Way	Index	DW	Unused

4.5.13.2 Details of Cache-ops

Table 4.11 indicates the operation and behavior of the L2 cache for each cache-op.

Table 4.11 Cache-ops

Cache-op	Effective Address Operand Type	Operation
Index WB inv/ Indx Inv (OPCODE: 0)	INDEX	• If the state of the cache line at the specified index is valid and dirty, the line is written back to the memory address specified by the cache tag. After that operation is completed, the state of the cache line is set to invalid. • If the line is valid but not dirty, the state of the line is set to invalid • The LRU bits are updated to Least-recently-used. • The dirty bits are updated to clean for that way.
Index Load Tag (OPCODE: 1) ErrCtl.WST = 0	INDEX	• The tag, valid, lock, and parity fields from the tag array for the cache line at the specified index are written into L23TagLo. Furthermore, the dirty bit from the WS array corresponding to the specified index is also written into L23TagLo. (First beat of return data) • For the first beat of return data, the two halves of the 64-bit data bus are identical. • The indexed doubleword is written into {L23DataHi, L23DataLo}. (2nd beat of return data) • ErrCtl.PO is treated as a don’t care • The LRU bits are unchanged
Index Load WS (OPCODE: 1) ErrCtl.WST = 1	INDEX	• The dirty, dirty parity, and LRU fields from the WS array for the cache line at the specified index are written into L23TagLo. (First beat of return data) • For the first beat of return data, the two halves of the 64-bit data bus are identical. • The WS data at the indexed location is written into L23TagLo. (First beat of return data) • The indexed doubleword’s ECC is written into {L23DataHi, L23DataLo}. (2nd beat of return data) • ErrCtl.PO is treated as a don’t care • The LRU bits are unchanged • Data RAM: • The DW ECC to be read in the line is determined by *PB_MAddr[4:3]*

Table 4.11 Cache-ops*(continued)*

Cache-op	Effective Address Operand Type	Operation
Index Store Tag (OPCODE: 2) ErrCtl.WST = 0	INDEX	• The tag, valid, and lock fields in the Tag array at the indexed location are written from L23TagLo. • If ErrCtl.PO==1, the parity and total parity fields in the Tag array at the indexed location are written from L23TagLo. • If ErrCtl.PO==0, the parity and total parity fields in the Tag array at the indexed location are written with hardware generated values. • If valid==1, the LRU bits in the WS array are updated to make the indexed way most-recently-used. If valid==0, the LRU bits are updated with least-recently-used. • If valid==1, the dirty bit in the WS array at the indexed location is written from L23TagLo. • If valid==0, the dirty bit in the WS array at the indexed location is cleared. • The dirty parity bit in the WS array at the indexed location is written with the correct hardware generated values.
Index Store WS (OPCODE: 2) ErrCtl.WST = 1	INDEX	• The dirty and LRU fields for all 8 ways of the WS array at the indexed location are written from L23TagLo • If ErrCtl.PO==1, the dirty parity fields for all 8 ways of the WS array at the indexed location are written from L23TagLo • If ErrCtl.PO==0, the dirty parity fields for all 8 ways of the WS array at the indexed location are written with hardware generated values
Index Store Data (OPCODE: 3) ErrCtl.WST = 0	INDEX	• The doubleword in the data array at the indexed location and doubleword offset is written from {L23DataHi, L23DataLo} regardless of the PB_MDataByteEn value. • The Parity/ECC field in the data array at the indexed location and doubleword offset is written with a hardware generated value. • The LRU bits in the WS array are updated to make the indexed way most-recently-used.
Index Store ECC (OPCODE: 3) ErrCtl.WST = 1	INDEX	• The Parity/ECC field in the data array at the indexed location and doubleword offset is written from L23DataLo[7:0]. • The LRU bits in the WS array are updated to make the indexed way most-recently-used.
HIT Inv (OPCODE: 4)	ADDRESS	• If the address is not contained in L2, nothing happens. • If the address hits in L2, it is invalidated, the dirty bit is cleared, and the LRU bits in the WS array are updated to make the invalidated way least-recently-used. • If any arrays are written, the appropriate parity fields are updated by hardware.
HIT WB Inv (OPCODE: 5)	ADDRESS	• If the address is not contained in L2, nothing happens. • If the address hits in L2, and it is dirty, the line is written back to main memory. It is then invalidated, the dirty bit is cleared, and the LRU bits in the WS array are updated to make the invalidated way least -recently-used. • If the address hits in L2, and it is clean, it is invalidated and the LRU bits in the WS array are updated to make the invalidated way least-recently-used. • If any arrays are written, the appropriate parity fields are updated by hardware.
HIT WB (OPCODE: 6)	ADDRESS	• If the address is not contained in L2, nothing happens. • If the address hits in L2, and it is dirty, the line is written back to main memory, the dirty bit is cleared, and the LRU bits in the WS array are updated to make the invalidated way least -recently-used. • If the address hits in L2, and it is clean, nothing happens. • If any arrays are written, the appropriate parity fields are updated by hardware.
Fetch and Lock (OPCODE: 7)	ADDRESS	• If the address is not contained in L2, the line is refilled. The refilled line is then locked in the cache. The LRU bits in the WS array are updated to make the fetched way most-recently-used. The Dirty bit and the dirty parity bit are set to clean. • On a hit the line is locked and the operation retires. The LRU bits or the dirty bits are not affected.

4.5.13.3 Sync in L2

A Sync operation can be used to guarantee ordering of transactions. The L2 ensures that all transactions preceding a Sync request will be ordered in front of transactions received after the Sync request. Within the L2 only requests are ordered, not responses, i.e., there is no guarantee of the ordering between a read response vs. the Sync.

One example of the use of a Sync involves cache operations. Normally, the L2 does not guarantee the ordering between a cache operation, such as a Hit-Writeback-Invalidate, vs. an subsequent uncached request. If the software wants to ensure that any writes on the system interface due to the Hit-Writeback-Invalidate will be ordered in front of a subsequent uncached write, then a Sync must be issued between the cache operation and uncached write. Note that in order for a core to externalize a Sync request, *Config7.*$_{ES}$ bit must be set before the sync instruction.

The L2 issues a response to a Sync after all 3 of the following have completed:

- All previous requests have cleared the L2 pipeline
- The L2 has issued all requests to the system interface that are required by previous transactions, such as uncached requests, cache operations, cache misses, evictions, or previous Syncs.
- If the downstream system can take a sync OCP transaction (*L2_SyncTxEn=1*), it will externalize the sync transaction to the system once the above criteria has been satisfied. When the Sync response is received from the system interface, the L2 will return a Sync response to the processor interface.

4.5.13.4 L2 Cache Fetch and Lock

In the L2 cache, each line in a way can be locked independently. If a line is locked it will not be evicted. Software is not allowed to lock all available ways at the same cache index, since L2 would be unable to refill any other addresses at that index.

If the requested address is not contained in the L2 cache, the line is refilled and then locked in the cache. The LRU bits in the WS array are updated to make the fetched way most-recently-used. The dirty bit and the dirty parity bit are set to clean.

On a hit the L2 cache line is locked and the operation retires. The LRU bits or the dirty bits are not affected.

4.5.14 L2 Cache Error Management

This section describes ECC, parity, and bus error support for the L2 cache.

4.5.14.1 ECC/Parity Support

If ECC/Parity support is selected at build time, and this support is enabled via software by setting the *ErrCtl.*$_{L2EccEn}$ bit in the Error Control register (CP0 register 26, Select 0), then the tag and the data arrays are protected with single-error correction logic as well as double-error detection logic. The Way Select RAM is protected with single-error detection logic. Correctable errors are not reported to the processor, but uncorrectable errors are reported to the processor. If ECC/Parity support is either not selected at build time or disabled, then no errors are detected (or corrected) on any of the cache arrays.

The ECC logic uses Hamming’s error correcting code. In the data array, each 64-bit doubleword is independently ECC protected. This requires 8 parity bits per doubleword. The tag array requires 6 parity bits.

To perform a single detection and correction the parity bits are placed at 2^n locations among the data bits. The bits at different locations are then grouped together. The grouping is done by analyzing the binary weights of the particular location.

For example, to protect 8 data bits, 4 parity bits are needed which will be placed as below:

Table 4.12 Parity Bit Distribution

Bit Location	12	11	10	9	8	7	6	5	4	3	2	1
Parity and data bits	d7	d6	d5	d4	p3	d3	d2	d1	p2	d0	p1	p0

Note that Bit location 0 does not exist.

The binary weight of bit location 3 is 2^0 and 2^1, which is derived from its binary value 0011b. Therefore, bit location 3 falls in group g0 and g1. Similarly, Bit location 11 falls into groups g0, g1 and g3.

Parity bit p0 will belong to g0 and its value will is generated such that g0 will have an even parity. Similarly all other parity bits are generated such that their respective group ends up in even parity.

This sharing of binary weights across groups enables the L2 to determine precisely which data or parity bit was in error. That is achieved by recreating the parity bits from the data read from the memory and XORing it with the parity bits read from the memory. The XORed value, or the syndrome, points to the bit in error. Once this error is detected the L2 corrects it. A value of zero on the syndrome indicates that there was no error in the parity and data bits.

To achieve double bit error detection an even parity is generated across the parity and data bits, which is termed as the total parity bit. The total parity bit will be flipped in case of a single bit error, whereas for a double bit error it will remain the same. The syndrome along with the total parity bit is then used to detect a double bit error.

The WSRAM’s dirty bits are protected, whereas the LRU bits are not. For each dirty bit there is one more bit added called the dirty parity bit. The value of the dirty parity bit enforces even parity protection.

4.5.14.2 Tag, Data, and WS Array Format

Logical Tag Array Format

The width of the tag in an 8 way 128 MB cache is 18 bits per way. The data array format is as shown in Figure 4.13.

Table 4.13 Logical Tag Array Format for a 8 Way 128 MB Cache

Bit position	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
Content	TP	L	V	d17	d16	d15	d14	d13	d12	d11	p4	d10	d9	d8	d7	d6	d5	d4	p3	d3	d2	d1	p2	d0	p1	p0

Where,	d0-d17	: Tag
	V	: Valid bit
	L	: Lock bit
	p0-p4	: parity bits
	TP	: Total parity bit

For larger caches, the width of the tag reduces. In that case, the upper data bits are ignored from the calculation as appropriate.

Logical Data Array Format

The data array format is as shown in Figure 4.14.

Table 4.14 Logical Data Array Format

Bit position	72	71..65	64	63:33	32	31:17	16	15..9	8	7..5	4	3	2	1
Content	TP	[63:57]	p6	[56:26]	p5	[25:11]	p4	[10:4]	p3	[3:1]	p2	[0]	p1	p0

4.5.14.3 Cache Parity Error Handling

The three types of memory arrays in the L2 have an option for parity. If selected, this option provides single bit correction and double bit detection of the tag rams and data rams.

- The Tag RAM coverage is for each way.
- The Data RAM coverage is for each way and each double-word in each way.
- The Way Select RAM has parity for each dirty bit. A correctable bit failure is corrected and no notification of this event is present at the L2 pins.

The five types of ECC/parity errors are handled internally as follows.

Single Bit (correctable) Tag RAM Error in a Way

The corrected tag value is written back into the tag ram by replaying the request.

Single Bit (correctable) Data RAM Error in a Dword of a Way

The corrected data value is written back into the data ram by replaying the request. This may occur due to a read that hits or a partial write where a dword in the way has a single bit failure.

Double Bit (uncorrectable) Tag RAM Error in a Way

An uncorrectable tag ram failure kills the request in the L2 pipeline. A write request is dropped and a read request is treated as a hit to an arbitrary way.

Double Bit (uncorrectable) Data RAM Error in a Dword of a Way

For a read hit, the uncorrected data is returned.

Parity Error (uncorrectable) on a Dirty Bit in the Wsram

When a dirty parity error is detected, the L2 treats the state as dirty by default. This means that a victim line being evicted due to either allocation or invalidation by a request might not have really have needed to be written.

4.5.14.4 Multiple Uncorrectable Errors

This error is reported when more than one uncorrectable error is being reported on the same L2 clock cycle. Since double-bit Tag RAM errors, double-bit Data RAM error, and parity bit errors in the Way Select RAM are each reported in different L2 pipeline stages, this assertion indicates that different requests have encountered uncorrectable requests. In other words, if a single request suffers all three uncorrectable errors, the error will be reported three times.

4.5.14.5 Bus Error Handling

Bus errors are never originated by the L2. However, bus errors may be received from the system on an OCP read from the L2 to the system. The error is indicated when the read-data is returned back to the L2. The L2 propagates the bus error when returning data to the processor or CM2.

If a bus error is received on a 64-byte burst read to the system, the L2 signals the bus error for the processor read that originated the request. If the L2 receives a subsequent read to the same 64-byte cache line before all the data has been received from memory for the previous request, the new request also receives a bus error response.

In general, a bus error reported in a system response due to a processor/CM request is considered to be reporting the entire cache line as having a bus error. However, if the original request is satisfied before the L2 detects the system bus error, then the response to the processor/CM will not have a bus error.

There is no capability for signalling bus errors on writes.

4.6 The CACHE Instruction

The L1 instruction, L1 data, and L2 caches in the proAptiv Multiprocessing System support the CACHE instruction, which allow users to manipulate the contents of the Data and Tag arrays, including the locking of individual cache lines. The behavior of the CACHE instruction is identical for both the L1 instruction and data caches.

4.6.1 Decoding the Type of Cache Operation

The type of cache operation performed is encoded using a combination of the 5-bit *op* field of the CACHE instruction, and selected bits from the *ErrCtl* register (CP0 Register 26, Select 0). In addition to performing operations on the L1 instruction, L1 data, and L2 caches themselves, there are other CACHE operations that are performed on internal memories such as the way selection RAM, the scratch pad RAM, and the Dirty Bit RAM. The *ErrCtl* bits determine the type internal memory where the CACHE operation will be performed.

The selected bits of the ErrCtl register used to determine the type of CACHE operation are as follows:

- Bit 29, *WST*: If this bit is set, execution of a **`cache IndexLoadTag`** or **`cache IndexStoreTag`** instruction reads or writes the cache's internal *way-selection RAM* instead of the cache tags.
- Bit 28, *SPR*: If this bit is set, index-type cache instructions work on the data scratch pad (*DSPRAM)* and instruction scratch pad *(ISPRAM*), if implemented. Read the *$Config_{DSP}$* and *$Config_{ISP}$* bits to determine if the associated scratch pad RAM is present.
- Bit 21, *DYT*: Setting this bit allows **`cache`** load/store data operations to work on the "dirty array" associated with the L1 data cache.

4.6.2 CACHE Instruction Opcodes

Refer to the implementation-specific CACHE instruction at the back of this manual for a list of CACHE instruction opcodes.

4.6.3 Way Selection RAM Encoding

The CACHE Index Load Tag and Index Store Tag instructions can be used to read and write the Way Select (WS) RAM by setting the *WST* bit in the *ErrCtl* register. Similarly, the *SPR* bit in the *ErrCtl* register causes the **`Index Load Tag`** and **`Index Store Tag`** instructions to read the pseudo-tags associated with the scratch-pad 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 4.15.

Table 4.15 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

Table 4.15 Way Selection Encoding, 4 Ways*(continued)*

Selection Order[1]	WS[5:0]	Selection Order	WS[5:0]
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.

4.7 Write Back Buffer

The Bus Interface Unit (BIU) includes a Write Back Buffer (WBB) that holds data from the L1 cache that is going to memory. This includes evictions from the data cache, uncached stores, and uncached accelerated stores. The WBB consists of eight 32-byte entries. The WBB also holds L2 **`CACHE`** instructions that are to be sent out on the bus. The WBB gathers uncached accelerated (UCA) stores to allow full line burst writes.

WBB entries are ‘flushed’ under a variety of conditions. When a buffer is flushed, the write command is queued in the BIU and the WBB entry will not accept any more activity until the data has been written to the bus and the buffer is freed up. Some flush conditions are shown here:

- Uncached (non-accelerated) stores flush immediately
- L2 CACHE instruction commands are also flushed immediately
- Entries for L1 data cache evictions are flushed when all 4 double-words (32B) of data have been gathered

When coherence is enabled, the CPU is the ‘owner’ of a cache line until the self-intervention for the writeback request has been seen. The WBB entry cannot be deallocated until that point so that the CPU can respond with the data if another CPU requests it. The WBB is also used for staging data responses to interventions. To avoid deadlock, one WBB entry must be reserved for this purpose.

Chapter 5

Exceptions and Interrupts

The proAptiv Multiprocessing System 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:

- Section 5.1 “Exception Conditions”
- Section 5.2 “TLB Read Inhibit, Execute Inhibit and FTLB Parity Exceptions”
- Section 5.3 “Exception Priority”
- Section 5.4 “Exception Vector Locations”
- Section 5.5 “General Exception Processing”
- Section 5.6 “Debug Exception Processing”
- Section 5.7 “Exception Descriptions”
- Section 5.8 “Exception Handling and Servicing Flowcharts”
- Section 5.9 “Interrupts”

5.1 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 CPU aborts that instruction and all instructions that follow. When the instruction graduates, 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 program. An instruction taking an exception may itself be aborted 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.

5.2 TLB Read Inhibit, Execute Inhibit and FTLB Parity Exceptions

The proAptiv core supports the following new types of exceptions listed below:

- TLB Execute-Inhibit
- TLB Read-Inhibit
- FTLB Parity

The *TLB Execute Inhibit* exception (TLBXI) is taken when there is a TLB hit during an instruction fetch, the XI bit of the entry is set, the Valid (V) bit is set, and the $PageGrain_{EIC}$ bit is set. If the $PageGrain_{EIC}$ bit is cleared, a *TLBL* exception is taken. This type of exception is used by the operating system to prevent execute accesses to a particular page. Refer to Section 5.7.12 "TLB Execute-Inhibit Exception" for more information.

The *TLB Read Inhibit* exception (TLBRI) is taken when there is a TLB hit during a read operation, the RI bit of the entry is set, the Valid (V) bit is set, and the $PageGrain_{EIC}$ bit is set. If the $PageGrain_{EIC}$ bit is cleared, a *TLBL* exception is taken. This type of exception is used by the operating system to prevent read accesses from a particular page. Refer to Section 5.7.13 "TLB Read-Inhibit Exception" for more information.

An *FTLB Parity* exception is taken whenever a parity error is detected on an FTLB read. The error can occur in either the FTLB Tag RAM or FTLB Data RAM. The FTLB parity exception is taken only when bit 31 of the CP0 *Error Control* register ($ErrCtl._{PE}$) is set. If this bit is cleared, FTLB parity errors are ignored. Refer to Section 5.7.14 "FTLB Parity Exception" for more information.

5.3 Exception Priority

Table 5.1 contains a list and a brief description of all exception conditions, The exceptions are listed in the order of their relative priority, from highest priority (Reset) to lowest priority (Load/store bus error). When several exceptions occur simultaneously, the exception with the highest priority is taken.

Table 5.1 Priority of Exceptions

Exception	Description
Reset	Assertion of SI_Reset signal.
DSS	EJTAG Debug Single Step.

Table 5.1 Priority of Exceptions*(continued)*

Exception	Description
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.
IFTLBPAR	FTLB instruction fetch parity error.
TLBL	Fetch TLB miss. Fetch TLB hit to page with V=0.
TLBXI	TLB Execute Inhibit. Occurs when there is an execute access from a page table whose XI bit is set.
I-cache 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.
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.
DFTLBPAR	FTLB data load/store parity error.
TLBL	Load TLB miss. Load TLB hit to page with V=0
TLBS	Store TLB miss. Store TLB hit to page with V=0.

Table 5.1 Priority of Exceptions*(continued)*

Exception	Description
TLBRI	TLB Read Inhibit. Occurs when there is an attempt to access a page table whose RI bit is set.
TLB Mod	Store to TLB page with D = 0.
D-cache Error	Cache parity error. Imprecise.
DBE	Load or store bus error. Imprecise.

5.4 Exception Vector Locations

The location of the exception vector in the proAptiv core depends on the operating mode. If the core is in the legacy setting, the exception vector location is the same as in previous MIPS processors. However, if the core is configured for Enhanced Virtual Address (EVA), the exception vector can effectively be placed anywhere within kernel address space.

The *SI_EVAReset* pin determines the addressing scheme and whether the device boots up in the legacy setting or the EVA setting. The legacy setting is defined as having the traditional MIPS virtual memory map used in previous generation processors. The EVA setting places the device in the enhanced virtual address configuration, where the initial size and function of each segment in the virtual memory map is determined from the segmentation control registers (*SegCtl0* - *SegCtl2*).

If the *SI_EVAReset* pin is deasserted at reset, the proAptiv core comes up in the legacy configuration and hardware takes the following actions:

- The *CONFIG5.*$_K$ bit becomes read-write and is programmed by hardware to a value of 0 to indicate the legacy configuration. In this case, the cache coherency attributes for the kseg0 segment are derived from the *Config.*$_{K0}$ field as described in the previous subsection. In addition to selecting the location of the cache coherency attributes, the *CONFIG5.*$_K$ bit also causes hardware to generate two boot exception overlay segments, one for kseg0 and one for kseg1, as described in Section 3.7, "Boot Exception Vector Relocation in Kernel Mode".
- Hardware programs the CP0 memory segmentation registers (*SegCtl0* - *SegCtl2*) for the legacy setting. Note that these registers are new in the proAptiv core and are not used by legacy software. However, they are used by hardware during normal operation, so their default values should not be changed.

If the *SI_EVAReset* pin is asserted at reset, the proAptiv core comes up in the EVA configuration (default is *xkseg0* space = 3 GB) and hardware takes the following actions:

- The *CONFIG5.*$_K$ bit becomes read-only and is forced to a value of 1 to indicate the EVA configuration. In this case, the *CONFIG.*$_{K0}$ field is ignored and is no longer used to determine the kseg0 cache coherency attributes (CCA). Rather, the values in bits 2:0 (segments 0, 2, and 4) and bits 18:16 (segments 1, 3, and 5) of the *SegCtl0* - *SegCtl2* registers are used to define the CCA for each memory segment. In this case, hardware generates only one BEV overlay segment as described in Section 3.7, "Boot Exception Vector Relocation in Kernel Mode".
- Hardware sets the CP0 memory segmentation registers (*SegCtl0* - *SegCtl2*) for the EVA configuration.

When the *SI_UseExceptionBase* pin is 0 and the *Config5.*$_K$ bit is cleared, the device is in legacy mode. In this mode the exception vector location defaults of 0xBFC0_0000 and the *SI_ExceptionBase[31:12]* pins are ignored.

When the *SI_UseExceptionBase* pin is 1 and the *Config5.*$_K$ bit is cleared, the device is still in legacy mode, but the *SI_ExceptionBase[29:12]* pins are used to indicate the location of the exception vector. Bits 31:30 are forced to a value of 2'b10, placing the exception vector somewhere in kseg0/kseg1 space.

If the *Config5*.$_K$ bit is set, the device is in EVA mode. In this case the *SI_UseExceptionBase* pin is ignored and the *SI_ExceptionBase[31:12]* pins are used to derive the location of the exception vector.

The function of the *Config5*.$_K$ bit and the *SI_LegacyUseExceptionBase* pin is shown in Table 5.2. For more information on EVA mode, refer to the MMU chapter.

Table 5.2 *SI_UseExceptionBase* Pin and CONFIG5.K Encoding

CONFIG5.K Bit	*SI_UseExceptionBase* Pin	Condition	Action
0	0	Legacy Mode *SI_ExceptionBase[31:12]* pins are not used.	Use default BEV location of 0xBFC0_0000.
0	1	Legacy Mode Use only *SI_ExceptionBase[29:12]* for the BEV base location. Bits 31:30 are forced to a value of 2’b10 to put the BEV vector into KSEG0/KSEG1 virtual address space.	The BEV location is determined as follows: *SI_ExceptionBase[31:12]* = 2’b10, *SI_ExceptionBase[29:12]* pins, 12’b0 Bits 31:30 are forced to a value of 2’b10 to put the BEV vector into KSEG0/KSEG1 virtual address space.
1	Don’t care	EVA Mode Use *SI_ExceptionBase[31:12]* pins.	The *SI_ExceptionBase[31:12]* pins are used directly to derive the BEV location. The *SI_LegacyUseExceptionBase* pin is ignored.

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*, *SI_ExceptionBase*[31:12], and *SI_ExceptionBaseMask*[27:20].

In the legacy setting, when the *SI_UseExceptionBase* pin is 0, the Reset, Soft Reset, NMI, and EJTAG Debug exceptions are vectored to a specific location, as shown in Table 5.3. Addresses for all other exceptions are a combination of a vector offset and a vector base address. In the proAptiv core, software is allowed to specify the vector base address via the *EBase* register for exceptions that occur when *Status*$_{BEV}$ equals 0. Table 5.3 shows the vector base address when the core is in legacy setting and the *SI_UseExceptionBase* pin is 0.

Table 5.4 shows the vector base addresses when the core is in legacy setting and the *SI_UseExceptionBase* equals 1. As can be seen in Table 5.4, when *SI_UseExceptionBase* equals 1, the exception vectors for cases where *Status*$_{BEV}$ = 0 are not affected.

Table 5.3 Exception Vector Base Addresses — Legacy Mode, SI_UseExceptionBase = 0

Exception	Status$_{BEV}$	
	0	1
Reset, NMI	`0xBFC0.0000`	
EJTAG Debug (with *ProbEn* = 0, in the EJTAG_Control_register and *DCR.RDVec*=0)	`0xBFC0.0480`	

Table 5.3 Exception Vector Base Addresses — Legacy Mode, SI_UseExceptionBase = 0 *(continued)*

Exception	Status$_{BEV}$	
	0	1
EJTAG Debug (with *ProbEn* = 0, in the EJTAG_Control_register and *DCR.RDVec*=1)	*DebugVectorAddr* `[31:7] || 2b0000000`	
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 of `2b’10`	`0xBFC0.0300`
Other	$EBase_{31\ 12}$ || `0x000` Note that $EBase_{31\ 30}$ have the fixed value of 2’b`10`	`0xBFC0.0200`
‘||’ denotes bit string concatenation		

In legacy mode, when the *SI_UseExceptionBase* pin is 0, the Reset, Soft Reset, NMI, and EJTAG Debug exceptions are vectored to a specific location, as shown in Table 5.4.

Table 5.4 Exception Vector Base Addresses — Legacy Mode, 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 and *DCR.RDVec*=0)	`0b10 ||`*SI_ExceptionBase*`[29:12] || 0x480`	
EJTAG Debug (with *ProbEn* = 0 in the EJTAG_Control_register and *DCR.RDVec*=1)	*DebugVectorAddr*`[31:7] || 2b0000000`	
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 `2’b10. Exception vector resides in kseg1.`
		`0b101 ||` *SI_ExceptionBase*`[28:12] || 0x300`
		`Exception vector resides in kseg1.`

Other	$EBase_{31\ 12}$ || `0x000`
	Note that $EBase_{31\ 30}$ have the fixed value `2’b10`
	`Exception vector resides in kseg0/kseg1.`
		`0b10 ||` *SI_ExceptionBase*`[29:12] || 0x200`
		`Exception vector resides in kseg0/kseg1.`

‘||’ denotes bit string concatenation		

Table 5.5 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 5.25 (on page 334) shows the offset from the base address in the case where $Status_{BEV}$ = 0 and $Cause_{IV}$ = 1. Table 5.7 combines these three 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 5.5 Exception Vector Offsets

Exception	Vector Offset
TLB Refill, *EXL* = 0	`0x000`
General Exception	`0x180`
Interrupt, $Cause_{IV}$ = 1	`0x200` (In Release 3 implementations, this is the base of the vectored interrupt table when $Status_{BEV}$ = 0)
Reset, NMI	None (uses reset base address)

In EVA mode, when the *SI_UseExceptionBase* pin is ignored and the Reset, Soft Reset, NMI, and EJTAG Debug exceptions are vectored to a location determined by the programming of the three Segment Control registers (*SegCtl0* - *SegCtl2*), as shown in Table 5.6.

Table 5.6 Exception Vector Base Addresses — EVA Mode

Exception	$Status_{BEV}$	
	0	**1**
Reset, NMI	*SI_ExceptionBase* `[31:12] || 0x000`	
EJTAG Debug (with *ProbEn* = 0 in the EJTAG_Control_register and *DCR.RDVec*=0)	*SI_ExceptionBase* `[31:12] || 0x480`	
EJTAG Debug (with *ProbEn* = 0 in the EJTAG_Control_register and *DCR.RDVec*=1)	*DebugVectorAddr* `[31:7] || 2b0000000`	
EJTAG Debug (with *ProbEn* = 1 in the EJTAG_Control_register)	`0xFF20.0200`	
Cache Error	$EBase_{31\ \ 12}$ ‖ `0x000`	*SI_ExceptionBase* `[31:12] || 0x300` `(Forced uncached)`
Other	$EBase_{31\ \ 12}$ ‖ `0x000`	*SI_ExceptionBase* `[31:12] || 0x200`
'‖' denotes bit string concatenation		

Table 5.7 Exception Vectors

Exception	Config5$_{K}$	SI_LegacyUseExceptionBase	Status$_{BEV}$	Status$_{EXL}$	Cause$_{IV}$	EJTAG ProbEn	Vector (IntCtl$_{VS}$ = 0)
Reset, NMI	0	0	x	x	x	x	`0xBFC0.0000`
Reset, NMI	0	1	x	x	x	x	`2’b10 ||` *SI_ExceptionBase* `[29:12] || 0x000`
Reset, NMI	1	x	x	x	x	x	*SI_ExceptionBase* `[31:12] || 0x000`
EJTAG Debug	0	0	x	x	x	0	`0xBFC0.0480 (if` *DCR.RDVec* `=0)` *DebugVectorAddr* `[31:7] || 2b0000000 (if` *DCR.RDVec* `=1)`
EJTAG Debug	0	1	x	x	x	0	`2’b10 ||` *SI_ExceptionBase* `[29:12] || 0x480 (if` *DCR.RDVec* `=0)` *DebugVectorAddr* `[31:7] || 2b0000000 (if` *DCR.RDVec* `=1)`
EJTAG Debug	1	x	x	x	x	0	*SI_ExceptionBase* `[31:12] || 0x480 (if` *DCR.RDVec* `=0)` *DebugVectorAddr* `[31:7] || 2b0000000 (if` *DCR.RDVec* `=1)`
EJTAG Debug	x	x	x	x	x	1	`0xFF20.0200`
TLB Refill	x	x	0	0	x	x	*EBase* `[31:12] || 0x000`
TLB Refill	x	x	0	1	x	x	*EBase* `[31:12] || 0x180`
TLB Refill	0	0	1	0	x	x	`0xBFC0.0200`
TLB Refill	0	1	1	0	x	x	`2’b10 ||` *SI_ExceptionBase* `[29:12] || 0x200`
TLB Refill	1	x	1	0	x	x	*SI_ExceptionBase* `[31:12] || 0x200`
TLB Refill	0	0	1	1	x	x	`0xBFC0.0380`
TLB Refill	0	1	1	1	x	x	`2’b10 ||` *SI_ExceptionBase* `[29:12] || 0x380`
TLB Refill	1	x	1	1	x	x	*SI_ExceptionBase* `[31:12] || 0x380`
Cache Error	0	x	0	x	x	x	*EBase* `[31:30] || 0b1 ||` *EBase* `[28:12] || 0x100`
Cache Error	1	x	0	x	x	x	`0xBFC0.0100`
Cache Error	0	0	1	x	x	x	`0xBFC0.0300`
Cache Error	0	1	1	x	x	x	`2’b101 ||` *SI_ExceptionBase* `[28:12] || 0x300`
Cache Error	1	x	1	x	x	x	*SI_ExceptionBase* `[31:12] || 0x300`
Interrupt	x	x	0	0	0	x	*EBase* `[31:12] || 0x180`
Interrupt	x	x	0	0	1	x	*EBase* `[31:12] || 0x200`
Interrupt	0	0	1	0	0	x	`0xBFC0.0380`
Interrupt	0	1	1	0	0	x	`2’b10 ||` *SI_ExceptionBase* `[29:12] || 0x380`
Interrupt	1	x	1	0	0	x	*SI_ExceptionBase* `[31:12] || 0x380`
Interrupt	0	0	1	0	1	x	`0xBFC0.0400`
Interrupt	0	1	1	0	1	x	`2’b10 ||` *SI_ExceptionBase* `[29:12] || 0x400`

Table 5.7 Exception Vectors *(continued)*

Exception	$Config5_K$	SI_LegacyUseExceptionBase	$Status_{BEV}$	$Status_{EXL}$	$Cause_{IV}$	EJTAG ProbEn	Vector ($IntCtl_{VS}$ = 0)
Interrupt	1	x	1	0	1	x	*SI_ExceptionBase* `[31:12] || 0x400`
All others	x	x	0	x	x	x	*EBase* `[31:12] || 0x180`
All others	0	0	1	x	x	x	`0xBFC0.0380`
All others	0	1	1	x	x	x	`2’b10 ||` *SI_ExceptionBase* `[29:12] || 0x380`
All others	1	x	1	x	x	x	*SI_ExceptionBase* `[31:12] || 0x380`

‘x’ denotes don’t care,
‘||’ denotes bit string concatenation

5.5 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. The value loaded into the *EPC* register is dependent on whether the processor implements the MIPS16 Module, and whether the instruction is in the delay slot of a branch or jump which has delay slots. Table 5.8 shows the value stored in each of the CP0 PC registers, including *EPC*.

  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.

Table 5.8 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 the EPC nor Cause$_{BD}$ are modified */
if Status$_{EXL}$ = 1 then
    vectorOffset ← 0x180
else
    /* For implementations that include the MIPS16e Module, calculate potential */
    /* PC adjustment for exceptions in the delay slot */
    if Config1$_{CA}$ = 0 then
        restartPC ← PC
        branchAdjust ← 4        /* Possible adjustment for delay slot */
    else
```

```
        restartPC ← $PC_{31..1}$ || ISAMode
        if (ISAMode = 0) or ExtendedMIPS16Instruction
            branchAdjust ← 4    /* Possible adjustment for 32-bit MIPS delay slot */
        else
            branchAdjust ← 2    /* Possible adjustment for MIPS16 delay slot */
        endif
    endif
    if InstructionInBranchDelaySlot then
        EPC ← restartPC - branchAdjust/* PC of branch/jump */
        $Cause_{BD}$ ← 1
    else
        EPC ← restartPC              /* PC of instruction */
        $Cause_{BD}$ ← 0
    endif

    /* Compute vector offsets as a function of the type of exception */
    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 ← $Cause_{RIPL}$
                else
                    VecNum ← VIntPriorityEncoder()
                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 */
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 ← $vectorBase_{31..30}$ || ($vectorBase_{29..0}$ + $vectorOffset_{29..0}$)
```

```
                                        /* No carry between bits 29 and 30 */
```

5.6 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 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
   if $DebugControlRegister_{RDVec}$ = 1 then
      if CacheErr then
         PC ← 2#101 || $DebugVectorAddr_{28..7}$ || 2#0000000
      else
         PC ← 2#10 || $DebugVectorAddr_{29..7}$ || 2#0000000
   else
      if SI_UseExceptionBase
         if CacheErr then
```

```
                PC ← 2#101 || SI_ExceptionBase[28:12] || 0x000
            else
                PC ← 2#10 || SI_ExceptionBase[29:12] || 0x000
        else
            PC ← 0xBFC0_0480
endif
```

The location of the debug exception vector is determined by the *ProbTrap* bit in the *EJTAG Control* register (*ECR*) and the *RDVec* bit in the *Debug Control* register (*DCR*), as shown in Table 5.9.

Table 5.9 Debug Exception Vector Addresses

ProbTrap bit in ECR Register	RDVec bit in DCR Register	Debug Exception Vector Address
0	0	0xBFC0 0480
0	1	DebugVectorAddr$_{31\ 7}$ ‖ 0000000
1	0	0xFF20 0200 in dmseg
1	1	

The value in the optional drseg register *DebugVectorAddr* (offset 0x00020) is used as the debug exception vector when the *ECR ProbTrap* bit is 0 and when enabled through the optional *RDVec* control bit in the *Debug Control Register* (*DCR*). Bit 0 of *DebugVectorAddr* determines the ISA mode used to execute the handler. Figure 5.1 shows the format of the *DebugVectorAddr* register; Table 5.10 describes the *DebugVectorAddr* register fields.

Figure 5.1 DebugVectorAddr Register Format

31	30	29 … 7	6 …	0
1	0	DebugVectorOffset	0	IM

Table 5.10 DebugVectorAddr Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
1	31	Ignored on write; returns one on read.	R	1
DebugVectorOffset	29:7	Programmable Debug Exception Vector Offset	R/W	Preset to 0x7F8009
IM	0	ISA mode to be used for exception handler	R	0
0	30,6:1	Ignored on write; returns zero on read.	R	0

Bits 31..30 of the *DebugVectorAddr* register are fixed with the value 0b10, and the addition of the base address and the exception offset is done inhibiting a carry between bit 29 and bit 30 of the final exception address. The combination of these two restrictions forces the final exception address to be in the kseg0 or kseg1 unmapped virtual address segments. For cache error exceptions, bit 29 is forced to a 1 in the ultimate exception base address, so that this exception always runs in the kseg1 unmapped, uncached virtual address segment.

When MIPS16 is implemented, the power-up state of *IM* is zero. If the implementation does not include MIPS16, the *IM* field is read-only, should be written with zero and will return 0 on a read.

If the TAP is not implemented, the debug exception vector location is as if *ProbTrap*=0.

5.7 Exception Descriptions

The following subsections describe each of the exceptions listed in the same sequence as shown in Table 5.1.

5.7.1 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 (exact vector address depends on mode of operation - Legacy/EVA)

Operation:

```
Random ← TLBEntries - 1
Wired ← 0
Config ← ConfigurationState
$Status_{RP}$ ← 0
$Status_{BEV}$ ← 1
$Status_{TS}$ ← 0
$Status_{SR}$ ← 0
$Status_{NMI}$ ← 0
$Status_{ERL}$ ← 1
$WatchLo_{I}$ ← 0
$WatchLo_{R}$ ← 0
$WatchLo_{W}$ ← 0
if InstructionInBranchDelaySlot then
    ErrorEPC ← PC - 4
```

```
else
    ErrorEPC ← PC
endif
PC ← 0xBFC0_0000
```

5.7.2 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

5.7.3 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

5.7.4 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 (exact vector address depends on mode of operation - Legacy/EVA)

Operation:

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

5.7.5 Machine Check Exception

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

- A TLBWI instruction to the FTLB and the index and VPN2 are not consistent and the EHINV bit is not set. See Section 3.12 of the MMU chapter.

- A TLBWI instruction to the FTLB and the PageMask register does not correspond to the FTLB page size setting in bits 12:8 of the *Config4* register (*Config4*$_{FTLB\ Page\ Size}$)
- A TLBP instruction and a duplicate/overlap is detected across the FTLB/VTLB.
- Any TLB lookup and a duplicate/overlap is detected across the FTLB/VTLB.

The machine check exception can be either precise or imprecise depending on the type of error.

The machine check exception is imprecise on:

- – A Load/Store Unit (LSU) or Instruction Fetch Unit (IFU) lookup matching duplicate entries

The machine check exception is precise on:

- – TLBP matching duplicate entries.
- – TLBWI to the FTLB with the page size != the FTLB page size.
- – TLBWI to the FTLB with EHINV=0 and the FTLB set implied by the VPN not the same as the set implied by the index.

Cause Register ExcCode Value:

MCheck

Additional State Saved:

None

Entry Vector Used:

General exception vector (offset 0x180)

5.7.6 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 5.9 “Interrupts” on page 328 for more details about the processing of interrupts.

Register ExcCode Value:

Int

Additional State Saved:

Table 5.11 Register States an Interrupt Exception

Register State	Value
CauseIP	Indicates the interrupts that are pending.

Entry Vector Used:

See 5.9.2 “Generation of Exception Vector Offsets for Vectored Interrupts” on page 334 for the entry vector used, depending on the interrupt mode the processor is operating in.

5.7.7 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

5.7.8 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 5.12 Register States on Watch Exception

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

Entry Vector Used:

General exception vector (offset 0x180)

5.7.9 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
- Reference to a non-user address space when using the new EVA instructions

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 5.13 CP0 Register States on Address Exception Error

Register State	Value
BadVAddr	Failing address
$Context_{VPN2}$	UNPREDICTABLE
$EntryHi_{VPN2}$	UNPREDICTABLE
EntryLo0	UNPREDICTABLE
EntryLo1	UNPREDICTABLE

Entry Vector Used:

General exception vector (offset 0x180)

5.7.10 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 5.14 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 $Status_{EXL}$ = 0 at the time of exception;

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

5.7.11 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 5.15 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)

5.7.12 TLB Execute-Inhibit Exception

A *TLB execute-inhibit* exception occurs when there is a execute access from a TLB entry whose XI bit is set. The *TLB execute-inhibit* exception type can only occur if execute-inhibit exceptions are enabled by setting bit 30 (XIE) in the *PageGrain* register.

In addition, the type of exception taken depends on the state of the $PageGrain_{IEC}$ bit. If the XI bit of the entry is set, and the $PageGrain_{IEC}$ bit is set, a TLBXI exception is taken. If the $PageGrain_{IEC}$ bit is cleared, a *TLBL* exception is taken.

Cause Register ExcCode Value:

if $PageGrain._{IEC}$ == 0 TLBL

if $PageGrain._{IEC}$ == 1 TLBXI

Additional State Saved:

Table 5.16 CP0 Register States on TLB Execute-Inhibit Exception

Register State	Value
BadVAddr	Failing address.
Context	If the $Config3._{CTXTC}$ bit is set, then the bits of the *Context* register corresponding to the set bits of the *VirtualIndex* field of the *ContextConfig* register are loaded with the high-order bits of the virtual address that misssed. If the $Config3._{CTXTC}$ bit is clear, then 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)

5.7.13 TLB Read-Inhibit Exception

A *TLB read-inhibit* exception occurs when there is an attempt to read a TLB entry whose RI bit is set. The *TLB read-inhibit* exception type can only occur if read-inhibit exceptions are enabled by setting bit 31 (RIE) in the *PageGrain* register.

In addition, the type of exception taken depends on the state of the $PageGrain_{IEC}$ bit. If the RI bit of the entry is set, and the $PageGrain_{IEC}$ bit is set, a TLBRI exception is taken. If the $PageGrain_{IEC}$ bit is cleared, a *TLBL* exception is taken.

Cause Register ExcCode Value:

if $PageGrain._{IEC}$ == 0 TLBL

if *PageGrain*.$_{IEC}$ == 1 TLBRI

Additional State Saved:

Table 5.17 CP0 Register States on TLB Read-Inhibit Exception

Register State	Value
BadVAddr	Failing address.
Context	If the *Config3*.$_{CTXTC}$ bit is set, then the bits of the *Context* register corresponding to the set bits of the *VirtualIndex* field of the *ContextConfig* register are loaded with the high-order bits of the virtual address that misssed. If the *Config3*.$_{CTXTC}$ bit is clear, then 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)

5.7.14 FTLB Parity Exception

An F*TLB parity* exception occures when a parity error is detected on an FTLB read operation. The error can occur in either the FTLB Tag RAM of the FTLB Data RAM. Note that FTLB parity errors can only occur when the bit 31 (PE) of the CP0 *Error Control* register (*ErrCtl*.$_{PE}$) is set, enabling system-wide parity errors.

When an FTLB parity error occurs, hardware sets bits 31:30 of the CP0 *Cache Error* register (*CacheErr*.$_{EREC}$) to a value of 2’b11 to indicate that the register contains information based on a TLB error. When the EREC field is set to 2’b11, bits 29:28 of the *Cache Error* register (*CacheErr*.$_{ED}$ and *CacheErr*.$_{ET}$) indicate if the error occurred in the FTLB data RAM or the FTLB tag RAM respectively.

Cause Register ExcCode Value:

0x10: FTLBPAR

Additional State Saved:

Table 5.18 CP0 Register States on an FTLB Parity Exception

Register State	Value
CacheErr	Error state. Defined in bits 31:28 of this register.
ErrorEPC	Restart PC

Entry Vector Used:

General exception vector (offset 0x180)

5.7.15 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. Additionally, because the caches on the cores within the proAptiv Multiprocessing System are coherent, cache errors detected on other cores could indicate data corruption for a process on this CPU. An error on another CPU will still cause a Cache Error exception, with the $\mathit{CacheErr}_{EE}$ indicating that the error occurred on another processor.

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

Cause Register ExcCode Value

N/A

Additional State Saved

Table 5.19 CP0 Register States on Cache Error Exception

Register State	Value
CacheErr	Error state
ErrorEPC	Restart PC

Entry Vector Used

Cache error vector (offset 0x100)

5.7.16 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)

5.7.17 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

5.7.18 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)

5.7.19 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)

5.7.20 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)

5.7.21 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 5.20 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)

5.7.22 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)

5.7.23 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)

5.7.24 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 5.21 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)

5.7.25 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)

5.7.26 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)

5.7.27 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

5.7.28 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 5.22 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)

5.8 Exception Handling and Servicing Flowcharts

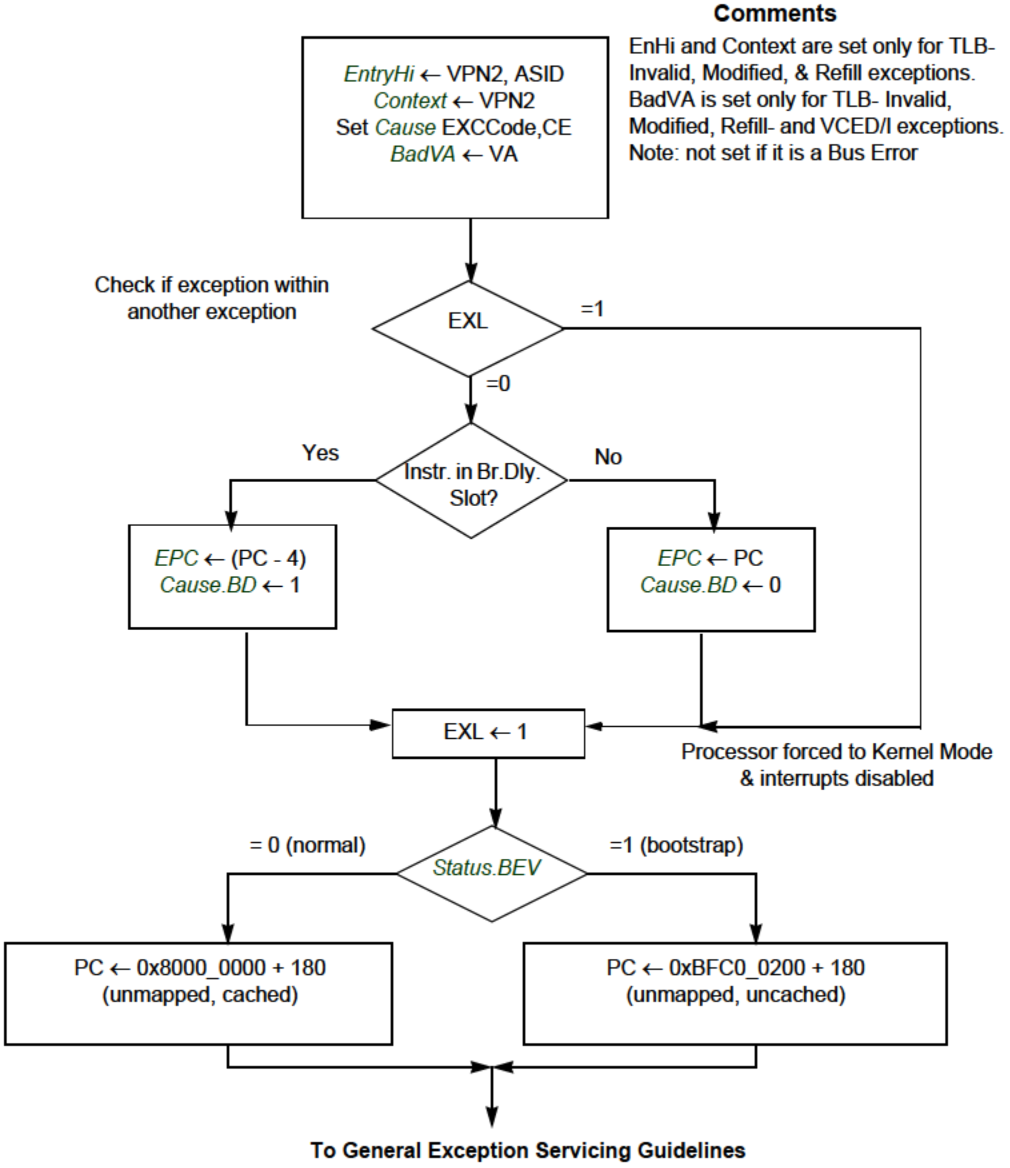
The remainder of this chapter contains flowcharts for the following exceptions and guidelines for their handlers:

- 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 5.2 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 & interrupts 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 5.3 General Exception Servicing Guidelines (SW)

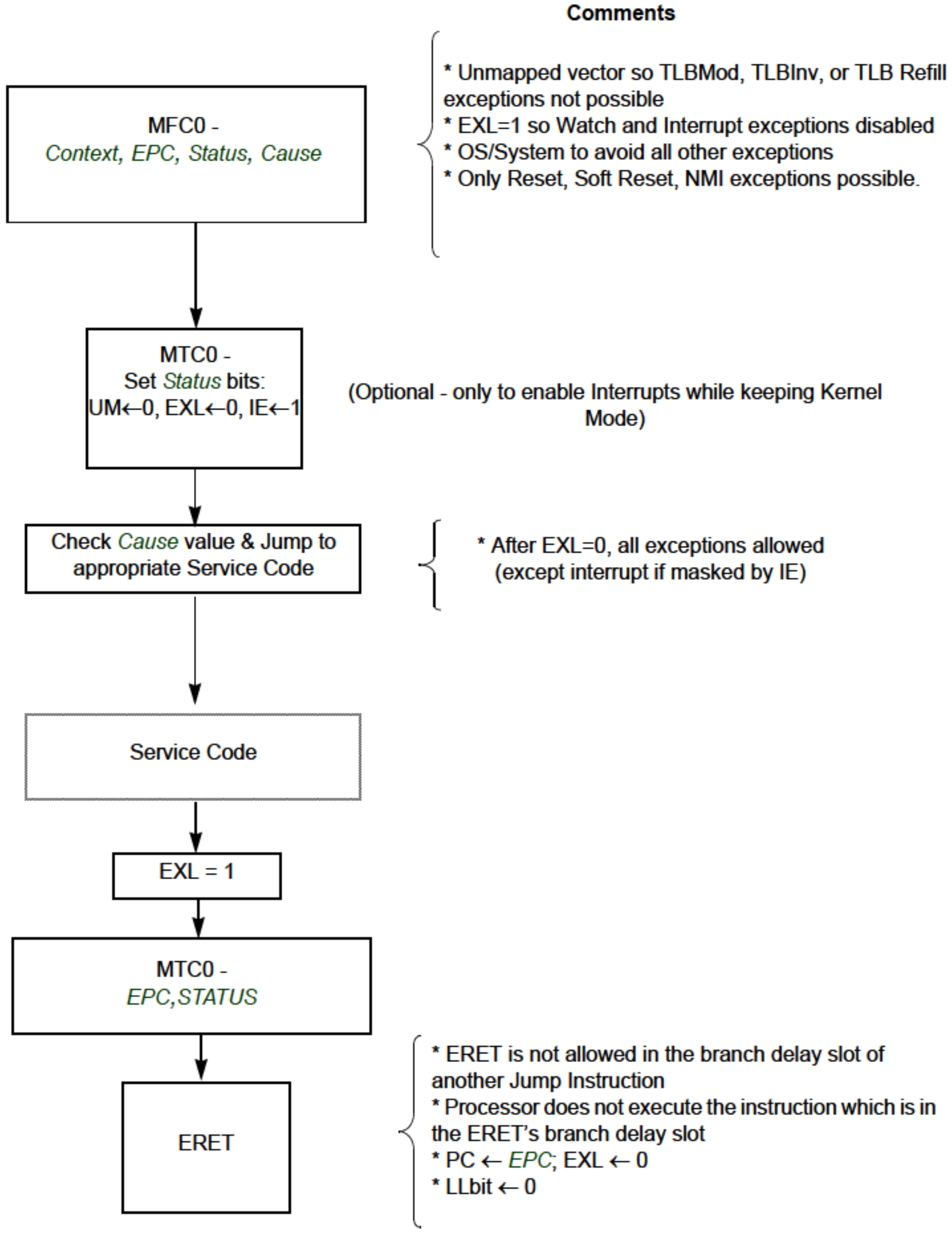

Comments
MFC0 -
Context, EPC, Status, Cause
* Unmapped vector so TLBMod, TLBInv, or TLB Refill exceptions not possible
* EXL=1 so Watch and Interrupt exceptions disabled
* OS/System to avoid all other exceptions
* Only Reset, Soft Reset, NMI exceptions possible.
MTC0 -
Set Status bits:
UM←0, EXL←0, IE←1
(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 5.4 TLB Miss Exception Handler (HW)

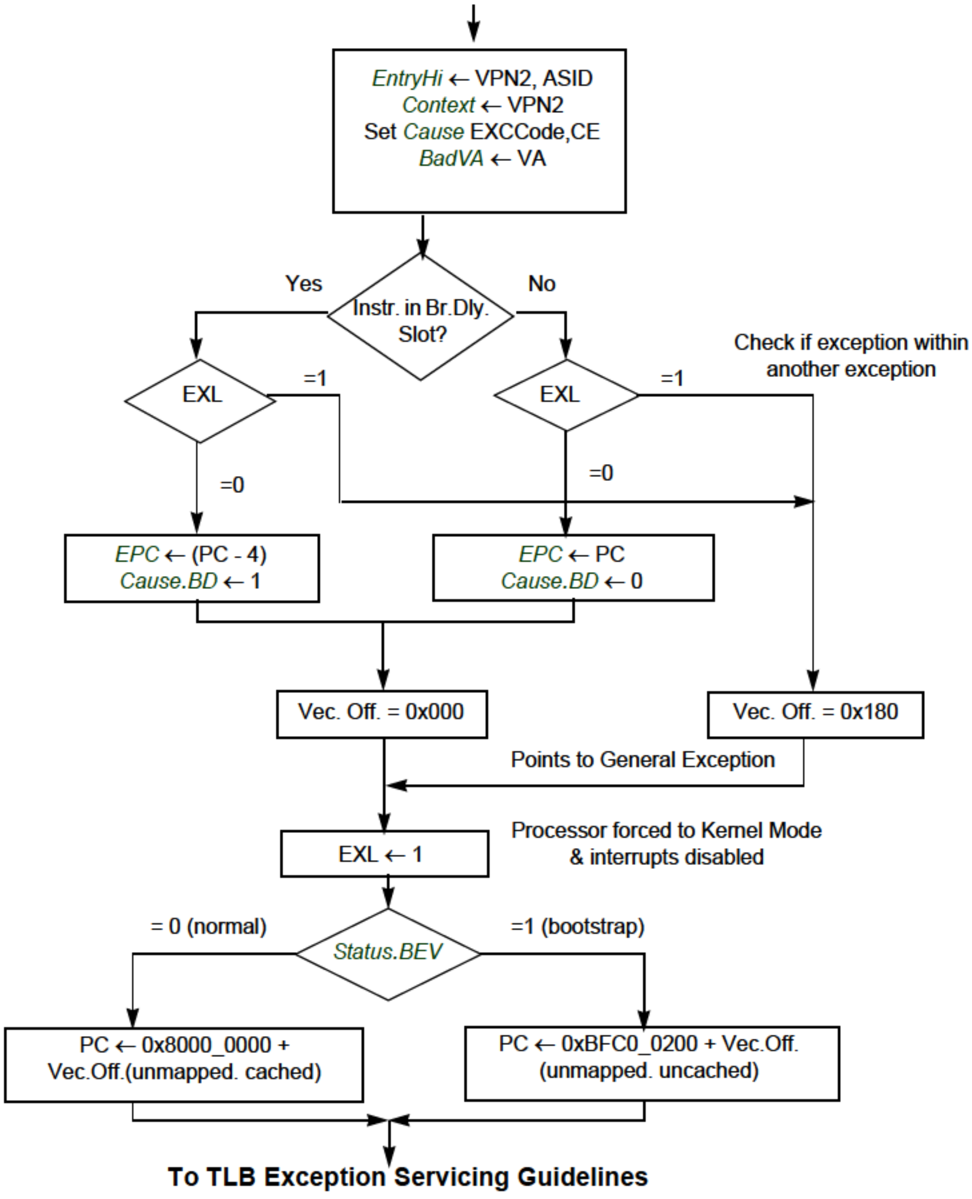
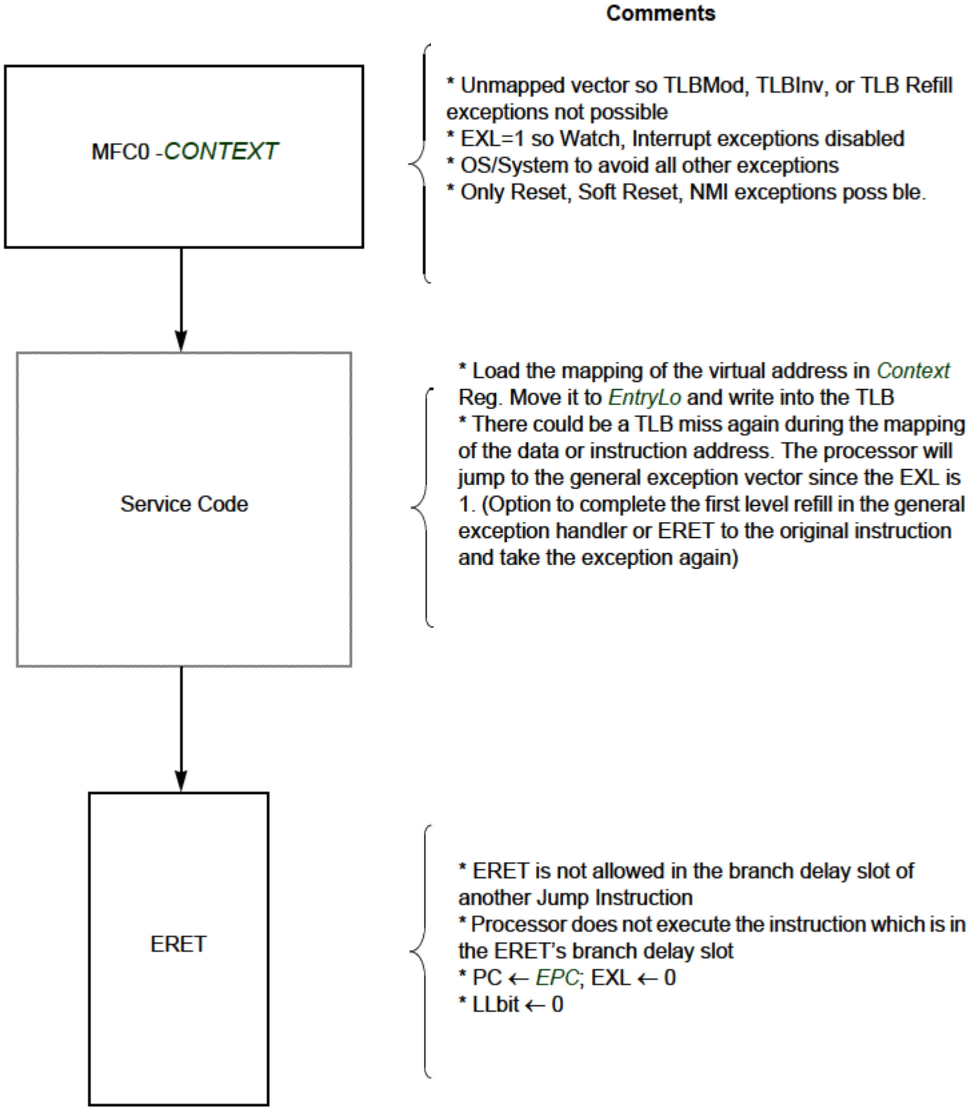

EntryHi ← VPN2, ASID
Context ← VPN2
Set Cause EXCCode,CE
BadVA ← VA
Yes
Instr. in Br.Dly. Slot?
No
EXL
=1
EXL
=1
Check if exception within another exception
=0
=0
EPC ← (PC - 4)
Cause.BD ← 1
EPC ← PC
Cause.BD ← 0
Vec. Off. = 0x000
Vec. Off. = 0x180
Points to General Exception
EXL ← 1
Processor forced 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 5.5 TLB Exception Servicing Guidelines (SW)



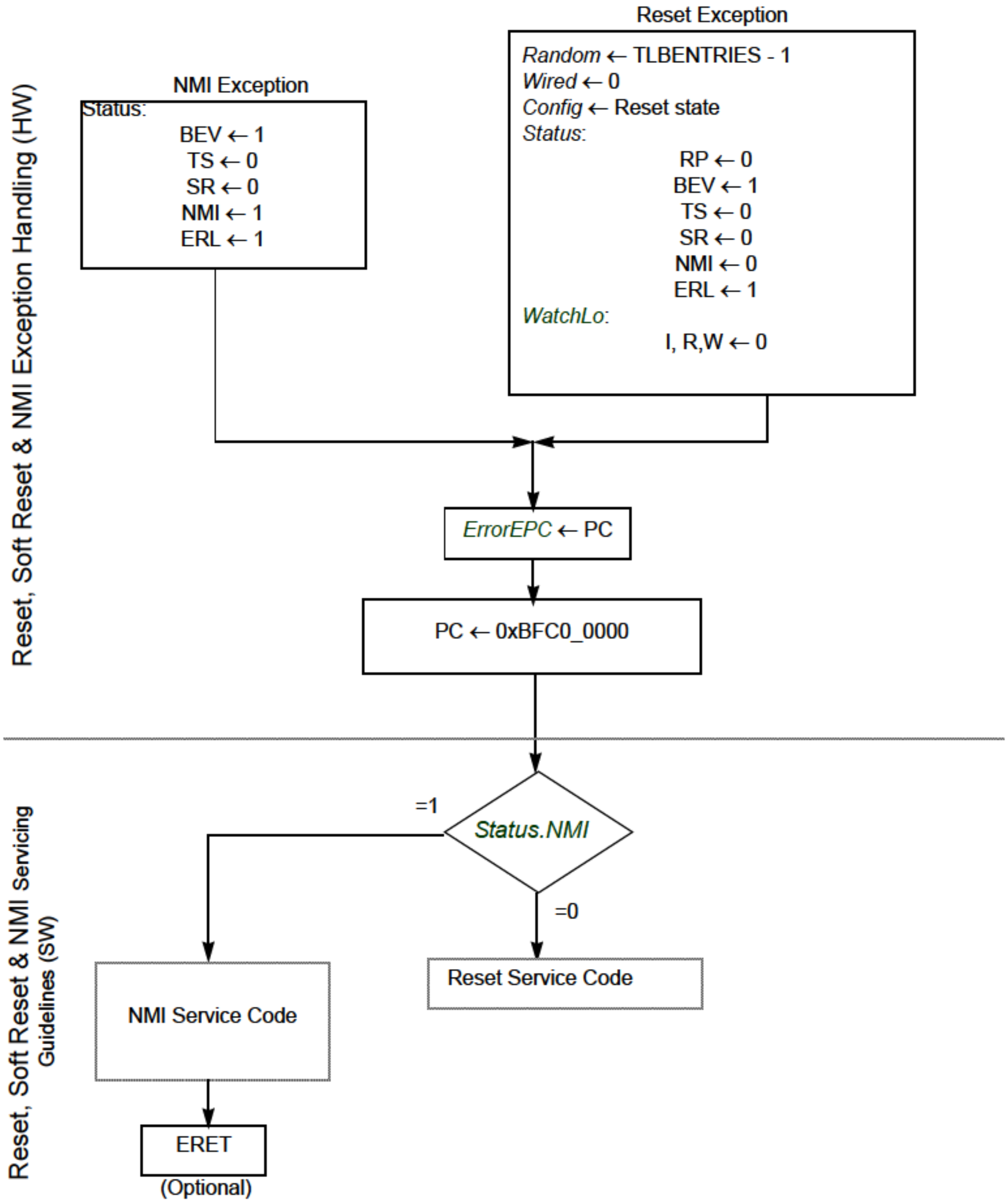
Comments

* 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 poss ble.

* 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 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 5.6 Reset and NMI Exception Handling and Servicing Guidelines



5.9 Interrupts

Release 3 of the MIPS32 architecture, implemented by the proAptiv Multiprocessing System CPU, includes 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 proAptiv Multiprocessing System CPU. These counters can be configured to count various events within the CPU. 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 CPU that can be brought back into the CPU's interrupt pins in a system-dependent manner.

The Fast Debug Channel feature in EJTAG provides a low overhead means for sending data between CPU software and the EJTAG probe. It includes a pair of FIFOs for transmit and receive data. Software can define FIFO thresholds for generating an interrupt. The fast debug channel interrupt is also routed similarly to the timer and performance counter interrupts. The interrupt status is made available on an output pin and can be brought back into the CPU's interrupt pins.

5.9.1 Interrupt Modes

The proAptiv Multiprocessing System CPU includes support for three interrupt modes, as defined by Release 3 of the Architecture:

- Interrupt Compatibility mode, in which the behavior of the proAptiv Multiprocessing System is identical to the behavior of an implementation of Release 1 of the Architecture.

- Vectored Interrupt (VI) mode adds the ability to prioritize and vector interrupts to a handler dedicated to that interrupt. 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 proAptiv Multiprocessing System CPU, 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 proAptiv Multiprocessing System CPU, 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 proAptiv Multiprocessing System processor defaults to Compatibility mode, which is fully compatible with all implementations of Release 1 of the Architecture.

Table 5.23 shows the current interrupt mode of the processor as a function of the Coprocessor 0 register fields that can affect the mode.

Table 5.23 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

Table 5.23 Interrupt Modes*(continued)*

StatusBEV	*CauseIV*	*IntCtlVS*	*Config3VINT*	*Config3VEIC*	**Interrupt Mode**
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					

5.9.1.1 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
 *  - 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 */
   sw     k0, EPCSave         /* Save in memory */
   mfc0   k0, C0_Status       /* Get Status value */
   sw     k0, StatusSave      /* Save in memory */
   li     k1, ~IMbitsToClear  /* Get IM bits to clear for this interrupt */
                              /*   this must include at least the IM bit */
                              /*   for the current interrupt, and may include */
                              /*   others */
   and    k0, k0, k1             /* Clear bits in copy of Status */
   ins    k0, zero, S_StatusEXL, (W_StatusKSU+W_StatusERL+W_StatusEXL)
                                 /* Clear KSU, ERL, EXL bits in k0 */
   mtc0   k0, C0_Status          /* Modify mask, switch to kernel mode, */
                                 /*   re-enable interrupts */

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

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

   di                         /* Disable interrupts - may not be required */
```

```
lw     k0, StatusSave      /* Get saved Status (including EXL set) */
lw     k1, EPCSave         /*   and EPC */
mtc0   k0, C0_Status       /* Restore the original value */
mtc0   k1, C0_EPC          /*  and EPC */
/* Restore GPRs and software state */
eret                       /* Dismiss the interrupt */
```

5.9.1.2 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. VI mode is in effect when all the following conditions are true:

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

In VI interrupt mode, the six hardware interrupts are interpreted as individual hardware interrupt requests. The timer, performance counter, and fast debug channel interrupts are combined in a system-dependent way (external to the CPU) with the hardware interrupts (the interrupt with which they are combined is indicated by the $IntCtl_{IPTI/IPCI/IPFDCI}$ 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 5.24.

Table 5.24 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

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.

A nested interrupt is similar to that shown for compatibility mode. Such a routine might look as follows:

```
NestedException:
/*
* Nested exceptions typically require saving the EPC and Status registers,
* 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.
*/
   mfc0   k0, C0_EPC          /* Get restart address */
   sw     k0, EPCSave         /* Save in memory */
   mfc0   k0, C0_Status       /* Get Status value */
   sw     k0, StatusSave      /* Save in memory */
   li     k1, ~IMbitsToClear  /* Get IM bits to clear for this interrupt */
                              /*   this must include at least the IM bit */
                              /*   for the current interrupt, and may include */
                              /*   others */
   and    k0, k0, k1              /* Clear bits in copy of Status */
   ins    k0, zero, S_StatusEXL, (W_StatusKSU+W_StatusERL+W_StatusEXL)
                                  /* Clear KSU, ERL, EXL bits in k0 */
   mtc0   k0, C0_Status           /* Modify mask, switch to kernel mode, */
                                  /*   re-enable interrupts */

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

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

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

5.9.1.3 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, fast debug channel, 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

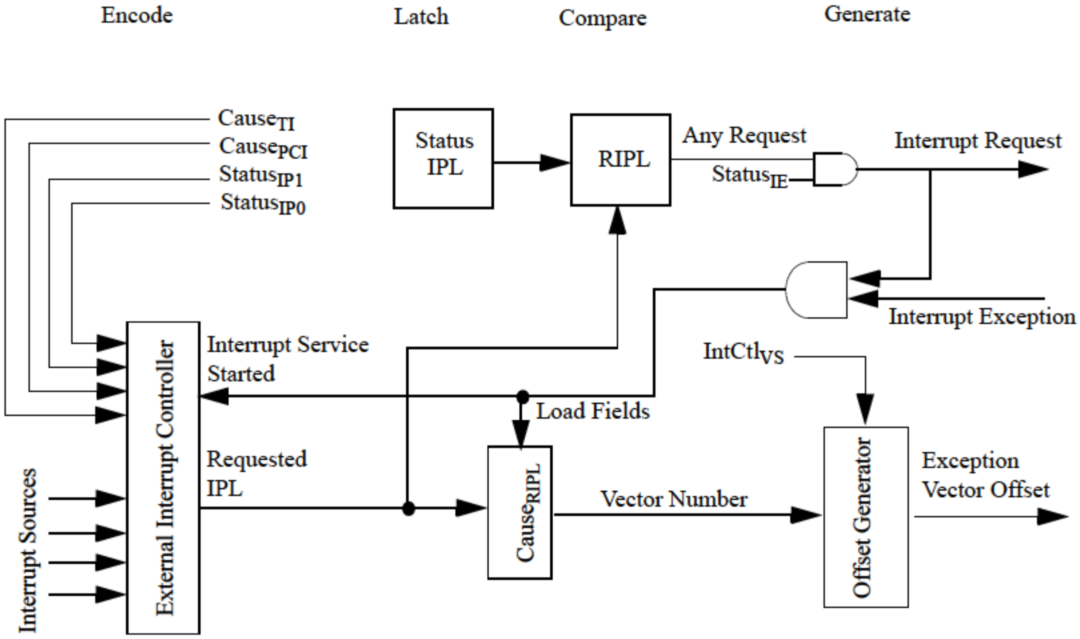
In EIC mode, the processor sends the state of the software interrupt requests ($Cause_{IP1..IP0}$) and the timer, performance counter, and fast debug channel interrupt requests ($Cause_{TI/PCI/FDCI}$) to the external interrupt controller, which prioritizes these interrupts in a system-dependent way with 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.

The operation of EIC interrupt mode is shown in Figure 5.7.

Figure 5.7 Interrupt Generation for External Interrupt Controller Interrupt Mode



A typical software handler for EIC mode bypasses the entire sequence of code following the `IV exception` label shown for the compatibility-mode handler above. Instead, the hardware performs the prioritization, dispatching directly to the interrupt processing routine.

A nested interrupt is similar to that shown for compatibility mod. 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 and Status registers,
* 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.
*/

   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 */
   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 */

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

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

5.9.2 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 5.25 shows the exception vector offset for a representative subset of the vector numbers and values of the *IntCtl*$_{VS}$ field.

Table 5.25 Exception Vector Offsets for Vectored Interrupts

	Value of IntCtl$_{VS}$ Field				
Vector Number	**5’b00001**	**5’b00010**	**5’b00100**	**5’b01000**	**5’b10000**
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

Table 5.25 Exception Vector Offsets for Vectored Interrupts *(continued)*

Vector Number	Value of IntCtl_{VS} Field				
	5’b00001	**5’b00010**	**5’b00100**	**5’b01000**	**5’b10000**
• • •					
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 × ($\text{IntCtl}_{VS}$ ∥ 0b00000))
```

5.9.3 Global Interrupt Controller

The Global Interrupt Controller (GIC) handles the routing and masking of local interrupts, such as the timer, performance counter, fast debug channel interrupts, inter-processor interrupts, and external interrupts. This block can be configured to support various numbers of external interrupts and to support any of the CPU interrupt modes.

An interactive GUI is available to simplify the setup of desired event-routing through the GIC. The tool outputs a C-language function covering all required programming registers of the GIC.

Chapter 6

Programming Concepts

This chapter describes some programming concepts that can be followed when programming in either User mode or Kernel mode.

6.1 Ordering and Synchronization

6.1.1 Consistency Model

The proAptiv Multiprocessing System uses weak memory ordering, that is, cacheable loads and stores on a processor can be executed out of program order (for example, for hit-under-miss). Software must include SYNC instructions to enforce ordering in the cases where it is required.

6.1.2 LL/SC

The Load Linked (LL) and Store Conditional (SC) instructions provide a mechanism that ensures atomic access to a memory location.

An LL instruction reads a memory location and sets an internal state bit called the LL bit. The address read by the LL instruction is stored in the *LLAddr* register. The LL bit can be cleared because of actions on the processor, such as an ERET instruction. If the LL bit is cleared before the SC completes, the SC fails and does not update memory.

On the proAptiv Multiprocessing System, the value in *LLAddr* is also checked on interventions. If another CPU requests write access to the cache line, the LL bit will be cleared. LL instructions always request the line in a Shared state, so that an LL itself does not clear the LL bit in another CPU. If the line is installed as Shared, when the SC is executed, it must make a CohUpgrade request to obtain write access to the line. If multiple cores are trying to access the same location, there can be a race, and the first Upgrade request to be serialized in the CM will win. This will cause the other SCs to fail, and the other cores must retry the sequence from the LL.

These actions allow the memory location to appear as though it were atomically updated—the SC will not write the location unless the update will appear atomic.

6.1.3 Memory Barriers

The SYNC instruction is used to enforce the ordering of loads and stores. Because the core processes instructions in order and generates memory requests in order, these SYNCs can complete with much less delay than the traditional heavyweight SYNC. All of the lightweight stypes (0x4, 0x10-0x13) are treated identically by the CPU as follows:

1. The LSU forces any pending evictions to complete their cache reads and send the writes to the BIU.

2. The BIU flushes the write-back buffer.

3. The BIU indicates that it is complete and allows the LSU to resume processing instructions.

4. No external SYNC request is generated.

Additionally, the CPU supports two implementation-specific `stype` values as well as the standard `stype` 0x0. These are used to explicitly set the ‘level’, which controls how far into the system SYNCs are propagated:

If Coherence is enabled:

- `stype` 0x2: A coherent SYNC is sent to the CM. The CM responds when all older coherent requests have completed their interventions.

- `stype` 0x3: A Coherent SYNC transaction is sent to the CM. If *SI_CM_SyncTxEn* is 0 or *CM_SYNC_TX_DISABLE* is 1then the CM responds when all previous coherent requests have completed their interventions and all previus requests have been accepted on the L2/Memory interface. If *SI_CM_SyncTxEn* is 1 and *CM_SYNC_TX_DISABLE* is 0, then the CM waits until all previous coherent requests have been completed and before issuing a Legacy SYNC transaction to L2/memory (behind all previous coherent and non-coherent requests from this CPU) to enforce ordering throughout the system. In this case, the CM responds when it has received a response from L2/Memory.

- `stype` 0x0: The level that normal SYNCs use can be controlled by the SYNCCTL bit in the *Global CM2 Control* register located at offset address 0x0010. Refer to the *Global CM2 Control* register in the CM2 Registers chapter for more information.

All other stypes are reserved and currently default to type 0x0.

If Coherence is disabled:

- `stype` 0x0, 0x2, 0x3: A legacy SYNC transaction is issued to the CM. If *SI_CM_SyncTxEn* is 0 or *CM_SYNC_TX_DISABLE* is 1 then CM responds when all previous requests have been accepted on the L2/Memory interface. If *SI_CM_SyncTxEn* is 1 and *CM_SYNC_TX_DISABLE* is 0 then the CM issues a Legacy SYNC transaction to L2/Memory (behind all previous non-coherent requests from this CPU) to enforce ordering throughout the system. In this case, the CM responds when it has received a response from L2/Memory.

- All other stypes are reserved and currently default to type 0x0.

Table 6.1 Supported SYNC stypes

Coherence Enabled?	stype	Behavior
Yes	0x0	Can be configured as level0 or level1, as defined below.
	0x2	Level0 - SYNC transaction is sent to the Coherence Manager and waits for all previous coherent transactions to finish their intervention stage.
	0x3	Level1 - After completing level0 steps, memory accesses are also completed. Depending on the setting of the *SyncTxEn* and *CM_SYNC_TX_DISABLE* bits, an external SYNC transaction may also be generated to flush external devices.
No	0x0, 0x2, 0x3	Core issues Legacy SYNC
-	0x4, 0x10-0x13	Lightweight SYNC - handled entirely within the CPU, completes evictions and flushes WBB.
	All others	Reserved. Default to type 0x0.

6.1.4 CACHE and SYNCI Instructions

Coordinating software maintenance of the caches across multiple cores can be rather challenging and involve a lot of overhead. To simplify the task of maintaining cache Coherence via software, the proAptiv Multiprocessing System includes hardware support for the globalization of a number of cache maintenance operations. When a cache operation is globalized, it becomes a coherent request and is sent through the Coherence Manager to be performed on all of the cores. The decision of whether or not to globalize an operation is based on whether or not the target address for the operation is coherent. The operations that are globalized are Hit-type L1 CACHE instructions and SYNCI instructions.

Several special cases deserve additional consideration and are discussed below.

6.1.4.1 Cache Line Locking

Locking lines into a cache is somewhat counter to the idea of coherence. If a line is locked into a particular cache, it is expected that any processes utilizing that data will be locked to that processor and coherence is not needed. Based on this usage model, locking coherent lines into the cache is not recommended. If it is done, the cores will use the following rules:

- SYNCI instructions are user-mode instructions. Because locking is a kernel-mode feature (it requires the CACHE instruction), SYNCI is not allowed to unlock cache lines. This applies to both local and globalized SYNCI instructions.

- Locking overrides coherence. Intervention requests from other cores and I/O devices that match on a locked line will be treated as misses.

- Self-intervention requests for globalized CACHE instructions will be allowed to affect a locked line. This is done primarily for handling lock and unlock requests for kseg0 addresses when kseg0 is being treated coherently.

6.1.4.2 Index Type and Optimized Routines

Index-type CACHE instructions are not globalized. Because they refer to a specific cache location, it does not make sense to apply them to other caches, particularly if the cache configurations are not homogeneous.

One case where software may attempt to use index-type CACHE instructions is an optimization used when flushing large blocks of memory. If the region to be flushed is larger than the size of the cache, flushing the entire cache could be faster than walking through the region and flushing each cache line individually (though the flushing of unrelated cache lines may mitigate the benefit of this optimization). Because indexed operations are not globalized, this sequence only flushes the local cache. If flushing of the remote caches is also required, the code sequence must also run on the remote cores. It is probably better to disable this software optimization and make use of the efficiency of the globalized hit-type CACHE instructions.

6.1.4.3 Completion

Globalizing a cache operation changes its timing, compared to a local operation. The external request must be made, serialized in the Coherence Manager, and then sent to the cores on the intervention port. This is not a blocking action, and subsequent instructions on the requesting CPU will continue to execute. In order to guarantee that the operation has been completed, a SYNC instruction must be executed prior to any instruction that requires the updated state. This can be a single SYNC after a series of cache operations. This SYNC should also be used on non-coherent cores in the Cluster to ensure maximum compatibility moving forward.

6.1.4.4 L2 CACHE Instructions

It is important to note that L2 CACHE instructions only impact the L2 cache and do not affect the L1 data or instruction caches.

6.1.5 PREF Instructions

Prefetch instructions are also impacted by coherence. The different types of PREF react differently, as described below.

- *Normal*: Load/store(_*) type hint values will cause the appropriate type of request to be issued when a coherent CCA is used—a store hint will request Exclusive ownership, and a load hint will request either Shared or Exclusive, depending on the CCA. However, a store-type PREF that hits on a Shared line will not make an Upgrade request.

- *Writeback_invalidate* (also called *nudge*): This operation behaves the same for both coherent and non-coherent CCAs and in both cases will only force a writeback (if needed) from the local cache.

- *Prepare for Store*: This operation is intended to avoid the memory read when software is going to be writing an entire cache line. When a coherent address is used, an Invalidate request is generated to clear the lines of any other data caches in the system and acquire Exclusive ownership on the local processor.
  Note: This operation changes the state of memory, and the data values are unpredictable until the series of stores has completed. If other software (running on other processors) accesses the line before the series of stores has completed, this unpredictable intermediate state can be observed.

6.2 User Mode Programming

This section contains the following programming concepts relative to user mode programming:

- Section 6.2.1, "User Mode Accessible CP0 Registers"
- Section 6.2.2, "Prefetching Data Using the pref and prefx Instructions": how it works.
- Section 6.2.3, "Using “SYNCI” When Writing Instructions": writing instructions without needing to use privileged cache management instructions.
- Section 6.2.4, "Integer Multiply and Divide": multiply, multiply/accumulate and divide timings.
- Section 6.2.5, "Tuning Software for the Pipeline": for determined programmers, and for compiler writers. It includes information about the timing of the DSP ASE instructions.
- Section 6.2.6, "Branch Misprediction Delays": the floating-point unit often runs at half speed, and some of its interactions (particularly about potential exceptions) are complicated. This section offers some guidance about the timing issues you’ll encounter.
- Section 6.2.7, "Load Delayed by (Unrelated) Recent Store"
- Section 6.2.8, "Minimum Load-miss Penalty"
- Section 6.2.9, "Data Dependency Delays"
- Section 6.2.10, "Advice on Tuning Instruction Sequences (particularly DSP)"
- Section 6.2.11, "Multiply/Divide Unit and Timings"

6.2.1 User Mode Accessible CP0 Registers

In the proAptiv Multiprocessing System architecure, privileged code executed in kernel mode can access any CP0 register. Conversely, unprivileged user mode code does not have access to any CP0 register. However, there are instances where unprivileged user mode programs may need information from some of the CPU registers, normally to share information which is worth making accessible to programs without the overhead of a system call. There are two ways to allow user mode programs access to CP0 registers.

- Set bit 28 (CU_0) of the CP0 Status register located at Register 12, Select 0. Setting this bit allows user mode programs access to all fields of all CP0 registers. See Section 6.2.1.1 “Setting the CU0 Bit of the Status Register” for more information.
- Set selected bits in the CP0 Hardware Enable (HWREna) register located at Register 7, Select 0. See Section 6.2.1.2 “Programming the HWREna Register” for more information.

6.2.1.1 Setting the CU0 Bit of the Status Register

Setting the CU_0 bit of the CP0 Status register allows user mode programs to have unrestricted access to the CP0 register set. This permission is granted by the kernel, but this is typically never done. As described in the following subsection, the HWREna register can be used by user mode programs to extract selected information from the CP0 register set.

6.2.1.2 Programming the HWREna Register

To facilitate non-privileged user mode accesses to selected CP0 registers, the proAptiv Multiprocessing System core allows selected information from the CP0 register set to be accessed via the **`rdhwr`** instruction. The operating system can control access to each register individually, through a bitmask in the CP0 register *HWREna* - (set bit 0 to enable register 0 etc) register. *HWREna* is cleared to all-zeroes on reset, so software has to explicitly enable user access.

Figure 6.1 shows the bit assignments for the *HWREna* register. Note that the entire register is cleared to zero on reset, so that no hardware register is accessible without positive OS clearance.

Figure 6.1 Fields in the HWREna Register

31 30	29	28 4	3	2	1	0
Impl	UL	0	CCRes	CC	SYNCI_Step	CPUNum
0	0		0	0	0	0

The *HWREna* register contains five bit-fields that allow access to the following information:

- $HWREna_{CPUNum}$ (bit 0): Software sets this bit to enable hardware fetch the number of the CPU on which the program is currently running. Upon execution of a **`rdhwr 0`** instruction, hardware fetches this information directly from the CP0 $EBase_{CPUNum}$ field and places it into the GPR register designated by the *rt* field of the **`rdhwr 0`** instruction. The *CPUNum* field in the *EBase* register is initially set by hardware based on the setting of external pins at reset.

- $HWREna_{SYNCI\_Step}$ (bit 1): When this bit is set, execution of a **`rdhwr 1`** instruction returns the effective size of an L1 cache line. This information is important to user programs because they can now do things to the caches using the **`synci`** instruction to make recently written instructions visible for execution. The information returned indicates the “step size” — the address increment between successive **`synci`** instructions required to cover all the instructions in a range. In the proAptiv core, the line size is always 32-bytes.

- $HWREna_{CC}$ (bit 2): When this bit is set, the proAptiv Multiprocessing System hardware allows user mode read-only access to the CP0 *Count* register, for high-resolution counting. Execution of a **`rdhwr 2`** instruction returns the current value in the *Count* register and places it into the GPR register designated by the *rt* field of the **`rdhwr 2`** instruction.

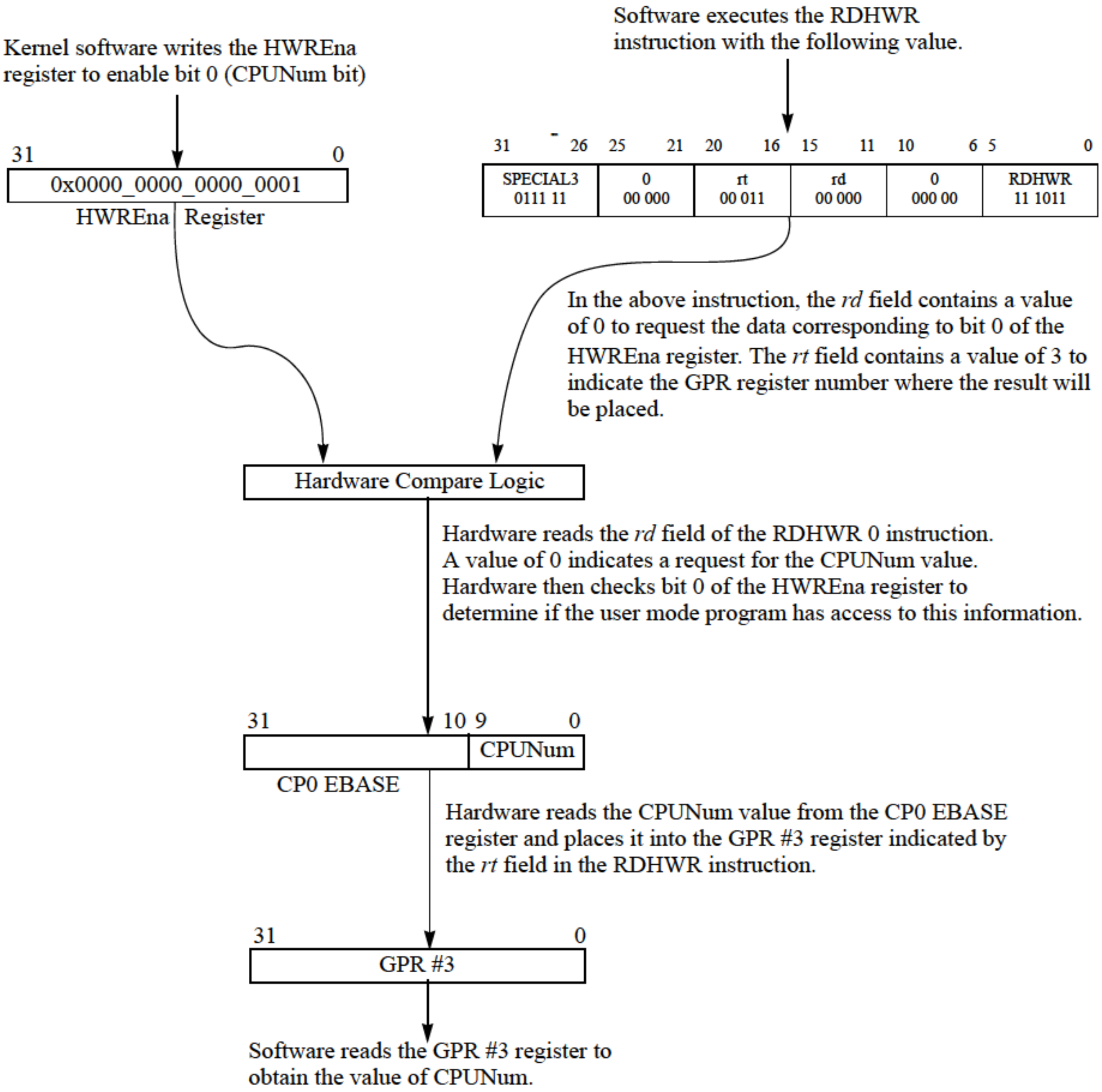
- $HWREna_{CCRes}$ (bit 3): When this bit is set, which tells you how fast *Count* counts. It’s a divider from the pipeline clock. If the **`rdhwr 3`** instruction reads a value of “2”, then the *Count* register increments every 2 cycles, at half the pipeline clock rate. In the proAptiv Multiprocessing System core, the CCRes value is always 2 to indicate that the CC register increments every second core cycle.

- $HWREna_{UL}$ (bit 29): When this bit is set, hardware allows user mode read-only access to the CP0 *UserLocal* register. The execution of a **`rdhwr 29`** instruction provides a core identifier to user mode programs.

6.2.1.3 Programming Example

The following example shows the flow of information through the proAptiv core when user code accesses the CPU number from the CP0 register set.

Figure 6.2 Obtaining the CPU Number in User Mode



Kernel software writes the HWREna register to enable bit 0 (CPUNum bit)

Software executes the RDHWR instruction with the following value.

In the above instruction, the *rd* field contains a value of 0 to request the data corresponding to bit 0 of the HWREna register. The *rt* field contains a value of 3 to indicate the GPR register number where the result will be placed.

Hardware reads the *rd* field of the RDHWR 0 instruction. A value of 0 indicates a request for the CPUNum value. Hardware then checks bit 0 of the HWREna register to determine if the user mode program has access to this information.

Hardware reads the CPUNum value from the CP0 EBASE register and places it into the GPR #3 register indicated by the *rt* field in the RDHWR instruction.

Software reads the GPR #3 register to obtain the value of CPUNum.

6.2.2 Prefetching Data Using the pref and prefx Instructions

MIPS32 CPUs are being increasingly used for computations which feature loops accessing large arrays, and the runtime is often dominated by cache misses.

These are excellent candidates for using the **pref/prefx** instructions, which gets data into the cache without affecting the CPUs other state. In a well-optimized loop with prefetch, data for the next iteration can be fetched into the cache in parallel with computation for the last iteration. The **pref/prefx** instructions is logically a no-op[1]and should have *no software-visible effect* other than to make things go faster.

The **pref/prefx** instructions come with various possible “hints” which allow the program to express its best guess about the likely fate of the cache line. The “load” and “store” variants of the hints for this instruction perform differently. Specifically, when dealing with coherent addresses, the “store” types will request Exclusive ownership so that an eventual store does not need to make an upgrade request before modifying the line.

The proAptiv Multiprocessing System core acts on hints as summarized in Table 6.2.

Table 6.2 Hints for pref and prefx Instructions

Hint		***Action Taken by the Core***	***Usage***
Number	***Name***		
0	load	Read the cache line into the D-cache if not present.	When you expect to read the data soon. Use “store” hint if you also expect to modify it.
1	store		
4	load_streamed	Fetch data, but always use cache way zero - so a large sequence of “streamed” prefetches will only ever use a quarter of the cache.	For data you expect to process sequentially, and can afford to discard from the cache once processed
5	store_streamed		
6	load_retained	Fetch data, but never use cache way zero. That means if you do a mixture of “streamed” and “retained” operations, they will not displace each other from the cache.	For data you expect to use more than once, and which may be subject to competition from “streamed” data.
7	store_retained		
25	writeback_invalidate/nudge	If the line is in the cache, invalidate it (writing it back first if it was dirty). Otherwise do nothing. However (with the proAptiv Multiprocessing System CPU only): if this line is in a region marked for “uncached accelerated write” behavior, then write-back this line.	When you know you’ve finished with the data, and want to make sure it loses in any future competition for cache resources.
30	PrepareForStore	If the line is not in the cache, create a cache line - but instead of reading it from memory, fill it with zeroes and mark it as “dirty”. If the line is already in the cache do nothing - *this operation cannot be relied upon to zero the line*.	When you know you will overwrite the whole line, so reading the old data from memory is unnecessary. A recycled line is zero-filled only because its former contents could have belonged to a sensitive application - allowing them to be visible to the new owner would be a security breach.
31	PrepareForStoreNZ	As type 30 above, except that the line is not filled with zeroes.	Yields the highest possible performance when you’re going to overwrite the whole line. However, this is at the cost of a security leak: a user-mode application which uses this prefetch can (somewhat randomly) obtain a view of kernel or other-process memory data it should not be able to see. An OS can make this instruction safe (same as **pref 30** above) by keeping *Config7[FPFS]* zero — see XREF Figure B.3 and notes.

6.2.3 Using “SYNCI” When Writing Instructions

The **synci** instruction (introduced with Revision 2 of the MIPS32 architecture specification) ensures that instructions written by a program (necessarily through the D-cache) get written back from the D-cache and corresponding I-

[1] Note that the **pref** instruction with the “PrepareForStore” hint can zero out some data which wasn’t previously zero.

cache locations invalidated, so that any future execution at the address will reliably execute the new instructions. **synci** takes an address argument, and it takes effect on a whole enclosing cache-line sized piece of memory. User-level programs can discover the cache line size because it's available in a "hardware registers" accessed by the **rdhwr** instruction, as described in Section 6.2.1, "User Mode Accessible CP0 Registers" above.

Since **synci** is modifying the program's own instruction stream, it's inherently an "instruction hazard". Therefore, once the last instruction has been written and the last **synci** has been issues, programmer's should use a **jr.hb** or equivalent to call the new instructions.

The following code example shows how the **synci** can be used.

```
/*
* This routine makes changes to the instruction stream effective to the hardware. It should be called after the instruc-
* tion stream is written. On return, the new instructions are effective.
*
* Inputs:
* a0 = Start address of new instruction stream
* a1 = Size, in bytes, of new instruction stream
*/

        beq     a1, zero, 20f               /* If size==0, */
        nop                                 /* branch around */
        addu    a1, a0, a1                  /* Calculate end address + 1 */
        rdhwr   v0, HW_SYNCI_Step           /* Get step size for SYNCI from new */
                                            /* Release 2 instruction */
        beq     v0, zero, 20f               /* If no caches require synchronization, */
        nop                                 /* branch around */
10:     synci   0(a0)                               /* Synchronize all caches around address */
        addu    a0, a0, v0                  /* Add step size in delay slot */
        sltu    v1, a0, a1                  /* Compare current with end address */
        bne     v1, zero, 10b               /* Branch if more to do */
        nop                                 /* branch around */
        sync                                /* Clear memory hazards */
20:     jr hb   ra                          /* Return, clearing instruction hazards */
        nop
```

6.2.4 Integer Multiply and Divide

As is traditional with MIPS CPUs, the integer multiplier is a semi-detached unit with its own pipeline. All MIPS32 CPUs implement:

- **mult**/**multu**: a 32×32 multiply of two GPRs (signed and unsigned versions) with a 64-bit result delivered in the multiply unit's pseudo-registers *hi* and *lo* (readable only using the special instructions **mfhi** and **mflo**, which are interlocked and stall until the result is available).

- **madd**, **maddu**, **msub**, **msubu**: multiply/accumulate instructions collecting their result in *hi*/*lo*.

- **mul**/**mulu**: simple 3-operand multiply as a single instruction.

- **div**/**divu**: divide - the quotient goes into *lo* and the remainder into *hi*.

No multiply/divide operation ever produces an exception - even divide-by-zero is silent - so compilers typically insert explicit check code where it's required.

The proAptiv Multiprocessing System core multiplier is high performance and pipelined. *Multiply/accumulate* instructions can run at a rate of 1 per clock, but a 32×32 3-operand multiply takes six clocks longer than a simple ALU operation. Divides use a bit-per-clock algorithm, which is short-cut for smaller dividends. Multiply/divide instructions are generally slow enough that it is difficult to arrange programs so that their results will be ready when needed.

6.2.5 Tuning Software for the Pipeline

This section is addressed to low-level programmers who are tuning software by hand and to those working on efficient compilers or code translators.

6.2.5.1 Cache Delays

In a typical CPU implementation a cache miss which has to be refilled from DRAM memory will be delayed by a some period of time, perhaps long enough to run 50-200 instructions. In addition, a miss or uncached read may easily be several times slower.

Because these delays are so large, there is little that can be done when a cache miss except wait for it to be resolved. To mitigate cache misses, the proAptiv Multiprocessing System core supports non-blocking loads. Therefore, if the programmer can provide separation in the code stream between a load instruction producer and its consumer, the memory delay will not begin until the consuming instruction is executed.

Compilers and programmers may find it difficult to move fragments of an algorithm around like this, so the proAptiv Multiprocessing System architecture also provides *prefetch* instructions, such as **`pref`** an **`prefx`**, which fetch designated data into the D-cache, but do nothing else. Any loop which walks predictably through a large array is a candidate for prefetch instructions, which are conveniently placed within one iteration to prefetch data for the next.

The **`pref PrepareForStore`** prefetch saves a cache refill read, for cache lines which are intended to be overwritten in their entirety. Read more about prefetch in Section 6.2.2, "Prefetching Data Using the pref and prefx Instructions" above.

Tuning Data-Intensive Common Functions

Bulk operations like **`bcopy()`** and **`bzero()`** can benefit from CPU-specific tuning. To get excellent performance for in-cache data, it’s only necessary to reorganize the software enough to cover the address-to-store and load-to-use delays. To get the loop to achieve the best performance when the cache misses, **`pref`** instuctions can be used.

6.2.6 Branch Misprediction Delays

In a pipelined design with multiple stages like proAptiv Multiprocessing System, branch delays would be lengthy if software waited until the branch was executed before fetching any more instructions. In general, the amount of delay depends on the type of branch. For example, a conditional branch which closes a tight loop will almost always be predicted correctly after the first time around.

However, too many branches in too short a period of time can overwhelm the ability of the instruction fetch logic to keep ahead with its predictions, even if the predictions are almost always right. Three empty cycles occur between the delivery of the branch delay slot instruction and the first instruction(s) from the branch target location. To mitigate the effects of ‘branchy’ code, the code can be replaced by conditional moves or tight loops “unrolled” to get at least 6-8 instructions between branches. This should provide a significant performance benefit.

The branch-likely instructions deprecated by the MIPS32 architecture document are predicted just like any other branch. The misprediction of branch-likely instructions costs an extra cycle or two, because the branch and the delay slot instruction needs to be re-executed after a mispredict. Branch-likely instructions sometimes improve the performance of small loops on the proAptiv Multiprocessing System core, but they set problems for the designers of complex CPUs, and may one day disappear from the standard. Good compilers for the MIPS32 architecture should provide an option to avoid these instructions.

6.2.7 Load Delayed by (Unrelated) Recent Store

Load instructions are handled within the execution unit with "standard" timing, just so long as they hit in the cache. When a load misses (or turns out to be uncached) then a dependent operation which has already been issued will have to be reissued if the dependent instruction has been dispatched, which can generate additional delay. If the dependent instruction has not been dispatched, it remains in the data queue (DDQ) until the load data becomes available.

Conversely, store instructions are graduated before they are completed. This is because store instructions cannot be written to the cache (or commit a write to real memory) until they graduate and cease to be speculative. This can present a problem in that a programmer may write code which stores a value in memory, then immediately loads the same value. The CPU pipeline detects circumstances where instructions are dependent for register values, but cannot do the same for addresses. As such, the load can get the right data from an incomplete store as a side-effect of checking whether the requested data might be in the FSB (fill/store buffer) attached to the D-cache. In addition, the store data can also be in intermediate stages/queues before being written into the FSB. Any data that matches stores in such intermediate queues will also be bypassed back to the pipeline as if the load hit in the cache.

6.2.8 Minimum Load-miss Penalty

The proAptiv Multiprocessing System core runs at high frequencies, so any load that misses in the L1 D-cache is likely to be substantially delayed, waiting for the memory data to come back from the L2 cache. If the load misses in the L2 cache, a much greater delay in incurred.

If an instruction that consumes the loaded data issues before it is determined that the load has missed, then that instruction will have to be re-executed by stopping execution and starting again on the consuming instruction. This is likely to occur if the consuming instruction is only a few places behind in the instruction sequence. That means it has to be re-fetched from the I-cache, and this involves a delay of approximately 15 cycles.

6.2.9 Data Dependency Delays

The out-of-order pipeline in the proAptiv Multiprocessing System core allows dependent instructions to be executed as soon as possible, in hardware. So to some extent the out-of-order pipeline makes it unnecessary to manage data delays by moving instructions around in the program sequence.

Compilers might reasonably try to schedule code to create more opportunities for dual-issue and so that instructions might be issued at full speed despite dependencies, but should rarely do so if the cost is significant — the hardware is already gaining much of this advantage within its out-of-order window, and compiler scheduling will not be worth many extra instructions or significant code bloat unless it reaches beyond such a window. Loop unrolling will often help, but local scheduling will be unlikely to make a lot of difference.

The MIPS instruction set is efficient for short pipelines because, most of the time, dependent instructions can be run nose-to-tail, just one clock apart, without extra delay. Even in the more sophisticated proAptiv Multiprocessing System family CPUs, most dependent instructions can run just two clocks apart. Each register has a "standard" place in the pipeline where the producer should deliver its value and another place in the pipeline where the consumer picks it up: where those places are 1 cycle apart, the dependent instructions to run in successive cycles. Producer/consumer delays happen when either the producer is late delivering a result to the register, or the consumer insists on obtaining its operand early. If either of these conditions occurs, the delays can add up.

Most of these delays are hidden by out-of-order execution.

Different register classes are read/written in different “standard” pipeline slots, so it’s important to be clear what class of registers is involved in any of these delays. For non-floating-point user-level code, there are just three:

- General purpose registers (“GPR”).
- The multiply unit’s *hi*/*lo* pair together with the three additional multiply-unit accumulators defined by the MIPS DSP ASE (“ACC”).

  The MIPS architecture encourages implementations to provide integer multiply and divide operations in a separate pipelined unit capable of doing multiply-accumulate operations at a rate of one per clock. No multiply unit operation ever causes an exception, which makes the longer multiply-unit pipeline rather invisible. It shows up in late delivery of GPR values by those few multiply-unit instructions which deliver GPR results.
- The fields of the *DSPControl* register, used for condition codes and exceptional conditions resulting from DSP ASE operations.

So that gives us two tables: Table 6.3 for our consumers, and Table 6.4 for the producers.

Table 6.3 Register → Consumer Delays

Reg	→	Consumer	Del	Applies when...
GPR	→	load/store	1	The GPR value is an address operand. Store *data* is not needed early.
ACC	→	multiply instructions	3	The ACC value came from any multiply instruction which saturates the accumulator value.
ACC	→	DSP instructions which extract selected bits from an accumulator: **`extp`**..., **`extr`**... etc.	3	Always
		DSP instructions which write a shifted value back to the accumulator: **`mthlip`**, **`shilo`**, **`shilov`**.		

Table 6.4 Producer → Register Delays

Producer	→	Reg	Del	Applies when...
All bitwise logical instructions, including immediate versions	→	GPR	0	These instructions *only* are “not lazy”: their result can be used in the next cycle by any ALU instruction. Note that **`addu rd,rs,$0`** is used for **`mov`**. Results from **add**, **addi**, **addi** and **addiu** are available to consumers in ALU pipe with 0 delay. Consumers in AGEN pipe will see a delay of 1.
`lui`				
`addu rd,rs,$0` (add zero, aka **`mov`**)				
`sll` with shift amount 8 or less				
`srl` with shift amount 25 or more				
set-on-condition (**`slt`**, **`slti`**, **`sltiu`**, **`sltu`**)				
`seb`, **`seh`**				
`add, addu, addi, addiu`				
Any other ALU instruction	→	GPR	1	2-beat ALU for all but the simplest operations
Non-multiply DSP ASE instructions which don’t saturate.				

Table 6.4 Producer → Register Delays

DSP “ALU” instructions (which neither read nor write an accumulator, nor do a multiplication), but do saturate.	→	GPR	2	Always
Conditional move **movn**, **movz**	→	GPR	3	Run in the AGEN pipeline. They create trouble because they implicitly have three register operands (the “no-move” case is handled by reading the original value of the destination register and writing it back) — but in proAptiv Multiprocessing System CPUs an instruction may only use two read ports in the register file. So a conditional move instruction is issued in two consecutive clock phases: one to do the move, one to fetch the original value and write it back again. That makes sure that the right value is available in the CB entry and the pipeline by-passes.
Any load	→	GPR	2	That’s a cached load which hits, of course.
sc (store conditional)	→	GPR	8	The GPR is receiving the success/failure code. The instruction which consumes this code is not issued until the store has graduated and been acted on. The delay could be longer if there is work queued up in the load/store pipe, but in the normal **ll**/**sc** busy loop the dependency on the **ll** load will have left the pipe idle.
Integer multiply instructions producing a GPR result (**mul**, **mulu** etc).	→	GPR	6	Always (because the multiply unit pipeline is longer than the integer unit’s).
Instructions reading accumulators and writing GPR (e.g. **mflo**).				
div /**divu**	→	ACC	10-20	Dividend 255 or less
			10-50	Dividend 256 or more

How to Use the Tables

Assume an instruction sequence like this one:

```
addiu     $a0, $a0, 8
lw        $t0, 0($a0)   # [1]
lw        $t1, 4($a0)
addu      $t2, $t0, $t1# [2]
mul       $v0, $t2, $t3
sw        $v0, 0($a1)   # [3]
```

A review of the tables above should help determine whether any instructions will be held up. Look at the dependencies where an instruction is dependent on its predecessor. In this example, the following delays will occur:

[1] The **lw** will be held up by two clocks. One clock because **addiu** takes 2 clocks to produce its result, and another because its GPR address operand **$a0** was computed by the immediately preceding instruction (see the “load/store address” box of Table 6.3.) The second **lw** will be OK.

[2] The **addu** will be two clocks late, because the load data from the preceding **lw** arrives late in the GPR **$t1** (see the “load” box of Table 6.4.)

[3] The **sw** will be 6 clocks late starting while it waits for a result from the multiply pipe (the “multiply” box of Table 6.4.)

These can be additive. In the pointer-chasing sequence:

```
lw          $t1, 0($t0)
lw          $t2, 0($t1)
```

The second load will be held up three clocks: two because of the late delivery of load data in **`$t1`** ("load" box of Table 6.4), plus another because that data is required to form the address ("load/store address" box of Table 6.3.)

Delays Caused by Dependencies on DSPControl Fields

Some DSP ASE instructions are dependent because they produce and consume values kept in fields of the *DSPControl* register. However, the most performance-critical of these dependencies are "by-passed" to make sure no delay will occur - those are the dependencies between:

`addsc`	→	`DSPControl[c]`	→	**`addwc`**
`cmp.x`	→	`DSPControl[ccond]`	→	**`pick.x`**
`wrdsp`	→	`DSPControl[pos,scount]`	→	**`insv`**

But other dependencies passed in *DSPControl* may cause delays; in particular the *DSPControl[ouflag]* bits set by various kinds of overflow are not ready for a succeeding **`rddsp`** instruction. The access is interlocked, and will lead to a delay of up to three clocks.

6.2.9.1 More Complicated Dependencies

There can be delays which are dependent on the dynamic allocation of resources inside the CPU. These delays cannot be easily determined by doing a static code analysis. MIPS recommends using a cycle-accurate simulator or other trace equipment to help determine these types of delays.

6.2.10 Advice on Tuning Instruction Sequences (particularly DSP)

DSP algorithm functions are often the subject of intense tuning. There are four basic classes of DSP instructions:

- A group of specially-simple ALU instructions run in one cycle. This includes bitwise logical instructions, **`mov`** (an alias for **`addu`** with **`$0`**), shifts up to 8 positions down or up, test-and-set instructions, and sign-extend instructions.

- Simple DSP ASE operations (no multiply, no saturation) have 2-cycle latency, the same as most regular MIPS32 arithmetic.

- Non-multiply DSP instructions which feature saturation or rounding have 3-cycle latency.

- Special DSP multiply operations (or any other access to the multiply unit accumulators): these have timings like standard multiply and multiply-accumulate instructions, so they're in with the multiply operations under the next heading.

- Instruction dependencies relating to different fields in the *DSPControl* register are tracked separately, and efficiently, as if they were separate registers. But any **`rddsp`** or **`wrdsp`** instruction which reads/writes multiple fields at once is dependent on multiple fields, and that can't be tracked through the CB system. Such a **`rddsp`** is not issued until all predecessors have graduated, and such a **`wrdsp`** must graduate before its successors can issue. You can often avoid this by using the "masked" versions of these instructions to read or write only a particular field.

6.2.11 Multiply/Divide Unit and Timings

As is traditional with MIPS CPUs, the integer multiplier is a semi-detached unit with its own pipeline. This pipeline implements:

- **mult**/**multu**: multiply two 32-bit numbers from GPRs (signed and unsigned versions) with a 64-bit result delivered in the multiply unit's accumulator. The accumulator was traditionally seen as pseudo-registers *hi* and *lo*, readable only using the special instructions **mfhi** and **mflo**.Operations into the accumulator do not hold up the main CPU and run independently, but **mfhi**/**mflo** are interlocked and delay execution as required until the result is available.

- **madd**, **maddu**, **msub**, **msubu**: multiply/accumulate instructions collecting their result in the accumulator.

- **mul**/**mulu**: simple 3-operand multiply as a single instruction.

- **div**/**divu**: divide - the quotient goes into *lo* and the remainder into *hi*.

Many of the most powerful instructions in the MIPS DSP ASE are variants of multiply or multiply-accumulate operations, and are described in Chapter 13, "MIPS DSP-R2 Application-Specific Extension" on page 549. The DSP ASE also provides three additional "accumulators" which behave like the *hi*/*lo* pair: the now four accumulators are called *ac0-3*). When we talk about the "multiply/divide" group of instructions we include any instruction which reads or writes any accumulator.

No multiply/divide operation ever produces an exception - even divide-by-zero is silent — compilers typically insert explicit check code where it's required.

Timing varies. Multiply-accumulate instructions (there are many different flavors of MAC in the DSP ASE) have been pipelined and tuned to achieve a 1-instruction-per-clock repeat rate, even for sequences of instructions targeting the same accumulator. But because that requires a relatively long pipeline, multiply/divide unit instructions which produce a result in a GP register are relatively "slow": for example, an instruction consuming the register value from a **mflo** will not be issued until at least 7 cycles after the **mflo**.

What that means is that in an instruction sequence like:

```
mult $1, $2
mflo $3
addu $2, $3, 1
```

The **mflo** will be issued 4 cycles after the **mult**, and the **addu** will go at least 2 cycles after the **mflo**. The execution unit may (or may not) be able to find other instructions to keep it busy, but each trip through that code sequence will take a minimum of 9 cycles.

6.3 Kernel Mode Programming

This section covers the following topics:

- Section 6.3.1, "Hazard Barrier Instructions"
- Section 6.3.2, "MIPS32® Architecture Release 3 - Enhanced Interrupt System(s)"
- Section 6.3.3 "External Interrupt Controller (EIC) Mode"

6.3.1 Hazard Barrier Instructions

When privileged "CP0" instructions change the machine state, unexpected behavior can occur if an instruction is deferred out of its normal instruction sequence. But that can happen because the relevant control register only gets written some way down the pipeline, or because the changes it makes are sensed by other instructions early in their pipeline sequence: this is called a CP0 *hazard*.

The proAptiv Multiprocessing System core provides the option of removing many CP0 hazards by setting the *Config7[IHB]* bit. It's possible to get hazards in user mode code too, and many of the instructions described here are not solely for kernel-privilege code. But they're most often met around CP0 read/writes.

Traditionally, MIPS CPUs left the kernel/low-level software engineer with the job of designing sequences which are guaranteed to run correctly, usually by padding the dangerous operation with enough **`nop`** or **`ssnop`** instructions.

To help manage pipeline hazards, the proAptiv core implements explicit *hazard barrier* instructions. If a hazard barrier instruction is executed between the instruction which makes the change (the "producer") and the instruction which is sensitive to it (the "consumer"), you are guaranteed that the change will be seen as complete. Hazards can appear when the producer affects even the instruction fetch of the consumer - that's an "instruction hazard" - or only affecting the operation of the consuming instruction (an "execution hazard"). Hazard barriers come in two strengths: **`ehb`** deals only with execution hazards, while **`eret`**, **`jr.hb`** and **`jalr.hb`** are barriers to both kinds of hazard.

In most implementations the strong hazard barrier instructions are quite costly, often discarding most or all of the pipeline contents: they should not be used indiscriminately. For efficiency you should use the weaker **`ehb`** where it is enough. Since some implementations work by holding up execution of all instructions after the barrier, it's preferable to place the barrier just before the consumer, not just after the producer.

The following tables list the execution hazards and the instruction hazards for the proAptiv core.

6.3.1.1 Execution Hazards

Execution hazards are those created by the execution of one instruction, and seen by the execution of another instruction. Table 6.5 lists possible execution hazards and whether they can be resolved via setting of the IHB bit in the CP0 Config7 register.

Table 6.5 Execution Hazards

Producer	→	Consumer	Hazard On	Does Config7.IHB=1 resolve this Hazard?
TLBWR, TLBWI , TLBINV, TLBINVF	→	Load/store using new TLB entry	TLB entry	No
MTC0	→	Load/store affected by new state	*WatchHi* *WatchLo*	No

Table 6.5 Execution Hazards *(continued)*

Producer	→	Consumer	Hazard On	Does Config7.IHB=1 resolve this Hazard?
MTC0	→	MFC0	Any CP0 register	No
MTC0	→	EI/DI	*Status*	Yes
MTC0	→	RDHWR $3	*Count*	No
MTC0	→	Coprocessor instruction execution depends on the new value of Status_{CU}	$Status_{CU}$	No
MTC0	→	ERET	*EPC* *DEPC* *ErrorEPC*	Yes
MTC0	→	ERET	*Status*	Yes
EI, DI	→	Interrupted instruction	$Status_{IE}$	No
MTC0	→	Interrupted instruction	*Status*	No
MTC0	→	User-defined instruction	$Status_{ERL}$ $Status_{EXL}$	No
MTC0	→	Interrupted Instruction	$Status_{IM}$ $(Cause_{IP})$	No
TLBR	→	MFC0	*EntryHi,* *EntryLo0,* *EntryLo1,* *PageMask*	Yes
TLBP	→	MFC0	*Index*	Yes
MTC0 *ContextConfig*	→	MFC0	*Context* *ContextConfig*	Yes
MTC0	→	RDPGPR WRPGPR	$SRSCtl_{PSS}$	No
MTC0	→	Instruction not seeing a Timer Interrupt	Compare update that clears Timer Interrupt	No
MTC0	→	Instruction affected by change	Any other CP0 register	No
CACHE	→	MFC0	*TagHi*, *TagLo*, *DataHi*, *DataLo*	Yes

6.3.1.2 Instruction Hazards

Instruction hazards are those created by the execution of one instruction, and seen by the instruction fetch of another instruction. Table 6.6 lists the instruction hazards. Because the fetch unit is decoupled from the execution unit, these hazards are rather large. The use of a hazard barrier instructions is required for reliable clearing of instruction hazards.

Table 6.6 Instruction Hazards

Producer	→	Consumer	Hazard On
TLBWR, TLBWI, TLBINV, TLBINVF	→	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*$_{ASID}$
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

6.3.2 MIPS32® Architecture Release 3 - Enhanced Interrupt System(s)

The features for handling interrupts include:

- Vectored Interrupt (VI) mode offers multiple entry points (one for each of the interrupt sources), instead of the single general exception entry point.

  External Interrupt Controller (EIC) mode goes further, and reinterprets the six CPU interrupt input signals as a 64-value field - potentially 63 distinguished interrupts each with their own entry point (the zero code, of course, is reserved to mean “no interrupt active”).

  Both these modes need to be explicitly enabled by setting bits in the *Config3* register; if you don’t do that, the CPU behaves just as the original (release 1) MIPS32 specification required.

- Shadow registers - alternate sets of registers, often reserved for interrupt handlers, are described in Section 6.5, "Saving Power". Interrupt handlers using shadow registers avoid the overhead of saving and restoring user GPR values.

- The *Cause[TI]*, *Cause[FDCI]*, and *Cause[PCI]* bits provide a direct indication of pending interrupts from the on-CPU timer, fast debug channel, and performance counter subsystems (these interrupts are potentially shared with other interrupt inputs, and it previously required system-specific programming to discover the source of the interrupt and handle it appropriately).

The new interrupt options are enabled by the *IntCtl* register, whose fields are shown in Figure 6.3.

Figure 6.3 Fields in the IntCtl Register

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

IntCtl[IPTI,IPPCI,IPFDCI]: *IPTI*, *IPPCI*, and *IPFDCI* are read-only 3-bit fields. These fields indicate how the internal timer, performance counter, and fast debug channel interrupts are wired up. They are relevant in non-vectored and simple-vectored ("VI") interrupt modes, but have no meaning when using an EIC interrupt controller.

Read this field to get the number of the *Cause[IPnn]* where the corresponding interrupt is seen. Because *Cause[IP1-0]* are software interrupt bits, unconnected to any input, legal values for *IntCtl[IPTI], IntCtl[IPPCI]* , and *IntCtl[IPFDCI]* are between 2 and 7.

The timer, performance counter, and fast debug channel interrupt signals are taken out to the CPU interface and the SoC designer connects them back to one of the CPU's interrupt inputs. The SoC designer is supposed to hard-wire some CPU inputs which show up as the *IntCtl[IPTI,IPPCI,IPFDCI]* fields to match.

IntCtl[VS]: is writable to provide software control of the interrupt vector spacing. The spacing is calculated as $32 \times 2^{(VS-1)}$ bytes.

VS values of 1, 2, 4, 8 and 16 work provide 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. This would be the legacy behavior.

6.3.2.1 Traditional MIPS® Interrupt Signalling and Priority

In previous generation MIPS processors, the CPU takes an interrupt exception on any cycle where one of the eight possible interrupt sources visible in *Cause[IP]* is active, enabled by the corresponding enable bit in *Status[IM]*, and not otherwise inhibited. When that happens control is passed to the general exception handler and is recognized by the "interrupt" value in *Cause[ExcCode]*. All interrupts are equal in the hardware, and the hardware does nothing special if two or more interrupts are active and enabled simultaneously. All priority decisions are made by software.

Six of the interrupt sources are hardware signals brought into the CPU, while the other two are "software interrupts" taking whatever value is written to them in the *Cause* register.

The original MIPS32 specification adds an option to this. If you set the *Cause[IV]* bit, the same priority-blind interrupt handling happens but control is passed to an interrupt exception entry point which is separate from the general exception handler.

6.3.2.2 VI Mode - Multiple Entry Points, Interrupt Signalling and Priority

The traditional interrupt system commonly has a single piece of code which does the housekeeping associated with interrupts prior to calling an individual device-interrupt handler. However, a single entry point does not always fit well with embedded systems using very low-level interrupt handlers. These types of applications perform best when multiple entry points are provided. To accommodate this, the proAptiv architecture implements the "VI interrupt mode" where interrupts are despatched to one of eight possible entry points. To make this happen:

1. *Config3[VInt]* must be 1 to indicate that the CPU has the vectored-interrupts feature. This is a read-only bit that is always set in the proAptiv core to indicate support for vectored interrupts.

2. Set the *Cause[IV]* bit to request that interrupts use the special interrupt entry point.

3. Set the *IntCtl[VS]* to a non-zero value, setting the spacing between successive interrupt entry points. The proAptiv core allows spacing of 32, 64, 128, 256, and 512 bytes between entry points.

Interrupt exceptions vector to one of eight distinct entry points. The bit-number in *Cause[IP]* corresponding to the highest-numbered active interrupt becomes the “vector number” in the range 0-7. The vector number is multiplied by the “spacing” implied by the OS-written field *IntCtl[VS]* (see above) to generate an offset. This offset is then added to the special interrupt entry point (already an offset of 0x200 from the value defined in *EBase*) to produce the entry point to be used.

If multiple interrupts are active and enabled, the entry point will be the one associated with the higher-numbered interrupt: in VI mode interrupts are no longer all equal, and the hardware now has some role in interrupt “priority”.

6.3.3 External Interrupt Controller (EIC) Mode

Embedded systems have lots of interrupts, typically far exceeding the six input signals traditionally available. Most systems have an external interrupt controller to allow these interrupts to be masked and selected. In the proAptiv core, EIC mode allows the six interrupt input signals to be encoded to allows up to 63 distinct interrupt entry points. In a 4-core system, this would allow for up to 256 distinct interrupts.

To do this the same six hardware signals used in traditional and VI modes are redefined as a bus with 64 possible values: 0 means “no interrupt” and 1 - 63 represent distinct interrupts. That’s “EIC interrupt mode”, and you’re in EIC mode if you would be in VI mode (see previous section) and additionally the *Config3[VEIC]* bit is set. EIC mode is a little deceptive: the programming interface hardly seems to change, but the meaning of fields change quite a bit.

Firstly, once the interrupt bits are grouped the interrupt mask bits in *Status[IM]* can’t just be bitwise enables any more. Instead this field (strictly, the 6 high order bits of this field, excluding the mask bits for the software interrupts) is recycled to become a 6-bit *Status[IPL]* (“interrupt priority level”) field. Most of the time (when running application code, or even normal kernel code) *Status[IPL]* will be zero; the CPU takes an interrupt exception when the interrupt controller presents a number higher than the current value of *Status[IPL]* on its “bus” and interrupts are not otherwise inhibited.

As before, the interrupt handler will see the interrupt request number in *Cause[IP]* bits; the six MS of those bits are now relabelled as *Cause[RIPL]* (“requested IPL”). In EIC mode the software interrupt bits are not used in interrupt selection or prioritization: see below. But there’s an important difference; *Cause[RIPL]* holds a snapshot of the value presented to the CPU when it decided to take the interrupt, whereas the old *Cause[IP]* bits simply reflected the real-time state of the input signals[2].

When an exception is triggered the new IPL - as captured in *Cause[RIPL]* - is used directly as the interrupt number; it’s multiplied by the interrupt spacing implied by *IntCtl[RS]* and added to the special interrupt entry point, as described in the previous section. *Cause[RIPL]* retains its value until the CPU next takes any exception.

Software interrupts: the two bits in *Cause[IP1-0]* are still writable, but now become real signals which are fed out of the CPU CPU, and in most cases will become inputs - presumably low-priority ones - to the EIC-compliant interrupt controller.

In EIC mode the usual association of the internal timer, performance-counter overflow, and fast debug channel interrupts with individual bits of *Cause[IP]* is lost. These interrupts are turned into output signals from the CPU, and will themselves become inputs to the interrupt controller. Ask your system integrator how they are wired.

[2] Since the incoming IPL can change at any time - depending on the priority views of the interrupt controller - this is essential if the handler is going to know which interrupt it’s servicing.

6.4 Exception Entry Points

Early versions of the MIPS architecture had a rather simple exception system, with a small number of architecture-fixed entry points.

But there were already complications. When a CPU starts up main memory is typically random and the MIPS caches are unusable until initialized; so MIPS CPUs start up in uncached ROM memory space and the exception entry points are all there for a while (in fact, for so long as *Status[BEV]* is set); these “ROM entry points” are clustered near the top of kseg1, corresponding to 0x1FC0.0000 physical[3], which must decode as ROM.

ROM is slow and rigid; handlers for some exceptions are performance-critical, and OS’ want to handle exceptions without relying on ROM code. So once the OS boots up it’s essential to be able to redirect OS-handled exceptions into cached locations mapped to main memory (what exceptions are not OS-handled? well, there are no alternate entry points for system reset, NMI, and EJTAG debug).

So when *Status[BEV]* is flipped to zero, OS-relevant exception entry points are moved to the bottom of kseg0, starting from 0 in the physical map. The cache error exception is an exception... it would be silly to respond to a cache error by transferring control to a cached location, so the cache error entry point is physically close to all the others, but always mapped through the uncached “kseg1” region.

In MIPS CPUs prior to the MIPS32 architecture (with a few infrequent special cases) only common TLB miss exceptions got their own entry point; interrupts and all other OS-handled exceptions were all funneled through a single “general” exception entry point.

The CP0 EBase Register

The *EBase* register provides the ability for software to identify the specific processor within a multi-processor system, and allows the exception vectors for each processor to be different. Bits 31:12 of the *EBase* register are concatenated with zeros to form the base of the exception vectors when $Status_{BEV}$ is 0. The exception vector base address comes from the fixed defaults when $Status_{BEV}$ is 1, or for any EJTAG Debug exception. The reset state of bits 31:12 of the *EBase* register initialize the exception base register to `0x8000.0000`, providing backward compatibility with Release 1 implementations.

The size of the ExcBase field depends on the state of the WG bit. At reset, the WG bit is cleared by default. In this case, the ExcBase field is comprised of bits 29:12. Bits 31:30 of the EBase Register are not writeable and are forced to a value of 2'b10 by hardware so that the exception handler will be executed from the *kseg0/kseg1* segments.

When the WG bit is set, bits 31:30 of the ExcBase field become writeable and are used to relocate the ExcBase field to other segments after they have been setup using the *SegCtl0* through *SegCtl2* registers. Note that if the WG bit is set by software (allowing bits 31:30 to become part of the ExcBase field) and then cleared, bits 31:30 can no longer be written by software and the state of these bits remains unchanged for any writes after WG was cleared. Therefore, it is the responsibility of software to write a value of 2'b10 to bits 31:30 of the EBase register prior to clearing the WG bit if it wants to ensure that future exceptions will be executed from the kseg0 or kseg1 segments.

Refer to Section 2.3.1.10, "Exception Base Address — EBase (CP0 Register 15, Select 1)" for more information.

[3] Even this address can be changed by a brave and determined SoC integrator, see the note on RBASE in Section 6.4.1 “Summary of Exception Entry Points”.

6.4.1 Summary of Exception Entry Points

The incremental growth of exception entry points has left no one place where all the entry points are summarized; so here’s Table 6.7. But first:

> BASE is 0x8000.0000, as it will be where the software, ignoring the *EBase* register, leaves it at its power-on value — that’s also compatible with older MIPS CPUs. Otherwise BASE is the 4Kbyte-aligned address found in *EBase* after you ignore the low 12 bits...
>
> RBASE is the ROM/reset entry point base, usually `0xBFC0.0000`. However, proAptiv Multiprocessing System family CPUs can be configured to use a different base address by fixing some input signals to the CPU. Specifically, if the CPU is wired with *SI_UseExceptionBase* asserted, then RBASE bits 29-12 will be set by the values of the inputs *SI_ExceptionBase[29:12]* (the two high bits will be “10” to select the kseg0/kseg1 regions, and the low 12 bits are always zero). Relocating RBASE is strictly not compliant with the MIPS32 specification and may break all sorts of useful pieces of software, so it’s not to be done lightly.
>
> DebugVectorAddr is an alternative entry point for debug exceptions. It is specified via a drseg memory mapped register of the same name and enabled through the Debug Control Register. The probe handler still takes precedence, but this is higher priority than the regular ROM entry points.

Table 6.7 All Exception entry points

Memory region	*Entry point*	*Exceptions handled here*
EJTAG probe-mapped	`0xFF20.0200`	EJTAG debug, when mapped to “probe” memory.
Alternate Debug Vector	DebugVectorAddr	EJTAG debug, not probe, relocated, *DCR[RDVec]*==1
ROM-only entry points	`RBASE+0x0480`	EJTAG debug, when using normal ROM memory.*DCR[RDVec]*==1
	`RBASE+0x0000`	Post-reset and NMI entry point.
ROM entry points (when *Status[BEV]*==1)	`RBASE+0x0200`	Simple TLB Refill (*Status[EXL]*==0).
	`RBASE+0x0300`	Cache Error. Note that regardless of any relocation of RBASE (see above) the cache error entry point is always forced into kseg1.
	`RBASE+0x0400`	Interrupt special (*Cause[IV]*==1).
	`RBASE+0x0380`	All others
“RAM” entry points (*Status[BEV]*==0)	`BASE+0x100`	Cache error - in RAM. but always through uncached kseg1 window.
	`BASE+0x000`	Simple TLB Refill (*Status[EXL]*==0).
	`BASE+0x200`	Interrupt special (*Cause[IV]*==1).
	`BASE+0x200+...`	multiple interrupt entry points - seven more in “VI” mode, 63 in “EIC” mode; see Section 6.3.2, "MIPS32® Architecture Release 3 - Enhanced Interrupt System(s)".
	`BASE+0x180`	All others

6.5 Saving Power

There are just a couple of facilities:

- The **`wait`** instruction causes the CPU to enter a low-power sleep mode until woken by an interrupt. Most of the CPU logic is stopped, but the *Count* register, in particular, continues to run.

- The *Status[RP]* bit: this doesn’t do anything inside the CPU, but its state is made available at the CPU interface as *SI_RP*. Logic outside the CPU is encouraged to use this to control any logic which trades off power for speed - most often, that will be slowing the master clock input to the CPU.

- Via the Cluster Power Controller, it is possible to gate off the clocks or even the power going to an idle CPU. This functionality is described in the *MIPS32® proAptiv Multiprocessing System™ Multiprocessing System Hardware User's Manual*.

Chapter 7

Power Management and the Cluster Power Controller

This chapter describes the Cluster Power Controller (CPC) included in the proAptiv Multiprocessing System. The CPC organizes bootstrap, reset, tree root clock gating, and power gating of CPUs. The CPC also manages power cycling, reset, and clock gating of the Coherence Manager, dependent on the individual core status and shutdown policy.

The chapter contains the following sections:

- Section 7.1 “Introduction to the Cluster Power Controller”
- Section 7.3 “Cluster Power Controller Address Map”
- Section 7.4 “Cluster Power Controller Commands”
- Section 7.5 “proAptiv Core Power Management Options”
- Section 7.6 “proAptiv Core Clock Gating”
- Section 7.7 “proAptiv Core Power Gating”

7.1 Introduction to the Cluster Power Controller

The Cluster Power Controller (CPC) works in conjunction with the power management features of the individual proAptiv cores to provide a comprehensive power management scheme.

The main purpose of the Cluster Power Controller (CPC) is to manage static leakage and dynamic power consumption based on system-level power states assigned to the individual components of the proAptiv Multiprocessing System. As such, the CPC acts as a programmable platform peripheral, accessible through cluster CPU software and SOC-level hardware protocols.

The CPC is an integral part of the coherent cluster and is designed to boostrap, reset, tree root clock-gate and power-gate cluster CPUs and the Coherence Manager. Implementors may or may not chose to support some or all of the physical features the CPC is architected to control. The following physical power-management features can be selected independently:

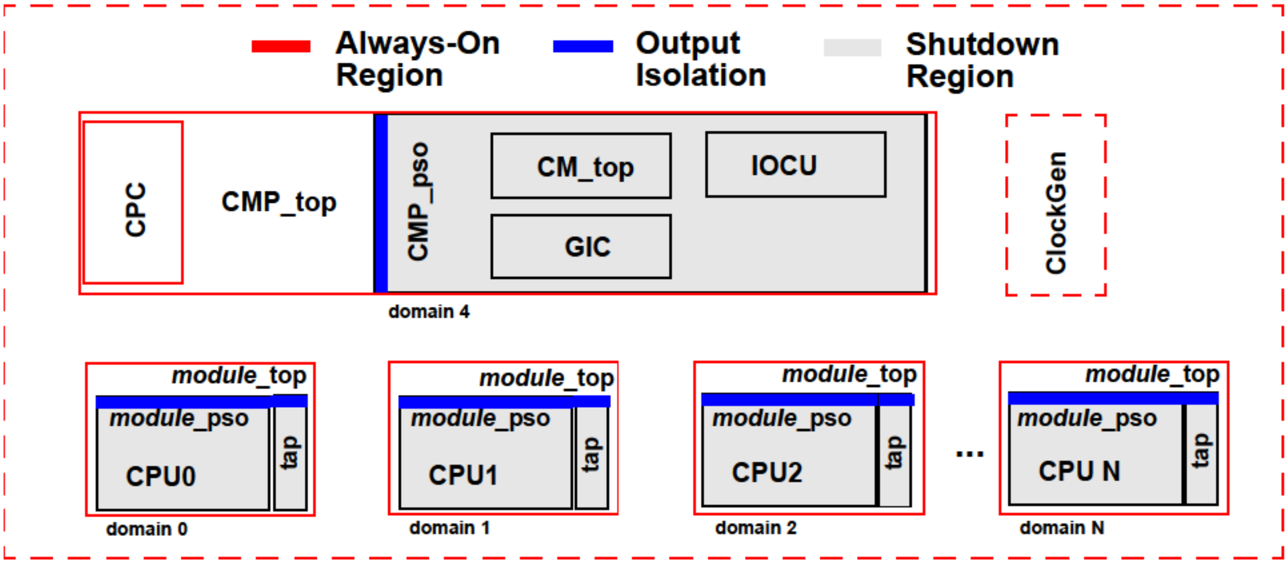
- **Power gating of selected CPUs and/or the CM.** Supported by industry-standard physical design flows, supply voltage of individual power domains can be switched on-chip. Currently, the Common Power Format (CPF) and Unified Power Format (UPF) are provided for a seamless front to back-end design flow. Besides CPF/UPF compliant EDA tools, standard cell libraries are required to provide power-gating header or footer cells, as well as isolate-high and isolate-low cells to separate unpowered domains from their active surroundings. The CPC provides a front-end RTL simulation environment and diagnostics to verify power-gating behavior.
- **Tree root clock gating.** Independent of CPU internal power-management features such as register-bank level clock gating and the sleep and doze modes, the CPC provides controls to gate clocks directly at or after

the PLL in order to quiesse the entire clock tree of a CPU. CPC clock-gating signals are designed to bridge large clock insertion delays and are controlled through system-level power states.

In addition to power-management functions, the CPC also acts as reset and boostrap controller of the Multiprocessing System (MPS) to initialize cores as they become operational, or re-initialize them upon system-level requests. The CPC also facilitates EJTAG debug probe access to cores by detecting the connection of a probe and enabling cores to respond to debug interrupt requests.

7.1.1 Power Domains of the proAptiv Multiprocessing System

Figure 7.1 proAptiv Multiprocessing System Power Domains



To individually power gate each core, independently controlled power domains are introduced to the proAptiv Multiprocessing System. RTL simulation as well as physical implementation of the CPS support five distinct domains, cpu0-N and the Coherence Manager. These components are intended to be implemented with power rail switch cells to allow shutdown. Each controllable domain also is required to drive isolation values towards the system. This ensures proper logic values from shutdown domain boundaries into powered surroundings.

The CPS top level can be implemented to belong to a voltage scaled supply domain. This enables dynamic voltage and frequency scaling over the full CPS with shutdown features for individual subdomains.

With shutdown of all cores, the Coherence Manager becomes inactive unless IOCU traffic is requested. The CPC provides programmable power down for these components.

Level 2 cache is part of the CPS. However, power management of the L2 cache is not handled by the CPC. The CMP cluster implementation will ensure that power-down of cores and Coherence Manager does not affect L2 status.

7.1.2 Operating Level Transitions

To reach power-down and clock-off mode, software and hardware are required to go through a sequence of steps on each operating level to reach the next level.

7.1.2.1 Coherent to Non-Coherent Mode Transition

To leave the coherent domain and operate independently or prepare for shutdown, the following sequence should be followed:

1. Switch to non-coherent CCA.

2. Flush dirty data from data cache using IndexWritebackInvalidate CACHE instruction on all lines in the cache.

3. If the instruction cache contains lines that are expected to be maintained by software as coherent (via globalized CACHE instructions), and the CPU is not going to go through a reset sequence, the instruction cache should be flushed using IndexInvalidate CACHE instructions.

4. Write GCR_CL_COHERENCE (Core Local GCR address 0x0008). Write 0 to all bits except bit for "self", which should stay set to 1. This is required so that the core can issue a coherent SYNC (step 6) to make sure all previous interventions are complete.

5. Read GCR_CL_COHERENCE (ensures step 4 has completed).

6. Issue Coherent SYNC (intervention-only SYNC is fine).

7. Write 0 to GCR_CL_COHERENCE to completely remove core from coherence domain.

8. Read GCR_CL_COHERENCE to ensure step 7 is complete.

7.1.2.2 Non-Coherent to Coherent Mode Transition

An independently operating core becomes a member of a coherent cluster.

- Caches must be initialized first (since last reset)

- There should be no data in the caches that will later be accessed coherently. Non-coherent data is treated as exclusive/modified which can lead to violations of the coherence protocol if other caches have copies of the data.

- The GCR local coherence control register is programmed to add the core to the coherent domain.

- Switch to coherent Cache Coherence Attribute (CCA).

- Regular coherent programs can now start on this core.

7.1.2.3 Non-Coherent to Power Down Mode Transition

A core which is not member of a coherent domain is powered down. NOTE: When an EJTAG probe is detected, the CPC will prevent power down to preserve the connectivity of the TAP scan chain. A power-down command will instead cause the core to enter clock off mode.

- The GIC might be programmed to re-route interrupts away from this core.

- The CPC must be programmed to enter power-down mode.

- Core outputs are held inactive towards the CM. Completion of pending bus traffic is awaited and start of new traffic prevented using the *SI_LPReq* protocol.

- The CPC initiates the clock and power shutdown micro-sequence.

7.1.2.4 Non-Coherent to Clock Off Mode Transition

A core is disconnected from bus and stops operation. Dynamic power consumption is removed.

- Programming a CPC ClkOff command will disable the clock tree root for this core.
- Core outputs are held inactive towards the CM. Completion of pending bus traffic is awaited and start of new traffic prevented using the *SI_LPReq* protocol.
- The GIC might be programmed to re-route interrupts for this core to others.

7.1.2.5 Clock Off to Power Down Mode Transition

Power supply is removed from a disconnected core. Dynamic and leakage power is removed.

- The CPC must be programmed to enter power-off mode.
- The CPC initiates the clock and power shutdown micro-sequence.

7.1.2.6 Clock Off to Non-Coherent Mode Transition

A disconnected core is reconnected to the bus and starts operation.

- The CPC command register is programmed to bring the core back on-line. A CPC_PwrUp command will let the core resume operation immediately, or, if a Reset command given, go through a reset sequence before becoming operational.
- If the core bus was isolated due to earlier power modes, this isolation is removed.
- The clock is applied and the core starts executing instructions.

7.1.2.7 PowerDown to Non-Coherent Mode Transition

A core is powered up and becomes operational.

- The GCR local coherence control register must be set inactive for this core. Powering up into a coherent state with uninitialized caches may corrupt coherent data.
- Software on another core can send a PwrUp or Reset command for this core or an SOC hardware signal can request for the CPC to schedule a power-up sequence targeting non-coherent mode.
- The CPC will schedule a power-up sequence and the core becomes operational outside the coherent domain. After the core becomes operational, execution continues at the boot vector provided while power-up mode reset. NOTE: reset is not automatically applied unless the core really was in the power-down state prior to a PwrUp command or hardware PwrUp signal.
- The GIC might be reprogrammed to perform interrupt routing to this core.

7.2 CPC Register Programming

This section describes some of the programming functions that can be performed via the CPC registers.

7.2.1 Requestor Access to CPC Registers

The CPC allows up to eight requestor's in a system. A requestor can be either a core or an IOCU. The proAptiv core allows up to 8 requestors in a multiprocessing system; six cores and two IOCU's.

The requestor's may not have unrestricted access to the CPC registers. During boot time, software determines which requestor's are provided access to the CPC registers by programming the 8-bit *CPC_ACCESS_EN* field of the *Global CPC Access Privilege* register located at offset 0x000. Each bit in this field corresponds to a specific requestor.

The MIPS default for this field is 0xFF, meaning that all requestor's in the system have access to the CPC register set. To disable access to the registers for a particular requestor, software need only clear the corresponding bit of this field to zero and all write requests to the CPC registers by that requestor will be ignored.

7.2.2 Global Sequence Delay Count

The Sequence Delay register (*CPC_SEQDEL_REG*) located at offset 0x0008 in the CPC Global Control Block, contains a 10-bit field that describes the number of clock cycles each domain micro-sequencer will take to advance. It describes a set of worst-case timing of the physical implementation and is used to ensure electrical and bus protocol integrity. Typically, the *CPC_SEQDEL_REG* contents would be defined at IP configuration time. However, runtime write capability allows fine tuning to optimize sequencer timing. Domain sequencing begins once the RAILDELAY field has counted down to zero. Refer to Section 7.2.3, "Rail Delay" for more information.

The 10-bit MICROSTEP field is encoded as follows:

Table 7.1 Encoding of MICROSTEP Field

Encoding	Description
0x000	1-cycle delay
0x001	2-cycle delay
0x002	3-cycle delay
0x003	4-cycle delay
0x004	5-cycle delay
.....	.....
0x3FD	1022-cycle delay
0x3FE	1023-cycle delay
0x3FF	1024-cycle delay

Note that the physical implementation might not allow power sequence micro steps to advance with full cluster speed. At cluster cold start, the counter divides cluster frequency by a hardcoded IP configuration value to derive a micro step width.

*****Need to elaborate on the last paragraph. Is this still relevant** How does this fit into the drawing below?*****

7.2.3 Rail Delay

The Rail Delay register (*CPC_RAIL_REG*) located at offset 0x010 in the CPC Global Register Block contains a 10-bit counter field (*RAILDELAY*) used to schedule delayed start of power domain sequencing after the *RailEnable* signal has been activated by the CPC. This allows the CPC to compensate for slew rates at the gated rail, since hardware interlocks such as *SI_VddOk* are either unavailable or don't reflect to complete power up time of a domain.

The 10-bit counter value delays the power-up sequence per domain after the *SI_RailStable* and *VddOK* signals become active. The power-up micro-sequence starts after RAILDELAY has been loaded into the internal counter and a count-down to zero has concluded.

After completion of the domain power-up micro-sequence, the DomainReady signal is raised and can be used for domain daisy-chaining.

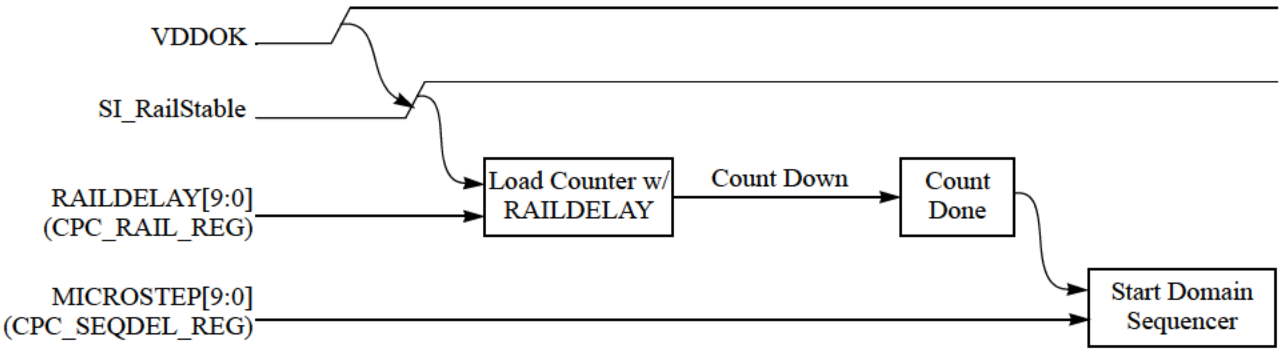
At IP configuration time, the contents of the *CPC_RAIL_REG* register are preset. However, for fine tuning, the register can be written at run time.

The 10-bit RAILDELAY field is encoded as follows:

Table 7.2 Encoding of RAILDELAY Field

Encoding	Description
0x000	1-cycle delay
0x001	2-cycle delay
0x002	3-cycle delay
0x003	4-cycle delay
0x004	5-cycle delay
.....	.....
0x3FD	1022-cycle delay
0x3FE	1023-cycle delay
0x3FF	1024-cycle delay

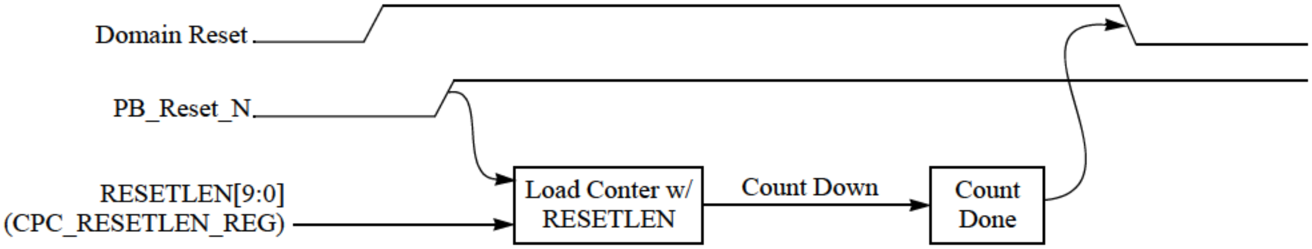
Figure 7.2 Relationship Between RAILDELAY and MICROSTEP During Power-Up Sequence



7.2.4 Reset Delay

Within the power-up micro-sequence, reset is applied. Typically, reset is active until the domain responds by asserting the internal *PB_Reset_N* signal. However, the *CPC_RESETLEN_REG* allows reset to be extended beyond the assertion of *PB_ResetN*. The down-counter starts after the sequencer has detected the assertion of *PB_Reset_N*. Domains without a *PB_ResetN* signal could tie this input low or connect it to an inverted reset signal.

Figure 7.3 Extending the Reset Sequence Beyond the Assertion of the Reset Signal



7.2.5 Executing a Power Sequence

The power sequence for the CPC block support the following commands:

- ClockOff: This command causes the domain to cycle into clock-off mode.It disables the clock to this power domain.
- PwrDown: This command uses the setup values in the CPC_STAT_CONF_REG register.
- PwrUp: This command uses the setup values in the CPC_STAT_CONF_REG register.
- Reset: When this command is issued, the domain is reset if it is in non-coherent mode.

A command can be executed in the local core by writing and encoded value to bits 3:0 of the Command register (CPC_CL_CMD_REG) of the Core-Local block located at offset address 0x000. To write a command to another core, bits 3:0 of the Command reigster (CPC_CO_CMD_REG) in the Core-Other block is used.

7.2.6 Accessing Another Core

To access another core, the number of the core to be accessed is programmed into bits 23:16 of the Core-Other Addressing register (CPC_CL_OTHER_REG) located at offset 0x010 of the Core-Local block. This field selects the core number of the register set to be accessed in Core-Other address space. Refer to Section 7.3.4.2, "Core-Other Addressing Register" for more information.

7.3 Cluster Power Controller Address Map

The CPC uses memory locations within the global, core-local, and core-others address space. The CPC location within the CPU address map is determined by the *GCR_CPC_BASE* register. All address locations in this document are relative to this base address.

In Table 7.3, all registers are accessed using 32-bit aligned uncached load/stores. In addition, the block offsets shown are relative to bits 31:15 of the *GCR_CPC_Base* register located in the CM2. Refer to Chapter 8, *CM2 Global Control Registers* for more information on this register.

Table 7.3 CPC Address Map (Relative to GCR_CPC_BASE[31:15])

Block Offset	Size (bytes)	Description
0x0000 - 0x1FFF	8 KB	**Global Control Block.** Contains registers pertaining to the global system functionality. This address section is visible to all CPUs.
0x2000 - 0x3FFF	8 KB	**Core-Local Control Block**. Aliased for each proAptiv Multiprocessing System core. Contains registers pertaining to the core issuing the request. Each core has its own copy of registers within this block.
0x4000 - 0x5FFF	8 KB	**Core-Other Control Block**. Aliased for each proAptiv Multiprocessing System core. This block of addresses gives each Core a window into another Core's Local Control Block. Before accessing this space, the *Core-Other_Addressing Register* in the Local Control Block must be set to the CORENum of the target Core.

7.3.1 Block Offsets Relative to the Base Address

The block offsets for each of the three blocks listed in Table 7.3 above are relative to a CPC base address and can be located anywhere in physical memory. The base address is a 17-bit value that is programmed into the GCR_CPE_BASE field of the *GCR CPC Base* register located at offset address 0x0088 in the Global Control Block of the CM2 registers. Note that this Global Control Block is different from the one listed in Table 7.3 above. Refer to the *GCR_CPC_BASE Register in* Chapter 8, *CM2 Global Control Registers* for more information on this register.

To determine the physical address of each block listed in Table 7.3, the base address written to the *GCR_CPC_BASE Register* this value would be added to the CPC block offset ranges to derive the absolute physical address as shown in Table 7.4. Note that an example base address of 0x1BDE_0 is used for these calculations.

Table 7.4 Example Physical Address Calculation of the CPC Register Blocks

Example Base Address		GCR Block Offset		Absolute Physical Address	Size (bytes)	Description
0x1BDE_0	+	0x0000 - 0x1FFF	=	0x1BDE_ 0000 - 0x1BDE_1FFF	8 KB	CPC Global Control Block.
0x1BDE_0	+	0x2000 - 0x3FFF	=	0x1BDE_ 2000 - 0x1BDE_3FFF	8 KB	CPC Core-Local Control Block.

Table 7.4 Example Physical Address Calculation of the CPC Register Blocks *(continued)*

Example Base Address		GCR Block Offset		Absolute Physical Address	Size (bytes)	Description
0x1BDE_0	+	0x4000 - 0x5FFF	=	0x1BDE_ 4000 - 0x1BDE_5FFF	8 KB	CPC Core-Other Control Block.

7.3.2 Register Offsets Relative to the Block Offsets

In addition to the block offsets, the register offsets provided in each register description of this chapter are relative to the block offsets shown in Table 7.4 above. To determine the physical address of each register, the base address programmed into the *GCR_CPC_BASE* register is added to the corresponding CPC block offset plus the actual register offset to derive the absolute physical address as shown in Table 7.5. In this table an example base address of 0x1BDE_0 is used.

Table 7.5 Absolute Address of Individual CPC Global Control Block Registers

MIPS Default Base		Global Register Block Offset		Global Register Offset		Absolute Physical Address	Global Control Register
0x1BDE_0	+	0x0000	+	0x0000	=	0x1BDE_0000	CPC Access Privilege.
0x1BDE_0	+	0x0000	+	0x0008	=	0x1BDE_0008	CPC Global Sequence Delay.
0x1BDE_0	+	0x0000	+	0x0010	=	0x1BDE_0010	CPC Rail Delay.
0x1BDE_0	+	0x0000	+	0x0018	=	0x1BDE_0018	CPC Reset Length.
0x1BDE_0	+	0x0000	+	0x0020	=	0x1BDE_0020	CPC Revision.

Table 7.6 shows the absolute physical addresses for the CPC Core-Local block. In this table an example base address of 0x1BDE_0 is used.

Table 7.6 Absolute Address of Individual CPC Core-Local Block Registers

MIPS Default Base		Core-Local Register Block Offset		Core-Local Register Offset		Absolute Physical Address	Core-Local Register
0x1BDE_0	+	0x2000	+	0x0000	=	0x1BDE_2000	CPC Core-Local Command.
0x1BDE_0	+	0x2000	+	0x0008	=	0x1BDE_2008	CPC Core-Local Status and Configuration.
0x1BDE_0	+	0x2000	+	0x0010	=	0x1BDE_2010	CPC Core-Other Addressing.

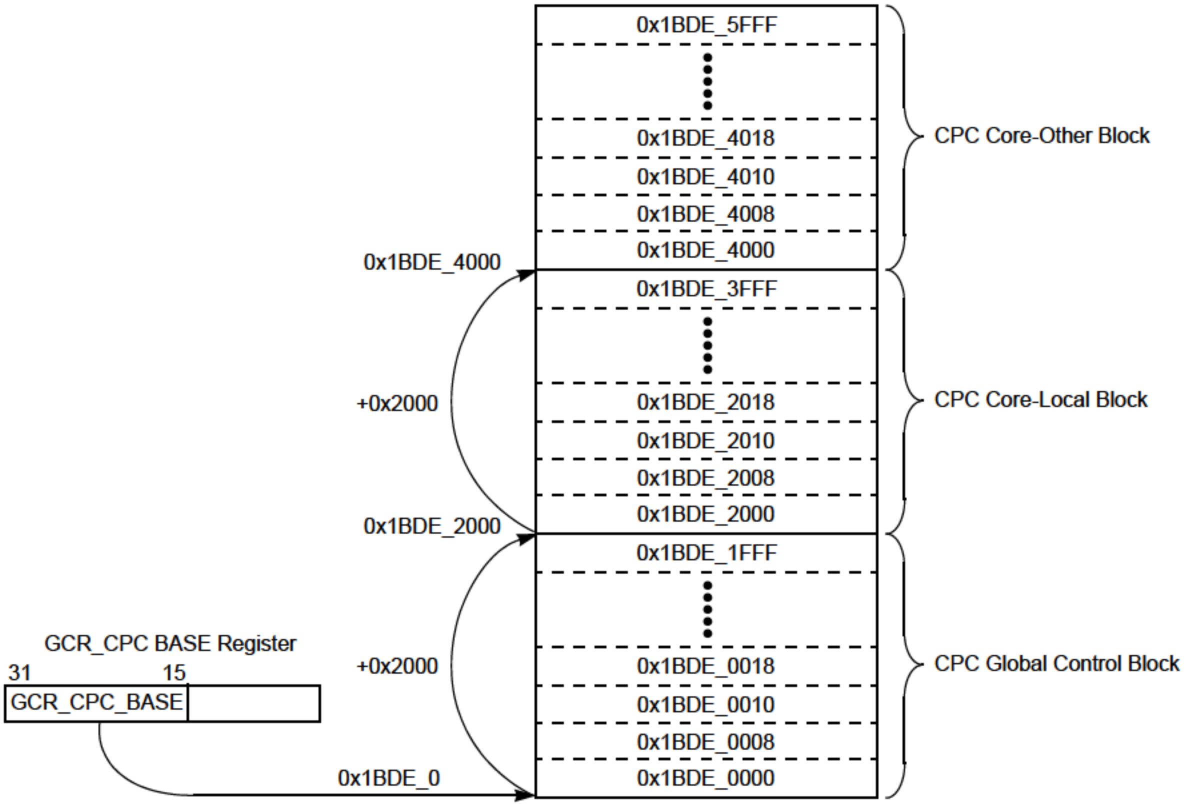
Table 7.6 shows the absolute physical addresses for the CPC Core-Other block. In this table an example base address of 0x1BDE_0 is used.

Table 7.7 Absolute Address of Individual CPC Core-Other Block Registers

MIPS Default Base		Core-Other Register Block Offset		Core-Other Register Offset		Absolute Physical Address	Core-Other Register
0x1BDE_0	+	0x4000	+	0x0000	=	0x1BDE_4000	CPC Core-Other Command.
0x1BDE_0	+	0x4000	+	0x0008	=	0x1BDE_4008	CPC Core-Other Status and Configuration.
0x1BDE_0	+	0x4000	+	0x0010	=	0x1BDE_4010	CPC Core-Other Addressing.

This concept is described in Figure 7.4 below. In this figure an example base address of 0x1BDE_0 is used.

Figure 7.4 CPC Register Addressing Scheme Using an Example Base Address of 0x1BDE_0



7.3.3 Global Control Block Register Map

All registers in the Global Control Block are 32 bits wide and should only be accessed using aligned 32-bit uncached load/stores. Reads from unpopulated registers in the CPC address space return 0x0, and writes to those locations are silently dropped without generating any exceptions.

Table 7.8 Global Control Block Register Map (Relative to Global Control Block offset)

Register Offset in Block	Name	Type	Description
0x000	CPC Global CSR Access Privilege Register (*CPC_ACCESS_REG*)	R/W	Controls which cores can modify the CPC Registers.
0x008	CPC Global Sequence Delay Counter (*CPC_SEQDEL_REG*)	R/W	Time between microsteps of a CPC domain sequencer in CPC clock cycles.
0x010	CPC Global Rail Delay Counter Register (*CPC_RAIL_REG*)	R/W	Rail power-up timer to delay CPS sequencer progress until the gated rail has stabilized.
0x018	CPC Global Reset Width Counter Register (*CPC_RESETLEN_REG*)	R/W	Duration of any domain reset sequence.
0x020	CPC Global Revision Register (*CPC_REVISION_REG*)	R	RTL Revision of CPC
0x028 0x0F8	CPC Global RESERVED registers.	-	For Future Extensions

7.3.3.1 Global CSR Access Privilege Register

Table 7.9 CPC Global CSR Access Privilege Register (CPC_ACCESS_REG Offset 0x000)

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
RESERVED	31:8	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0
CM_ACCESS_EN	7:0	Each bit in this field represents a power domain CPU. If the bit is set, that requester is able to write to the CPC registers (this includes all registers within the Global, Core-Local and Core-Other blocks. If the bit is clear, any write request from that requestor to the CPC registers (Global, Core-Local, Core-Other) will be dropped.	R/W	0xff

The Access privilege register configures the CPU access rights towards CPC programming registers. Its function is defined equally to the GCR Access Privilege Register. The CPU, as it is dedicated to power-management tasks of the cluster, should be granted full access to peer CPUs.

7.3.3.2 Global Sequence Delay Counter

The *CPC_SEQDEL_REG* describes globally the number of clock cycles each domain micro-sequencer will take to advance. It describes a set of worst-case timing of the physical implementation and is used to ensure electrical and bus protocol integrity. Mainly, buffer tree delays on *SI_Isolate* and/or *SI_RailEnable* can be used to set proper micro sequencer delay values.

Typically, the *CPC_SEQDEL_REG* contents would be defined at IP configuration time. However, runtime write capability allows fine tuning to optimize sequencer timing.

Table 7.10 Global Sequence Delay Counter Register (CPC_SEQDEL_REG, Offset 0x008)

Register Fields				
Name	**Bits**	**Description**	**Read/ Write**	**Reset State**
RESERVED	31:10	Read as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0
MICROSTEP	9:0	This field reflects the delay in clock cycles, taken by each power domain micro-sequencer to advance between atomic micro steps. Cycles/Step = MICROSTEP[9:0] value + 1; 0 => 1cycle, 1 => 2cycles... Physical implementation might not allow power sequence micro steps to advance with full cluster speed. At cluster cold start, the counter divides cluster frequency by a hardcoded IP configuration value to derive a micro step width.	R/W	IP Configuration Value

7.3.3.3 Global Rail Delay Counter

The *CPC_RAIL_REG* represents a 10-bit counter register to schedule delayed start of domain operation after the *RailEnable* signal has been activated by the CPC. This allows to compensate for slew rates at the gated rail, since hardware interlocks such as *SI_VddOk* are either unavailable or don't reflect to complete power up time of a domain.

At IP configuration time, the contents of *CPC_RAIL_REG* is preset. However, for fine tuning, the register can be written at run time.

Table 7.11 Global Rail Delay Counter Register (CPC_RAIL_REG, Offset 0x010)

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
RESERVED	31:10	Read as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0
RAILDELAY	9:0	10-bit counter value to delay power-up sequence per domain after *RailStable* and *VddOK* signals became active. The power-up micro-sequence starts after RAILDELAY has been loaded into the internal counter and a counted down to zero has concluded. After completion of the domain power-up micro-sequence, the *DomainReady* signal is raised and can be used for domain daisy-chaining.	R/W	IP Configuration Value

7.3.3.4 Global Reset Width Counter

Within the power-up micro-sequence, reset is applied. Typically, reset is active until the domain responds with *PB_Reset_N* feedback. However, the *CPC_RESETLEN_REG* allows reset to be extended beyond the *ResetN* feedback, or in case the reset feedback is unavailable. Counting down will start after the sequencer has received the *PB_Reset_N* feedback. Domains without *PB_ResetN* feedback could tie this input low or connect it to an inverted reset signal.

Table 7.12 Global Reset Width Counter Register (CPC_RESETLEN_REG, Offset 0x018)

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
RESERVED	31:10	Read as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0
RESETLEN	9:0	10-bit counter value to extend reset duration beyond *PB_Reset_N* feedback. The domain behavior after reset is determined by the domain local setup register.	R/W	IP Configuration Value

7.3.3.5 Revision Register

Table 7.13 Revision Register (CPC_Revision_REG, Offset 0x020)

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
RESERVED	31:16	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	
MAJOR_REV	15:8	This field reflects the major revision of the CPC block. A major revision might reflect the changes from one product generation to another.	R	Preset
MINOR_REV	7:0	This field reflects the minor revision of the CPC block. A minor revision might reflect the changes from one release to another.	R	Preset

7.3.4 Local and Core-Other Control Blocks

All registers in the CPC Local Control Block are 32 bits wide and should only be accessed using aligned 32-bit uncached load/stores. Reads from unpopulated registers in the CPC address space return 0x0, and writes to those locations are silently dropped without generating any exceptions.

A set of these registers exists for each proAptiv Multiprocessing System core in the proAptiv CPS. These registers can also be accessed from other cores by first writing the CPC *Core Other Addressing Register* (in the Core-Local Control Block) with the proper CoreNum and then accessing these registers using the Core Other address space.

The register offsets shown are relative to the offsets listed in Table .

Table 7.14 Core-Local Block Register Map

Register Offset in Block	Name	Type	Description
0x000	CPC Local Command Register (*CPC_CL_CMD_REG*)	R/W	Places a new CPC domain state command into this individual domain sequencer. This register is not available within the CM sequencer. Writes to the CM CMD register are ignored while reads will return zero.

Table 7.14 Core-Local Block Register Map *(continued)*

Register Offset in Block	Name	Type	Description
0x008	CPC Local Status and Configuration register (*CPC_CL_STAT_CONF_REG*)	R/W	Individual domain power status and domain configuration register. Reflects domain micro-sequencer execution. Initiates micro-sequencer after status register programming. Reflects command execution status.
0x010	CPC Core Other Addressing Register (*CPC_CL_OTHER_REG*)	R/W R/O for CM	Used to access local registers of another core.
0x018 0x0F8	CPC Local RESERVED registers	-	For Future Extensions

The register offsets shown are relative to the offsets listed in Table .

Table 7.15 Core-Other Block Register Map

Register Offset in Block	Name	Type	Description
0x000	CPC Local Command Register (*CPC_CO_CMD_REG*)	R/W	Places a new CPC domain state command into this individual domain sequencer. This register is not available within the CM sequencer. Writes to the CM CMD register are ignored while reads will return zero.
0x008	CPC Local Status and Configuration register (*CPC_CO_STAT_CONF_REG*)	R/W	Individual domain power status and domain configuration register. Reflects domain micro-sequencer execution. Initiates micro-sequencer after status register programming. Reflects command execution status.
0x010	CPC Core Other Addressing Register (*CPC_CO_OTHER_REG*)	R/W R/O for CM	Used to access local registers of another core.
0x018 0x0F8	CPC Local RESERVED registers	-	For Future Extensions

CPC Local register are used to set power-down conditions. After setup of conditions, the micro-sequencer can be activated through the command register. The execution of the micro-sequencer can be observed via the status register. Reading the command register retrieves the last executed command and status flags to reflect on recent commands given.

7.3.4.1 Command Register

Table 7.16 Local Command Register (CPC_CL[CO]_CMD_REG, Offset 0x000)

<table>
<tr><th colspan="2">Register Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/
Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>RESERVED</td><td>31:4</td><td>Read as 0x0. Writes ignored. Must be written with a value of 0x0.</td><td>R</td><td>0</td></tr>
<tr><td>CMD</td><td>3:0</td><td>Requests a new power sequence execution for this domain. Read value is the last executed command.
<table>
<tr><th>Code</th><th>Meaning</th></tr>
<tr><td>4’d1</td><td><b>ClockOff</b> -
This command causes the domain to cycle into clock-off mode.It disables the clock to this power domain. Only successful if <i>SI_CoherenceEnable</i> and other protocol interlocks are observed. If not, the command remains inactive until the protocol barriers subside. After that, the command is executed.
Dependent of current sequencer state, the command either causes power-up of a domain, or a domain leaves active duty to become inactive. A power-up leads to sequencer state U2, which will require the execution of a subsequent Reset or PwrUp command to make this domain operational.</td></tr>
<tr><td>4’d2</td><td><b>PwrDown</b> -
this domain using setup values in CPC_STAT_CONF_REG. Only successful if <i>SI_CoherenceEnable</i> inactive and all protocol interlocks are observed. If not, the command remains inactive until the protocol barriers subside. Then, the command is executed.</td></tr>
<tr><td>4’d3</td><td><b>PwrUp</b> -
this domain using setup values in CPC_STAT_CONF_REG. Usable only for Core-Others access. It is the software equivalent to <i>SI_PwrUp</i> hardware signal</td></tr>
<tr><td>4’d4</td><td><b>Reset</b> -
This domain is reset if in non-coherent mode. After the domain has been reset, the domain becomes operational and the CMD field reads as PwrUp cmd.</td></tr>
<tr><td>Others</td><td><b>Reserved</b></td></tr>
</table>
</td><td>R/W
Not available in CM domain</td><td>0</td></tr>
</table>

Table 7.17 Local Status and Configuration Register (CPC_CL[CO]_STAT_CONF_REG, Offset 0x008)

<table>
<tr><th colspan="2">Register Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/
Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>RESERVED</td><td>[31:24]</td><td>Reserved.</td><td>R</td><td>0</td></tr>
<tr><td>PWRUP_EVENT</td><td>23</td><td>The *SI_PowerUp* pin had been activated and caused the sequencer to cycle into power up state. The event also caused the sequencer to place a PwrUp command into the CMD field. Writing a 0 into the PWRUP_EVENT field will clear this bit.</td><td>R/W0</td><td>0</td></tr>
<tr><td>SEQ_STATE</td><td>[22:19]</td><td>Current domain sequencer state. State description:
<table>
<tr><th>Code</th><th>State</th></tr>
<tr><td>4’h0</td><td>D0 - PwrDwn</td></tr>
<tr><td>4’h1</td><td>U0 - VddOK</td></tr>
<tr><td>4’h2</td><td>U1 - UpDelay</td></tr>
<tr><td>4’h3</td><td>U2 - UClkOff</td></tr>
<tr><td>4’h4</td><td>U3 - Reset</td></tr>
<tr><td>4’h5</td><td>U4 - ResetDly</td></tr>
<tr><td>4’h6</td><td>U5 - nonCoherent execution</td></tr>
<tr><td>4’h7</td><td>U6 - Coherent execution</td></tr>
<tr><td>4’h8</td><td>D1 - Isolate</td></tr>
<tr><td>4’h9</td><td>D3 - ClrBus</td></tr>
<tr><td>4’ha</td><td>D2 - DClkOff</td></tr>
</table>
</td><td>R</td><td>0</td></tr>
<tr><td>RESERVED</td><td>18</td><td>Reserved.</td><td>R</td><td>-</td></tr>
<tr><td>CLKGAT_IMPL</td><td>17</td><td>If set, this domain is implemented with clock tree root gating.
If cleared, the CPC will still execute power-down/clock-off sequences if commanded; however, no physical clock gating is performed.</td><td>R</td><td>IP Configuration Value</td></tr>
<tr><td>PWRDN_IMPL</td><td>16</td><td>If set, this domain is implemented as power-gated.
If cleared, the CPC will still execute power-down sequences if commanded; however, no physical power switching is performed.</td><td>R</td><td>IP Configuration Value</td></tr>
<tr><td>EJTAG_PROBE</td><td>15</td><td>An EJTAG probe connection event has been seen. The domain powers up if required and observes a reset sequence. Thereafter the core transitions into clock-off mode. After a probe has been seen once, the power domain will not assume power-off mode until this bit is written to zero or the CPC experiences a cold reset.</td><td>R/W0</td><td>0</td></tr>
<tr><td>Reserved</td><td>14:11</td><td>Reserved.</td><td>R</td><td>0</td></tr>
<tr><td>Reserved</td><td>10</td><td>Reserved.</td><td>R/W</td><td>1</td></tr>
</table>

Table 7.17 Local Status and Configuration Register (CPC_CL[CO]_STAT_CONF_REG, Offset 0x008)

<table>
<tr><th colspan="2">Register Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/
Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>PWUP_POLICY</td><td>[9:8]</td><td>Each CPC domain sequencer is hardwired through the SI_ColdPwrUp signal to either power up, remain power-gated, go into clock-off mode, or become operational. To influence the cold start behavior of the domain, three distinct policies can be wired for this domain:
<table>
<tr><th>Code</th><th>Meaning</th></tr>
<tr><td>2’b00</td><td>This CPU remains powered down after a system cold start. A later PwrUp or Reset command, or SI_PwrUp signal assertion will make this domain operational.</td></tr>
<tr><td>2’b01</td><td>Go into Clock-Off mode. Disables domain clock after power-up sequence. Core will wake up through a CPC PwrUp or Reset command or a SI_PwrUp signal assertion.
In this Clock-Off mode, the core will not be initialized and its boundary isolation will be maintained.</td></tr>
<tr><td>2’b10</td><td>Power up this domain after system cold start. The CPU will be reset and become operational based on its boot vector contents.</td></tr>
<tr><td>2’b11</td><td>Reserved</td></tr>
</table>
Within a processor cluster, CPU zero would power-up, while peer CPU 1-3 remain unpowered until released through a PwrUp commands. The PWUP_POLICY field reflects the hard-wired SI_ColdPwrUp bus.</td><td>R</td><td>Hardwired IP Configuration Value
CM domain is hard coded to powerUp if any CPU domain is powered up initially.</td></tr>
<tr><td>RESERVED</td><td>[7:5]</td><td>Reads zero. Writes ignored</td><td>R</td><td>0</td></tr>
<tr><td>IO_TRFFC_EN</td><td>[4]</td><td>Enable CM for stand alone IOCU traffic. Setting this bit changes the low power state of the CM power domain from PwrDwn to ClkOff. The CM_IOPwrUp signal can be used by an external device to enable the CM to perform IOCU data transfers without CPU activities.
Deselecting IO_TRFFC_EN will power down the CM if all CPUs are powered down. In this case, CM_IOPwrUp signal activity is not observed by the CPC.
A powered down CM domain will clear all preset CM/IOCU control registers. Powering up due to CPU power-up will send the CM/IOCU through a reset sequence, together with the CPU.</td><td>R/O for CPUs, read zero
R/W for CM</td><td>0</td></tr>
<tr><td>CMD</td><td>3:0</td><td>Reflects most recent placed sequencer command. See definition in CPC_CMD_REG Table 7.3.4.1. The sequencer will overwrite the field after a Reset command, or SI_PwrUp signal caused power up of the domain. The command reads then as PwrUp.</td><td>R</td><td>0</td></tr>
</table>

7.3.4.2 Core-Other Addressing Register

This register must be written with the correct CoreNum value before accessing the Core-Other address segment. This register is not available within the CM local domain. Read access to the CM *CPC_OTHER_REG* will yield zero. Writes are ignored.

Table 7.18 Core-Other Addressing Register (CPC_CL[CO]_OTHER_REG Offset 0x010)

Register Fields		Description	Read/ Write	Reset State
Name	**Bits**			
RESERVED	31:24	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.-	R	0
CORENUM	23:16	CoreNum of the register set to be accessed in the Core-Other address space.	R/W	0x0
RESERVED	15:0	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.-	R	0

7.4 Cluster Power Controller Commands

The CPC provides a set of commands to establish a desired power domain state. CPC commands are:

- **ClockOff** - a power domain is brought into ClockOff state as programmed into the *CPC_CMD_REG* Table 7.3.4.1. If the domain was powered down before, the power-on sequence is applied according to *CPC_STAT_CONF_REG* settings. If the domain was active before and was in non-coherent operation, the domain is brought into ClockOff state D2. A domain in ClockOff state can be sent into operation using the PwrUp command. A ClockOff command given to a domain in coherent operation will remain inactive until the CPU has left the coherent mode of operation. Sending a ClkOff command to the CPC before a previous command completed will cause the CPC domain target to be redirected towards ClockOff. However, the previous steady state can be observed temporarily before the newly programmed state is reached.

- **PwrDwn** - a power domain is powered down into state D0. *CPC_STAT_CONF_REG* and *CPC_CMD_REG* settings determine the sequence observed by the CPC. Note, both register settings are observed dynamically. The sequencer will preempt an in flight command at the next steady state to execute the newly given command.

- **PwrUp** - the execution of this command depends on the previous domain power state. If the domain is powered down to state D0, a PwrUp command will enable power for the domain and bring the domain into operational state U5. However, if *SI_CoherenceEnable* is active, the domain will advance into state U6 - coherent operation. Please note, that a set of software initialization needs to complete to safely bring a non-coherent core into coherent state. If the previous power domain state was ‘ClkOff’, a PwrUp command will raise the domain state to either non-coherent or coherent operation, dependent on the GCR coherence status settings. This will be domain state U5 and U6 respectively.

  When bringing a domain up after a PwrDwn command is executed, the Reset command is generally preferable to PwrUp. If the domain did not reach state D0 or was prevented from entering D0 because an EJTAG probe was connected, the CPC may identify that a reset is not required for PwrUp and will simply restart the clocks. This may be fine, but also may cause some problems. One common example where a reset is required is if the core enters an infinite loop after requesting PwrDwn.

  A PwrUp command given to an active domain in non-coherent or coherent operation U5/U6 has no effect.

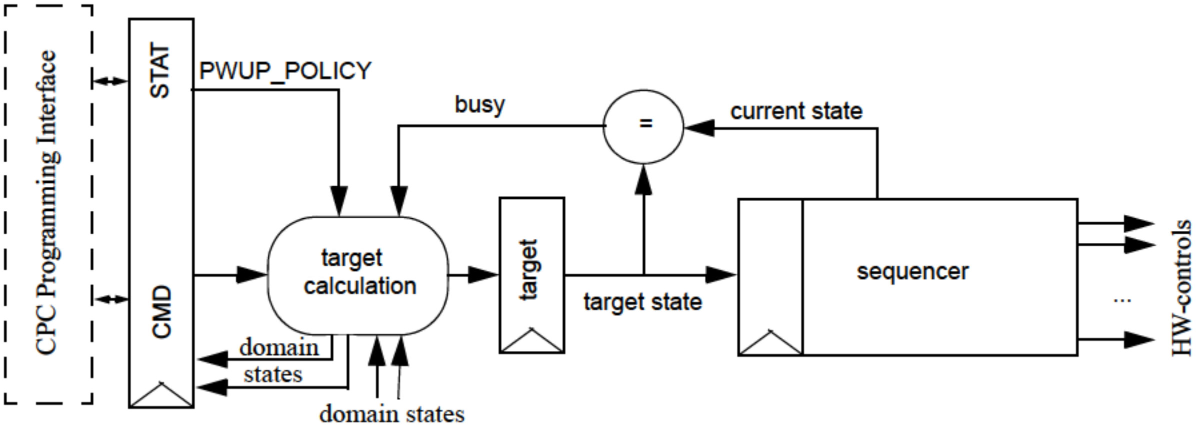
If a PwrUp command is given to the CPC while a previous command is still in flight, the command is placed in the CPC command register and is executed at the earliest possible state, i.e., when the sequencer has reached a non-transitional state.

The hardware *SI_PwrUp* signal activated for this domain will always bring the core into power-up mode with enabled clocks. The *PWUP_POLICY* settings of *CPC_STAT_CONF_REG* have no effect on hardware wake-ups. Also, the hardware wake-up has priority over software commands.The *PWRUP_EVENT* bit of *CPC_STAT_CONF_REG* is set after a hardware power-up has been executed.

- **Reset -** this command allows a domain in non-coherent operation (state U5) to be reset. It also can be sent to a domain in power-down or clock-off mode. The domain will then become active, and a reset sequence is executed which leads to an operational steady state of the domain (U5 or U6, dependent on GCR programming).

Figure 7.5 details the CPC domain command execution. A command given to a CPC power domain will be translated into a domain target state, and the domain sequencer will progress towards this target. A new command is accepted as soon as a suitable state transition is found within the traversed states. Domain sequencer states translate directly to hardware control signals for reset and power gating, as depicted in Figure 7.5.

Figure 7.5 CPC Command Execution



7.5 proAptiv Core Power Management Options

In addition to the Cluster Power Controller described in the previous sections, MIPS Technologies provides a mechanism for reducing power in the proAptiv core depending on the work load. The conditions under which the proAptiv core is placed in power-down mode are determined by the SOC.

The information in the following sections should be used only when all cores in the system are shut down. The processor and cache states need not be saved for each core shut down as long as their is one core operation. However, once the last core is to be shutdown by the SOC, the following procedure can be used to save the processor state.

There are two basic options for power management in the proAptiv core.

1. Clock gating: Used to stop the clocks and put the core into sleep mode. Refer to Section 7.6, "proAptiv Core Clock Gating" for more information. In this mode the VDD levels are maintained and power is preserved, so no data is lost.

2. Power gating: Used to shut down power to selected parts of the proAptiv core. In this mode certain elements of the core, such as registers, caches, TLB, etc. are saved, allowing for a more efficient power-up process. Refer to Section 7.7, "proAptiv Core Power Gating" for more information.

7.6 proAptiv Core Clock Gating

Clock gating provides a way for the proAptiv core to shut down the core clock under certain conditions. The mechanism used to suspend and then resume the core clock depends on the power management options selected during the core configuration process. These options include;

- Enabling of ‘top level clock gating’

- Enabling of ‘fine grain clock gating’

7.6.1 Designs Implementing Top Level Clock Gating

Top level clock gating is provided as an option during the core configuration process. For designs implementing top level clock gating, there are two ways to place the proAptiv core into sleep mode.

- Instruction-controlled power management

- Register-controlled power management

7.6.1.1 Instruction Controlled Clock Gating

Execution of the WAIT instruction can be used to place the proAptiv core into sleep mode. When the WAIT instruction is executed during normal operation, the proAptiv core completes all outstanding operations, then freezes the pipeline and asserts the SI_SLEEP signal, indicating to external logic that the proAptiv core has entered sleep mode.

If top level clock gating is enabled, the processor turns off the internal clock to most of the proAptiv core automatically once SI_SLEEP is asserted. The clock is maintained only for a small amount of logic that waits for an interrupt intended to bring the processor out of sleep mode. In addition to the interrupt logic, the following signals also remain active in sleep mode;

- SI_INT[5:0]

- SI_NMI
- SI_RESET
- EJ_DEBUGM

Once the clocks are suspended, the entire contents of the processor, including registers, caches, and TLB, are saved. Once the ‘wake’ interrupt is received, the processor restarts its internal clock and can resume normal operation within a few clock cycles. The ‘wake’ interrupt can be any enabled interrupt, NMI, or debug interrupt. This is the fastest and most efficient mechanism to transition the proAptiv core in and out of sleep mode.

Note that the SI_RESET signal can also be used to exit sleep mode. However, assertion of SI_RESET causes all internal data to be lost and the registers to revert back to their default values.

7.6.1.2 Register Controlled Clock Gating

In addition to instruction controlled clock gating, the MIPS architecture allows for software to initiate entry into sleep mode via the register interface. The *RP* bit in the *CP0 Status* register can be set by software to indicate the desire to place the proAptiv core into sleep mode. Once this bit is set, hardware asserts the *SI_RP* output signal.

On receipt of the *SI_RP* signal, external logic can then decide whether to suspend or reduce the frequency of the proAptiv core accordingly. Note that this mechanism is different than instruction controlled clock gating in that the core does not determine whether the clock is suspended. Rather, external logic can decide to suspend the clock, reduce the clock frequency from its current level, or take no action.

Table 7.19 Differences Between Instruction and Register Controlled Power Management

Type	Trigger	SIgnal Asserted by Hardware	Clock Suspended by On-Die Hardware	Interrupt Detection During Sleep Mode
Instruction controlled clock gating	WAIT instruction	SI_SLEEP	Yes	Yes
Register controlled power management	Setting RP bit in CP0 Status	SI_RP	No	Yes

7.6.1.3 Reduction of VDD During Sleep Mode

The information described above deals with clock gating only. In this example, during the time that the clocks are powered down, VDD remains at normal power levels. To obtain the maximum power savings during sleep mode, external logic can reduce the core VDD voltage once the proAptiv core has asserted SI_SLEEP. This additional step can greatly reduce leakage and consequently power consumption during sleep mode. The minimum VDD voltage that can be used, and still allow the proAptiv core to retain state, is process dependent.

The reduction of VDD can only be controlled by external means. The proAptiv core does not provide a mechanism to reduce VDD internally during sleep mode. Note that if this option is implemented, it will take longer to restart the processor since the VDD must be ramped up to appropriate level before asserting the wake interrupt.

Refer to Section 7.7 “proAptiv Core Power Gating” for more information.

7.6.1.4 Restart Latency Trade-Offs

Once the decision is made to enter sleep mode, some number of clocks are required to place the proAptiv core into sleep mode, and bring the core out of sleep mode. In most designs, once sleep mode is entered, the core must remain in sleep mode for at least 100 clock cycles. Otherwise, the trade-off in time and power savings becomes negligible.

7.6.2 Designs Not Implementing Top Level Clock Gating

If top level clock gating was not enabled during the core configuration process, both instruction and register controlled power management can still be used. The main difference is the level of involvement of the proAptiv core in either of these processes.

From an instruction standpoint, the WAIT instruction and SI_SLEEP signal can still be used to place the proAptiv core into sleep mode. However, since top level clock gating is disabled, it is incumbent upon external logic to suspend the input clock to the processor. If the input clock is suspended, it is suspended to the entire proAptiv core. As a result, the processor has no way to detect a 'wake' interrupt. Therefore, the assertion of SI_RESET is the only way to restart the proAptiv core. Note that if this method is used, all data will be lost and the registers will revert back to their default values.

From a register standpoint, software can still set the RP bit in the CP0 Status register to initiate the transfer to sleep mode. The processor responds by asserting the SI_RP bit to external logic. At this point, the proAptiv core does not control the clock behavior. It is incumbent upon external logic to provide the following functions:

- Suspend the core clock
- Reduce the core clock frequency
- Implement the interrupt detect function

7.6.3 Designs Implementing Fine Grain Clock Gating

Fine grain clock gating allows the proAptiv core to shut down the clocks to individual blocks of logic within the chip. When the 'fine grain clock gating' option is selected during build time, separate clock domains are assigned to the various register blocks within the proAptiv core. In the proAptiv core, there is one write enable that is used to write all registers at once. If fine grain clock gating is enabled, the clock can be enabled only to the register block that is being accessed. The write enable for the other blocks is still driven, but no clock is supplied to those blocks not being accessed.

The implementation of fine grain clock gating requires the logic required to implement multiple clock trees within the proAptiv core. Therefore, it works best in ASIC implementations where any number of clock domains can be assigned. It is less useful in FPGA implementations where the number of clock trees may be limited.

7.7 proAptiv Core Power Gating

In addition to clock gating, power gating can be used to gain additional power savings. The saving and restoring of processor state can be used when the power savings provided by clock gating alone are not enough. In clock gating, the state of the processor need not be saved externally because even though the clocks are suspended, the power is still applied to the proAptiv core, allowing the processor state to be saved internally.

In power gating, some or all of the power to the proAptiv core can be shut down. This causes all data within the corresponding power domain(s) to be lost once the voltage falls below the retention value as defined by the process vendor. As a result, careful consideration must be taken to save some or all of the processor states before the power is shut down. Some of the logic blocks that can be saved prior to suspending the processor are:

- Registers (GPR, CP0, CP1, and/or CP2)
- Caches (instruction and/or data)
- Translation Lookaside Buffer (TLB)
- Scratch Pad RAM (Instruction and/or Data)

There are two methods that can be used to implement a suspend/resume mechanism in a proAptiv core. These concepts are described in the following subsections.

- Hardware Suspend/Resume
- Software Suspend/Resume

7.7.1 Hardware Suspend/Resume

The hardware suspend/resume mechanism in the proAptiv core allows the state of the caches, scratch pad RAM, and TLB to be transferred to memory via hardware using the suspend/resume (BIST) sideband signals that are defined during chip configuration. This process of moving data to and from the proAptiv core is much faster than a pure software implementation. This process is covered in more detail in the proAptiv *Hardware User's Manual*.

7.7.2 Software Suspend/Resume

For systems that have not implemented any hardware suspend/resume mechanism as described in the previous section, a software mechanism can be used to save state and power down the proAptiv core. This section describes the tasks that should be performed during the suspend and resume processes.

7.7.2.1 Overview of Suspend/Resume Process

The recommended way of implementing a system suspend/resume in software is having a function that will perform a seamless suspend/resume operation. This means that to the rest of the software it looks like the function was entered and exited like any normal function, while in reality this function self-terminates in the middle of its execution by turning off the power the core, then resumes from where it left off shortly after power is restored.

At a high level, the assembly language skeleton should look like this:

```
/* Entry point to suspend/resume function, including the function prologue. */

suspend_resume:
...
...

/* Here we start the suspend sequence */

suspend:
...
...
...
```

```
/* At the end of the suspend sequence we turn off power to the core. The suspend sequence should never reach the
power_is_off label*/

power_is_off:

/* This is the starting point of the resume sequence. We will get here shortly after a warm reset.*/

resume:

...
...
...

/* At the end of the resume sequence we have the function epilogue, which includes a return to the calling function.*/

...
...
...

        jr      $31
        nop
```

As one can observe this function is clearly divided into two parts:

- The first part is the function entry (prologue) and the suspend sequence all the way down to the power shutdown. The suspend sequence includes the state saving and other supporting actions which are described in more details in the other sections.

- The second part is the resume sequence followed by the function exit (epilogue) and return to caller. The resume sequence includes state restoring and other actions which are described in more details in other sections.

If we look at the sequence of events on a time line it will look like this:

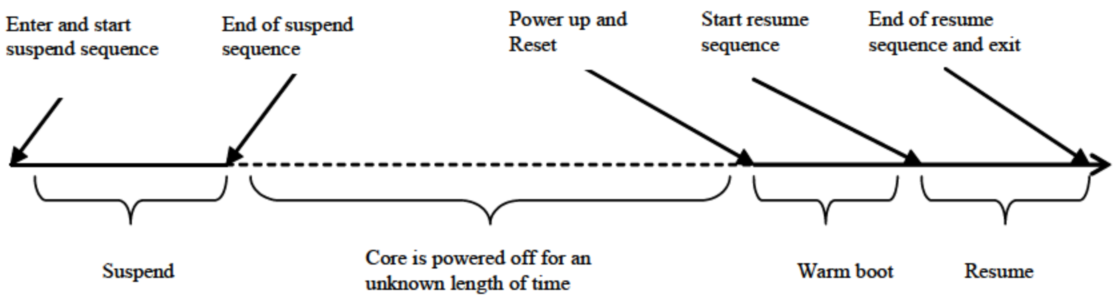

Enter and start suspend sequence
End of suspend sequence
Power up and Reset
Start resume sequence
End of resume sequence and exit
Suspend
Core is powered off for an unknown length of time
Warm boot
Resume

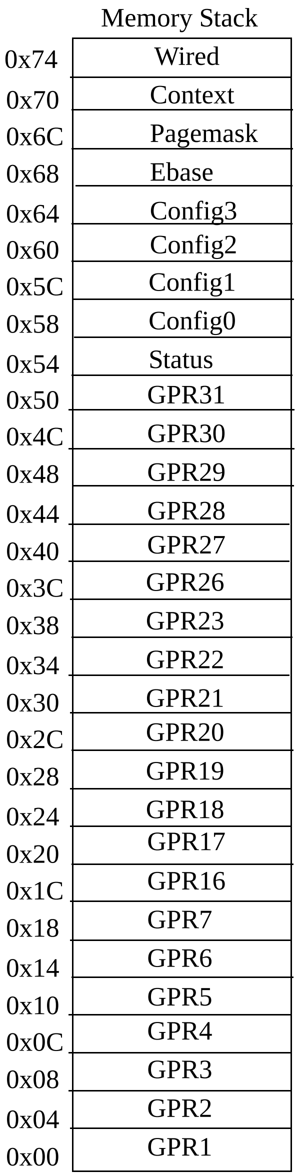
Figure 7.6 Suspend/Resume Sequence Time Line

7.7.3 Suspend Process

During a software suspend process, the following tasks are recommended. Each of these tasks is described in the following subsections.

- Save General Purpose Registers (GPR)
- Save some or all CP0 registers
- Flush the L1 data cache dirty lines and L2 cache dirty lines (if applicable)
- Save the return address
- Copy memory power down sequence into cache before switching memory to low-power mode (if applicable)
- Move memory to low-power mode (if applicable)
- Shut down power to the proAptiv core

The GPR and CP0 registers are moved to the memory stack prior so that they can be easily retrieved when power is restored to the proAptiv core. In this example, the registers would be moved to the stack and placed at the following memory offset addresses shown in Figure 7.7.



Memory Stack

0x74	Wired
0x70	Context
0x6C	Pagemask
0x68	Ebase
0x64	Config3
0x60	Config2
0x5C	Config1
0x58	Config0
0x54	Status
0x50	GPR31
0x4C	GPR30
0x48	GPR29
0x44	GPR28
0x40	GPR27
0x3C	GPR26
0x38	GPR23
0x34	GPR22
0x30	GPR21
0x2C	GPR20
0x28	GPR19
0x24	GPR18
0x20	GPR17
0x1C	GPR16
0x18	GPR7
0x14	GPR6
0x10	GPR5
0x0C	GPR4
0x08	GPR3
0x04	GPR2
0x00	GPR1

Figure 7.7 GPR and CP0 Register Locations in the Memory Stack

7.7.3.1 Save GPR Registers

MIPS recommends saving those GPR registers shown in the code example below. Note that the register numbers corresponding to the scratch registers are not saved. This includes GPR8 - GPR15, GPR24, and GPR25. For each GPR, a store word (*sw*) instruction is used to move the contents of the GPR register to memory.

```
sw      $1      0x00(sp)
sw      $2      0x04(sp)
sw      $3      0x08(sp)
sw      $4      0x0C(sp)
sw      $5      0x10(sp)
sw      $6      0x14(sp)
sw      $7      0x18(sp)
sw      $16     0x1C(sp)
sw      $17     0x20(sp)
sw      $18     0x24(sp)
sw      $19     0x28(sp)
sw      $20     0x2C(sp)
sw      $21     0x30(sp)
sw      $22     0x34(sp)
sw      $23     0x38(sp)
sw      $26     0x3C(sp)
sw      $27     0x40(sp)
sw      $28     0x44(sp)
sw      $29     0x48(sp)
sw      $30     0x4C(sp)
sw      $31     0x50(sp)
```

7.7.3.2 Save CP0 Registers

In the MIPS architecture the CP0 registers cannot be moved directly to memory. Therefore, they must first be moved to a GPR register. In this example the registers are moved to the k0 scratch pad register, then from the k0 register to memory at the location shown in the corresponding *sw* instruction. Note that the offset addresses for each *sw* instruction correspond to those shown in Figure 7.7.

As shown in the code snippet below, only a partial set of CP0 registers are saved. This is only an example. In some cases additional registers may need to be saved depending on the implementation.

```
mfco    k0,     CP0_STATUS          /*Move from coprocessor 0, CP0_STATUS to k0*/
sw      k0,     0x54(sp)            /*Store word k0 to offset 0x54 in memory*/
mfco    k0,     CP0_CONFIG0         /*Move from coprocessor 0, CP0_CONFIG0 to k0*/
sw      k0,     0x58(sp)            /*Store word k0 to offset 0x58 in memory*/
mfco    k0,     CP0_CONFIG1         /*Move from coprocessor 0, CP0_CONFIG1 to k0*/
sw      k0,     0x5C(sp)            /*Store word k0 to offset 0x5C in memory*/
mfco    k0,     CP0_CONFIG2         /*Move from coprocessor 0, CP0_CONFIG2 to k0*/
sw      k0,     0x60(sp)            /*Store word k0 to offset 0x60 in memory*/
mfco    k0,     CP0_CONFIG3         /*Move from coprocessor 0, CP0_CONFIG3 to k0*/
sw      k0,     0x64(sp)            /*Store word k0 to offset 0x64 in memory*/
mfco    k0,     CP0_EBASE           /*Move from coprocessor 0, CP0_EBASE to k0*/
sw      k0,     0x68(sp)            /*Store word k0 to offset 0x68 in memory*/
mfco    k0,     CP0_PAGEMASK        /*Move from coprocessor 0, CP0_PAGEMASK to k0*/
sw      k0,     0x6C(sp)            /*Store word k0 to offset 0x6C in memory*/
mfco    k0,     CP0_CONTEXT         /*Move from coprocessor 0, CP0_CONTEXT to k0*/
sw      k0,     0x70(sp)            /*Store word k0 to offset 0x70 in memory*/
mfco    k0,     CP0_WIRED           /*Move from coprocessor 0, CP0_WIRED to k0*/
sw      k0,     0x74(sp)            /*Store word k0 to offset 0x74 in memory*/
```

7.7.3.3 Flush Dirty Lines in L1 Data Cache

The following routine can be used to flush the dirty lines in a 32 Kbyte, 4-way set associative data cache with a 32-byte line size in preparation for shut-down. In this routine software examines each cache line and performs an invalidate on all non-dirty lines, and a writeback-invalidate on all dirty lines. A similar routine must be applied for L2 dirty lines in systems implementing a level 2 cache.

```
#define INDEX_BASE 0x80000000 // We use KSEG0 address as the base address for cache index access
#define WAY_SIZE 0x2000          // size of one way in a 4-way set associative 32K cache (8K)
#define WAYOFFSET 13             // offset of bits which determine the cache way to access
#define ASSOC   4                // associativity (4 ways)
#define LINE_SIZE 32             // size of each cache line
#define IDX_WB_INV_DC 0x01       // code of index write-back invalidate D-cache operation

/* This macro performs the same cache op on 32 consecutive lines. */

#define cache32_unroll32(base,op)              \

        __asm__ __volatile__(                               \
                ".set push                          \n"     \
                ".set noreorder                     \n"     \
                ".set mips3                         \n"     \
                "cache %1, 0x000(%0); cache %1, 0x020(%0)\n"     \
                "cache %1, 0x040(%0); cache %1, 0x060(%0)\n"     \

                "cache %1, 0x080(%0); cache %1, 0x0a0(%0)\n"     \
                "cache %1, 0x0c0(%0); cache %1, 0x0e0(%0)\n"     \
                "cache %1, 0x100(%0); cache %1, 0x120(%0)\n"     \
                "cache %1, 0x140(%0); cache %1, 0x160(%0)\n"     \|
                "cache %1, 0x180(%0); cache %1, 0x1a0(%0)\n"     \
                "cache %1, 0x1c0(%0); cache %1, 0x1e0(%0)\n"     \
                "cache %1, 0x200(%0); cache %1, 0x220(%0)\n"     \
                "cache %1, 0x240(%0); cache %1, 0x260(%0)\n"     \
                "cache %1, 0x280(%0); cache %1, 0x2a0(%0)\n"     \
                "cache %1, 0x2c0(%0); cache %1, 0x2e0(%0)\n"     \
                "cache %1, 0x300(%0); cache %1, 0x320(%0)\n"     \
                "cache %1, 0x340(%0); cache %1, 0x360(%0)\n"     \
                "cache %1, 0x380(%0); cache %1, 0x3a0(%0)\n"     \
                "cache %1, 0x3c0(%0); cache %1, 0x3e0(%0)\n"     \

                ".set pop                           \n"     \
                :                                           \
                : "r" (base),                               \
                  "i" (op));

/* This function scans a 4-way set associative 32K bytes data cache with 32-byte line size and performs an index
write-back invalidate cache operation on each of the cache lines.*/

static  void flush_32k_4way_32byteline_dcache(void)

{                                                                              \
        unsigned long start = INDEX_BASE;
        unsigned long end = start + WAY_SIZE;
        unsigned long ws_inc = 1UL << WAYOFFSET;
        unsigned long ws_end = ASSOC << WAYOFFSET;
        unsigned long ws, addr;
        /* For every way (ws = the bits in the address which dertmine the cache way to access). */
        for (ws = 0; ws < ws_end; ws += ws_inc)
                /* In each way go from start to end address. */
                for (addr = start; addr < end; addr += LINE_SIZE * 32)
                        /* Each time we perform the cache op on 32 lines. The address is a
                           combination of the cache line offset in side the way (addr) and the way bits (ws).*/
                        cache32_unroll32(addr|ws, IDX_WB_INV_DC);
```

7.7.3.4 Save the Resume Address

This routine takes the starting address of the resume sequence and saves it somewhere on the board, external to the proAptiv core. Later, after power up and reset, the warm boot sequence retrieves that address and jumps to it. This initiates execution of the resume process.

7.7.3.5 Copy Memory Power Down Sequence Into Cache

This piece of code loads the remaining instructions of the suspend sequence into the instruction cache. This is done since the memory (e.g. DRAM) is about to be put in low power mode and thus become inaccessible to the core. It is important that all instruction fetches hit in the instruction cache because if they miss the core won't be able to fetch them from memory.

```
*/

        .set noreorder

/* load the start address and end address of the remaining instructions */

        la      $8, mem_to_low_power
        la      $9, post_suspend        /*after power is removed*/

/* Now fill the cache line by line starting from the start address and incrementing the address by a line size in each
iteration until we get beyond the en address.*/

fill_icache:

        cache   0x14, 0($8)
        addiu   $8, $8, 32
        bltu    $8, $9, fill_icache
        nop

mem_to_low_power:
```

7.7.3.6 Move Memory to Low Power Mode

```
/* Here we have a sequence of instructions that will move the memory to low power mode. These instructions used to
perform this function are SOC specific depending on the particular way the memory is implemented and addressed.*/

...
...
...

/* The following label comes after the end of the suspend sequence. We should never get here because we are supposed to loose power earlier.*/
```

post_suspend:

7.7.3.7 Shut Down Power to the proAptiv core

Once all of the above tasks have been performed, power to the proAptiv core can be suspended by reducing VDD to 0V. This task is performed by the SOC and is implementation-dependent.

7.7.4 Resume Process

During the software resume process, the following tasks are recommended. The tasks are handled in the opposite order in which they were executed during the suspend operation.

- System Wake-up
- Power-Up VDD to the proAptiv core and Assert Power-On Reset
- Warm/Cold Boot Detection
- Exit memory low-power mode
- Initialize caches and TLB
- Jump to resume address
- Restore CP0 registers
- Restore GPR registers

7.7.4.1 System Wake-Up

In a typical system the power management (PM) module stays active after the system enters suspend mode. This component will consume very little power but will keep monitoring external signals that may trigger the system to resume normal operation. Once a trigger is detected, the PM block will wake up various system components, one of these being the proAptiv core. Since power to the core was shut down earlier, the core must be powered up and brought to its Reset state.

7.7.4.2 Power-Up VDD to the proAptiv Core and Assert Power-On Reset

Once the system logic detects a resume condition, the system power management block must raise the VDD levels of the proAptiv core to their normal operating levels and allow the voltage to stabilize. Once the voltages are stabilized, assert the power-on reset pin to the proAptiv core.

7.7.4.3 Warm/Cold Boot Detection

When a processor core goes to its reset state it starts executing instructions from its Reset vector address. We call the initial sequence of instructions "boot" and it typically starts executing off of "boot ROM" memory. At this point the system must distinguish between two boot modes: cold boot and warm boot.

- A cold boot is typically performed when the entire system is powered up and has to initialize all of its hardware components. In this scenario there is typically no (or little) memory of the system's state prior to boot (although some systems will save configuration information in non-volatile memory). After the initial boot the operating system has to go through its own complete boot sequence which takes a relatively long time.

- A warm boot is typically performed to resume a system that was previously suspended for power saving. In this case much of the system state prior to boot is available and can be restored (for example, it was saved into a memory component which did not loose power or otherwise in non-volatile memory). The warm boot sequence is typically short as users expect instant response (from a user point of view the system is available even when it was suspended for power saving). A warm boot does not require the operating system to perform its full boot sequence. For the most part the OS will continue from where it left off.

In the case of a warm boot, the boot software sequence starts from the same place (the Reset vector address) whether it is a cold boot or warm boot condition. However, shortly thereafter it detect its mode whether it is a cold or warm boot. If the system resumes from suspend mode, the boot software will detect this and decide to perform a warm boot. The indication that the system is coming back from suspend mode may be available in the PM block or in some piece of memory. This mechanism is implementation dependent.

Once a decision is made to perform a warm boot and not a cold boot, the warm boot sequence will perform a basic initialization and then jump to the resume address in the suspend/resume function. The resume address will be available in an implementation dependent location where it was saved by the suspend sequence. Then, as discussed earlier, the function will restore some system state and return to its caller as if nothing ever happened. The caller may have no indication that the system was suspended for a while.

Examples of basic core initialization that must be carried out regardless of the boot mode are caches and TLB initialization. Many users will opt not to save and restore their cache and/or TLB states. Note that the proAptiv core caches and TLB wake-up in a random state and must be initialized before data can be written to them.

7.7.4.4 Exit Memory Low-Power Mode

This is an optional system-dependent function. If the external memory devices were placed in low-power mode during the suspend process, the memory must exit its low-power mode before the instructions stored to the stack during the suspend process can be fetched by the proAptiv core.

7.7.4.5 Initialize Caches and TLB

The initialize caches and TLB routines are always performed when reset is asserted to the proAptiv core. This is done to bring the caches to an initial state. This routine would be exactly the same as the one used in the boot example that accompanies the delivery of each proAptiv core. Refer to the boot example associated with the proAptiv core package.

7.7.4.6 Jump to Resume Address

At this point the boot process is done with general initialization process initiated by the assertion of reset and is ready to start the actual resume sequence. It retrieves the starting address of the resume sequence that was saved earlier (as part of the suspend sequence) and jumps to it, thereby initiating execution of the resume sequence.

7.7.4.7 Restore CP0 Registers

In the MIPS architecture the CP0 registers cannot be moved directly from memory. Therefore, they must first be moved to a GPR register. In this example the registers are moved to the k0 scratch pad register, then from the k0 register to memory at the location shown in the corresponding *lw* instruction. Note that the offset addresses for each *lw* instruction correspond to those shown in Figure 7.7.

```
lw      k0,     0x74(sp)            /*Load word k0 from offset 0x74 in memory*/
mtco    k0,     CP0_WIRED           /*Move to coprocessor 0, CP0_WIRED from k0*/
lw      k0,     0x70(sp)            /*Load word k0 from offset 0x70 in memory*/
mtco    k0,     CP0_CONTEXT         /*Move to coprocessor 0, CP0_CONTEXT from k0*/
lw      k0,     0x6C(sp)            /*Load word k0 from offset 0x6C in memory*/
```

```
mtco    k0,    CP0_PAGEMASK     /*Move to coprocessor 0, CP0_PAGEMASK from k0*/
lw      k0,    0x68(sp)         /*Load word k0 from offset 0x68 in memory*/
mtco    k0,    CP0_EBASE        /*Move to coprocessor 0, CP0_EBASE from k0*/
lw      k0,    0x64(sp)         /*Load word k0 from offset 0x64 in memory*/
mfco    k0,    CP0_CONFIG3      /*Move to coprocessor 0, CP0_CONFIG3 from k0*/
lw      k0,    0x60(sp)         /*Load word k0 from offset 0x60 in memory*/
mtco    k0,    CP0_CONFIG2      /*Move to coprocessor 0, CP0_CONFIG2 from k0*/
lw      k0,    0x5C(sp)         /*Load word k0 from offset 0x5C in memory*/
mtco    k0,    CP0_CONFIG1      /*Move to coprocessor 0, CP0_CONFIG1 from k0*/
lw      k0,    0x58(sp)         /*Load word k0 from offset 0x58 in memory*/
mtco    k0,    CP0_CONFIG0      /*Move to coprocessor 0, CP0_CONFIG0 from k0*/
lw      k0,    0x54(sp)         /*Load word k0 from offset 0x54 in memory*/
mtco    k0,    CP0_STATUS       /*Move to coprocessor 0, CP0_STATUS from k0*/
```

7.7.4.8 Restore GPR Registers

MIPS recommends loading those GPR registers shown in the code example below. Note that the register numbers corresponding to the scratch pad registers are not loaded. This includes GPR8 - GPR15, GPR24, and GPR25. For each GPR, a load word (*lw*) instruction is used to move the contents of the corresponding memory location into the GPR.

```
lw      $31    0x50(sp)|
lw      $30    0x4C(sp)
lw      $29    0x48(sp)
lw      $28    0x44(sp)
lw      $27    0x40(sp)
lw      $26    0x3C(sp)
lw      $23    0x38(sp)
lw      $22    0x34(sp)
lw      $21    0x30(sp)
lw      $20    0x2C(sp)
lw      $19    0x28(sp)
lw      $18    0x24(sp)
lw      $17    0x20(sp)
lw      $16    0x1C(sp)
lw      $7     0x18(sp)
lw      $6     0x14(sp)
lw      $5     0x10(sp)
lw      $4     0x0C(sp)
lw      $3     0x08(sp)
lw      $2     0x04(sp)
lw      $1     0x00(sp)
```

Chapter 8

CM2 Global Control Registers

The proAptiv Global Control Registers address space (GCR) contains control/status registers for the entire proAptiv Multiprocessing System cluster (see Section 8.3 “Global Control Block”), as well as the individual proAptiv Multiprocessing System CPUs (see Section 8.4 “Core-Local and Core-Other Control Blocks”) in the cluster.

The GCR address space has a total size of 32 KBytes, which is divided into 8 KByte blocks as described in Section 8.1 “Coherence Manager Address Map”. The location of the GCR block in the system address map is controlled by the *GCR_BASE* register.

Physically, the registers are located within the GCR block of the Coherence Manager (CM2) and are accessed by the proAptiv Multiprocessing System CPUs using 32-bit aligned uncached load/store instructions, or by I/O devices via the I/O Coherence Unit (IOCU), using read/write instructions.

This chapter contains the following sections:

- Section 8.1 “Coherence Manager Address Map”
- Section 8.2 “CM2 Programming”
- Section 8.3 “Global Control Block”
- Section 8.4 “Core-Local and Core-Other Control Blocks”
- Section 8.5 “Global Debug Control Block”

8.1 Coherence Manager Address Map

Table 8.1 shows the address map of the four, 8-KB GCR blocks relative to the *GCR_BASE* as defined in the *GCR Base Register*. Each of these blocks of registers are described in the following sections.

Table 8.1 proAptiv Control Space Address Map (Relative to GCR_BASE[31:15])

Address Range	Size (bytes)	Description
0x0000 - 0x1FFF	8 KB	**Global Control Block.** Contains registers pertaining to the global system functionality. All cores can access this block of registers.
0x2000 - 0x3FFF	8 KB	**Core-Local Control Block** (aliased for each proAptiv Multiprocessing System CPU core). Contains registers pertaining to the proAptiv Multiprocessing System issuing the request. Each core has its own copy of registers within this block.
0x4000 - 0x5FFF	8 KB	**Core-Other Control Block** (aliased for each proAptiv Multiprocessing System CPU core). This block of addresses gives each Core a window into another CPU’s Core-Local Control Block. Before accessing this space, the *Core-Other_Addressing Register* in the Local Control Block must be set with the CORENum of the target Core.
0x6000 - 0x7FFF	8 KB	**Global Debug Block.** Contains global registers useful in debugging the proAptiv MPS.

8.1.1 Block Offsets Relative to the Base Address

The block offsets for each of the four blocks listed in Table 8.1 above are relative to a GCR base address and can be located anywhere in physical memory. The base address is a 17-bit value that is programmed into the GCR_BASE field of the *GCR Base* register located at offset address 0x0000 in the Global Control Block. The MIPS default location for the GCR_BASE address is 0x1FBF_8. To determine the physical address of each block using the MIPS default, this value would be added to the GCR block offset to derive the absolute physical address as shown in Table 8.2.

Table 8.2 Absolute Address of GCR Register Blocks Using the MIPS Default

MIPS Default Base		GCR Block Offset		Absolute Physical Address	Size (bytes)	Description
0x1FBF_8	+	0x0000 - 0x1FFF	=	0x1FBF_ 8000 - 0x1FBF_9FFF	8 KB	Global Control Block.
0x1FBF_8	+	0x2000 - 0x3FFF	=	0x1FBF_ A000 - 0x1FBF_BFFF	8 KB	Core-Local Control Block
0x1FBF_8	+	0x4000 - 0x5FFF	=	0x1FBF_ C000 - 0x1FBF_DFFF	8 KB	Core-Other Control Block
0x1FBF_8	+	0x6000 - 0x7FFF	=	0x1FBF_ E000 - 0x1FBF_FFFF	8 KB	Global Debug Block

8.1.2 Register Offsets Relative to the Block Offsets

In addition to the block offsets, the register offsets provided in each register description of this chapter are relative to the block offsets shown in Table 8.2 above. To determine the physical address of each register, the MIPS default base address is added to the corresponding GCR block offset plus the actual register offset to derive the absolute physical address as shown in Table 8.3. Note that this example shows only a few selected registers of the Global Control Block.

Table 8.3 Absolute Address of Individual Global Control Block Registers

MIPS Default Base		Global Register Block Offset		Global Register Offset		Absolute Physical Address	Global Control Register
0x1FBF_8	+	0x0000	+	0x0000	=	0x1FBF_ 8000	CM2 Configuration.
0x1FBF_8	+	0x0000	+	0x0008	=	0x1FBF_ 8008	GCR Base.
0x1FBF_8	+	0x0000	+	0x0010	=	0x1FBF_ 8010	CM2 Control.
0x1FBF_8	+	0x0000	+	0x0018	=	0x1FBF_ 8018	CM2 Control2.
0x1FBF_8	+	0x0000	+	0x0020	=	0x1FBF_ 8020	CM2 Access Privilege.
....		....		....		....	....
0x1FBF_8	+	0x0000	+	0x0228	=	0x1FBF_ 8228	Attribute-Only Region 3 Mask.

The registers within the Core-Local blocks would be accessed in a similar manner as shown in Table 8.4.

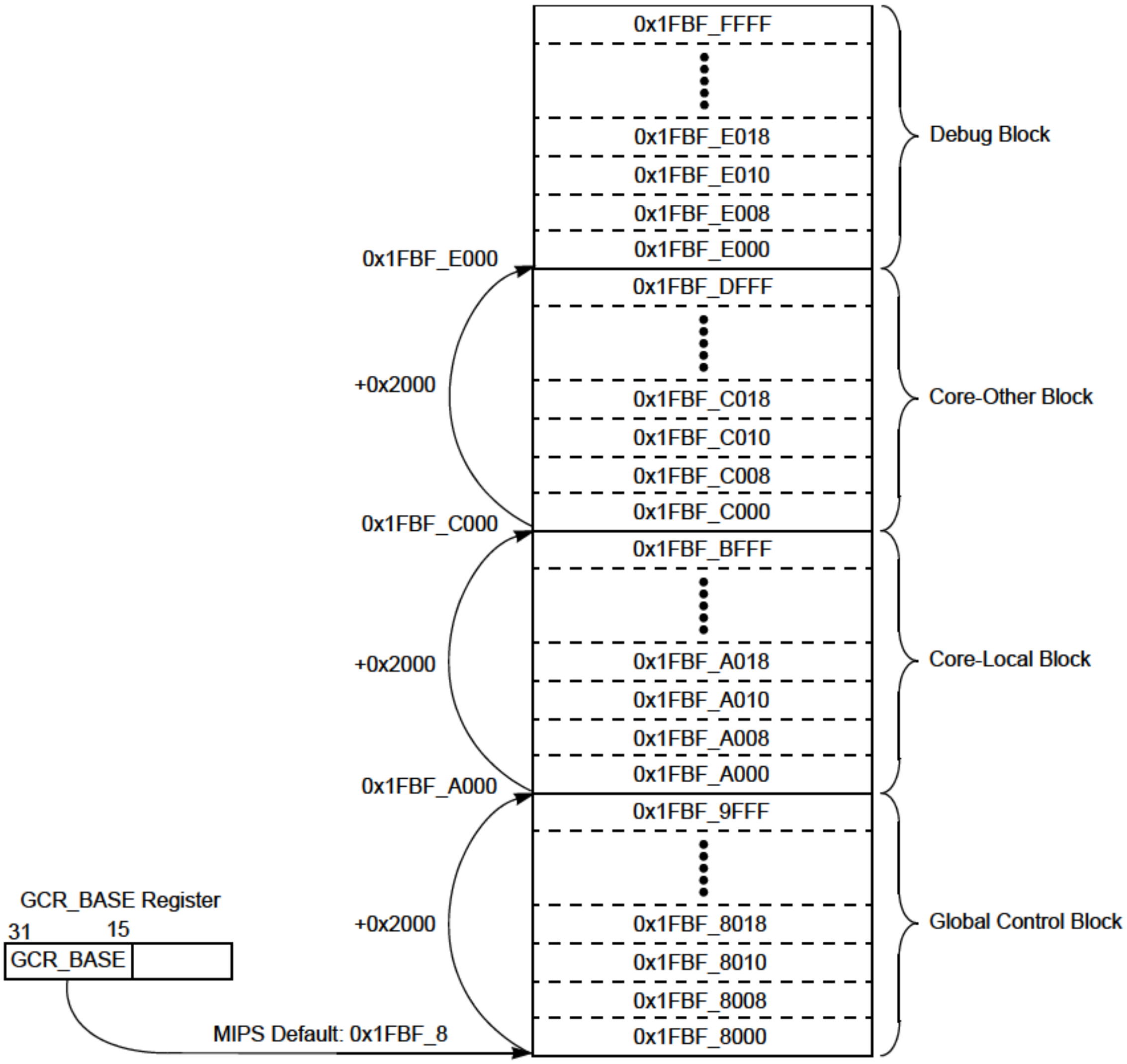
Table 8.4 Absolute Address of Individual Core-Local Block Registers

MIPS Default Base		Core-Local Block Offset		Core-Local Register Offset		Absolute Physical Address	Global Control Register
0x1FBF_8	+	0x2000	+	0x0000	=	0x1FBF_ A000	Reserved.
0x1FBF_8	+	0x2000	+	0x0008	=	0x1FBF_ A008	Core-Local Coherence Control.
0x1FBF_8	+	0x2000	+	0x0010	=	0x1FBF_ A010	Core-Local Configuration.
0x1FBF_8	+	0x2000	+	0x0018	=	0x1FBF_ A018	Core-Other Addressing.
0x1FBF_8	+	0x2000	+	0x0020	=	0x1FBF_ A020	Core-Local Reset Exception Base.
0x1FBF_8	+	0x2000	+	0x0028	=	0x1FBF_ A028	Core-Local Identification.
0x1FBF_8	+	0x2000	+	0x0030	=	0x1FBF_ A030	Core-Local Reset Exception Extended Base.
0x1FBF_8	+	0x2000	+	0x0040	=	0x1FBF_ A040	TCID 0 Priority.

The Core-Other block would be accessed in the same manner, just with a different (Core-Other) block offset (0x4000).

This concept is described in Figure 8.1 below. For simplicity, the MIPS default value is used for the GCR base address.

Figure 8.1 CM2 Register Addressing Scheme Using the MIPS Default in GCR_BASE



8.2 CM2 Programming

This section provides programming examples based on the capability of the CM2 register set. Some topics described are:

- Section 8.2.1, "Verifying Overall System Configuration"
- Section 8.2.2, "Requestor Access to GCR Registers"
- Section 8.2.3, "CM2 Interface Ports"
- Section 8.2.4, "Setting the CM2 Register Block Base Address"
- Section 8.2.5, "Address Regions"
- Section 8.2.6, "Address Map Programming Example"
- Section 8.2.9, "Accessing Another Cores CM2 GCR Registers"
- Section 8.2.10, "Boot Exception Vector Configuration"
- Section 8.2.11, "Coherency Domains"
- Section 8.2.12, "L2-Only SYNC Operation"
- Section 8.2.13, "Handling of Addresses Not Mapped to a Defined Region"
- Section 8.2.14, "Setting the Cache Coherency Attributes for Default Memory Transfers"
- Section 8.2.15, "In-Flight L1 and L2 Cache Operations"
- Section 8.2.16, "MIPS System Trace"
- Section 8.2.17, "Error Processing"
- Section 8.2.18, "Custom GCR Implementation"
- Section 8.2.19, "Attribute-Only Regions"

8.2.1 Verifying Overall System Configuration

At build-time, the developer selects the number of cores in the system, the number of I/O coherency units (IOCU's), and the number of address reqions. When the device is built, these values are hardwired into the *Global Configuration* register at offset address 0x0000. Reading this register provides the following information:

- Bits 7:0 — Number of cores in the system (1, 2, 3, 4, or 6)
- Bits 11:8 — Number of IOCU's (0, 1, or 2)
- Bits 19:16 — Number of address regions

8.2.2 Requestor Access to GCR Registers

The CM2 allows up to eight requestor's in a system. A requestor can be either a core or an IOCU. The proAptiv core allows up to 8 requestors in a multiprocessing system; six cores and two IOCU's.

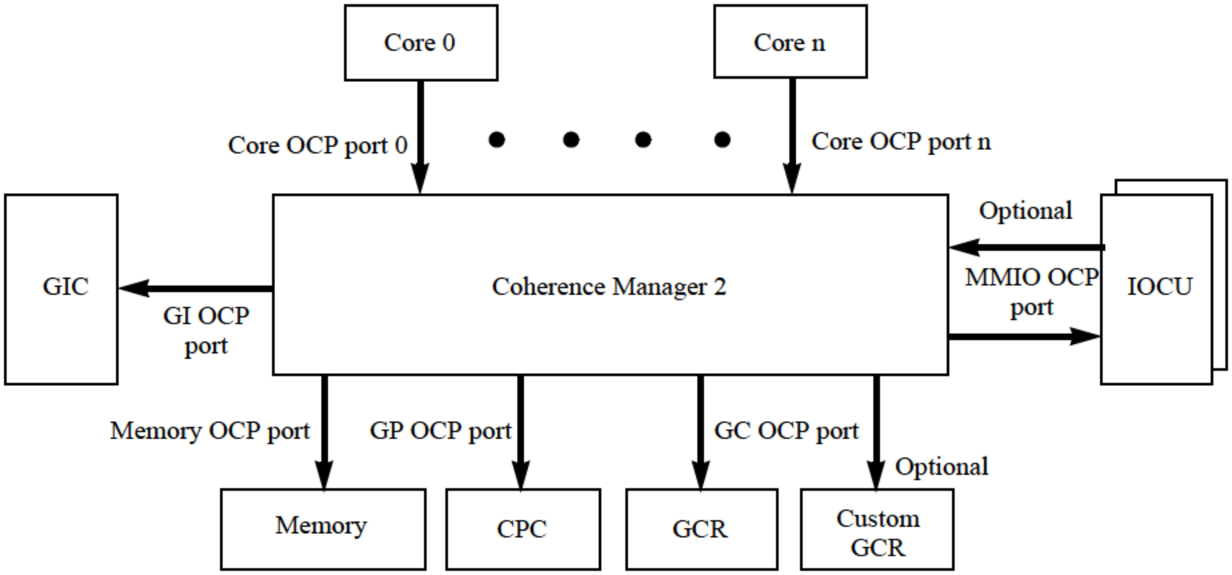
The requestor's may not have unrestricted access to the CM2 registers. During boot time, software determines which requestor's are provided access to the CM2 registers by programming the *CM2_ACCESS_EN* field of the *Global CSR Access Privilege* register located at offset 0x0020. Each bit in this field corresponds to a specific requestor.

The MIPS default for this field is 0xFF, meaning that all requestor's in the system have access to the CM2 register set. To disable access to the registers for a particular requestor, software need only clear the corresponding bit of this field to zero and all write requests to the CM2 registers by that requestor will be ignored.

8.2.3 CM2 Interface Ports

The CM2 contains numerous ports that allow the various system peripherals to communicate with the CM2. The ports connected to the CM2 are shown in Figure 8.2. The proAptiv Multiprocessing System can have up to 6 cores.

Figure 8.2 Interface Ports of the CM2



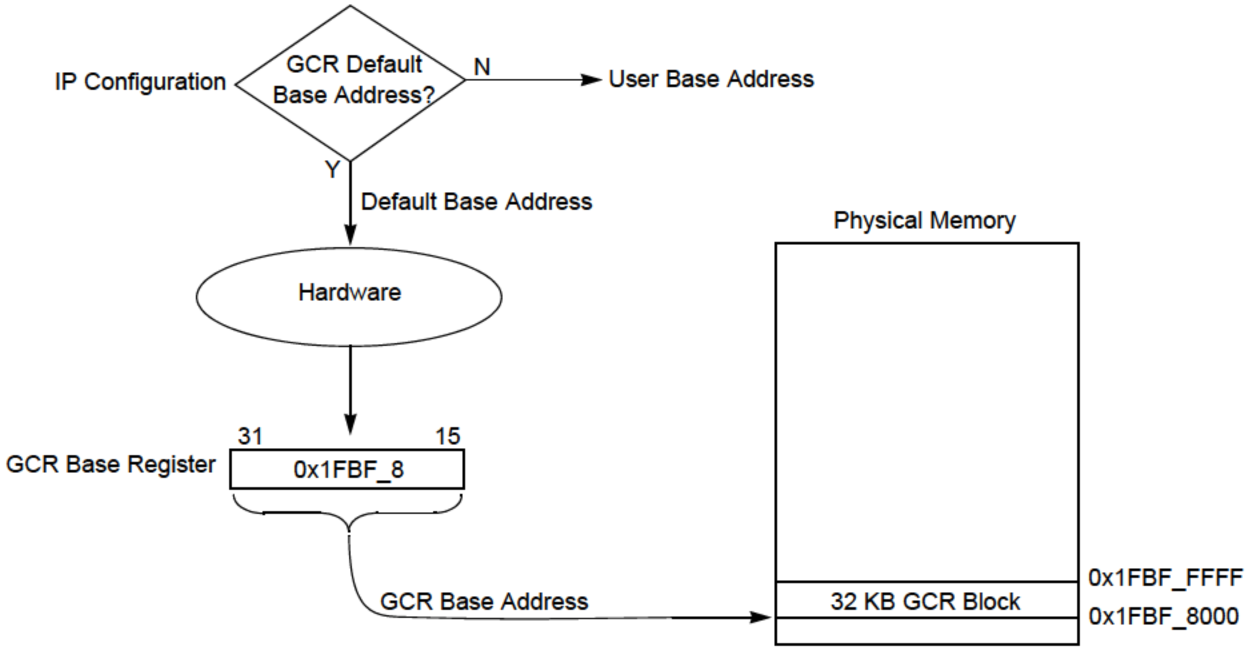
8.2.4 Setting the CM2 Register Block Base Address

As shown in Table 8.1 above, the CM2 register map contains four contiguous 8K blocks and can be located anywhere within physical memory. During IP configuration, the user can select the option to use the MIPS default base address of 0x1FBF_8, or they can select any 32 KB location in memory to locate the CM2 registers.

This decision determines how the 17-bit GCR_BASE field is programmed. If the MIPS default base address option is selected, a value of 0x1FBF_8 is loaded into this field. If the user selects their own base address, then that address is programmed into the GCR_BASE field. Refer to Section 8.3.2.2, "GCR Base Register (GCR_BASE Offset 0x0008)" for more information. In addition to the value in the GCR_BASE field, the user can also select whether this field is R/W or RO during IP configuration.

The following example shows the assignment of the CM2 GCR registers in memory using the MIPS default address. Note that the physical address is shown in this diagram. During actual programming, the programmer may use the virtual address associated with a physical address of 0x1FBF_8 to address the GCR block. The virtual address is provided prior to address translation and will be different from the resulting physical address. Refer to Chapter 3 of this manual for more information on virtual to physical address translation.

Figure 8.3 Mapping the CM2 Registers in Physical Memory Using the MIPS Default Value



8.2.5 Address Regions

The CM2 divides the address space into two types of regions:

- Fixed-size regions
- Variable-size regions

8.2.5.1 Fixed-Size Regions

Fixed-size regions are those that have a fixed size in memory. These include:

- GCR Base; contains the global, core-local, core-other, and debug register blocks, fixed at 32 KB.
- GIC (global interrupt controller) address space, fixed at 128 KB
- CPC (cluster power controller) address space, fixed at 32 KB
- Custom GCR address space, fixed at 64 KB

The 32 KB GCR Base region is futher divided into four 8 KB blocks as described in Table 8.1. Refer to Section 8.2.4, "Setting the CM2 Register Block Base Address" for more information on setting the base address in memory for the CM2 register block.

The GIC region is fixed at 128 KB. Refer to Section 8.3.3.1, "Global Interrupt Controller Base Address Register (GCR_GIC_BASE Offset 0x0080)" for more information on programming the base address for the GIC interface.

The CPC region is fixed at 32 KB. Refer to Section 8.3.3.2, "Cluster Power Controller Base Address Register (GCR_CPC_BASE Offset 0x0088)" for more information on programming the base address for the CPC interface.

The Custom GCR region is fixed at 64 KB. Refer to Section 8.3.2.11, "GCR Custom Base Register (GCR_CUSTOM_BASE Offset 0x0060)" for more information on programming the base address for the Custom GCR interface.

8.2.5.2 Variable-Size Regions

The proAptiv multiprocessing system may provide four programmable variable size address regions for mapping the IOCU's and memory. The number of regions is determined at IP configuration time. If an IOCU is not present, then the regions registers are not used. The number of regions implemented is determined as follows.

Table 8.5 Setting the Number of Regions

ADDR_REGIONS Field	Number of Regions	Region Assignments
0x0	0	None (typically used when there is no IOCU).
0x4	4	4 standard regions.
0x6	6	4 standard regions and 2 attribute-only regions.
0x8	8	4 standard regions and 4 attribute-only regions.

For more information, refer to the ADDR_REGIONS field in bits 19:16 of the Section 8.3.2.1, "Global Config Register (GCR_CONFIG Offset 0x0000)". For more information on the attribute-only regions, refer to Section 8.2.19.

Each region is controlled by a corresponding base and mask register as described below. These registers are used to determine not only the location and size of the memory space, but also whether this space is mapped to an IOCU or to memory. In addition, the cache coherency attributes (CCA) for each region can be defined as described in Section 8.2.5.6, "Setting the Cache Coherency Attributes for Region Memory Transfers".

In a MIPS core, mapped addresses are processed by the memory management unit (MMU) and the cache coherency attributes for a given memory page are determined. In this case, the CCA corresponds to both the L1 and L2 caches. In some sitations it may be advantageous to have the CCA of the L2 different from that of the L1 cache. In this case, software can use the *CCA_Override_Value* field of each *Region Address Mask* register to set the CCA for the L2 cache. This changes the attributes of the cache from what was originally assigned by the core.

The CM2 provides four base address and four address mask registers for controlling variable-size address regions 0 through 3. These regions control how some transactions are routed by the CM2. The possible routing options for requests that map to these variable-size regions are:

- To/From Memory via the CM2's system memory OCP port
- To/From the IOCU's via the CM2's MMIO OCP port for Memory-Mapped I/O (in hardware I/O coherent systems only)

Refer to Section 8.3.3.3, "CM2 Region [0 - 3] Base Address Register (GCR_REGn_BASE Offsets 0x0090, 0x00A0, 0x00B0, 0x00C0)" and Section 8.3.3.4, "CM2 Region [0 - 3] Address Mask Register (GCR_REGn_MASK Offsets 0x0098, 0x00A8, 0x00B8, 0x00C8)" for more information on these registers.

8.2.5.3 Address Region Priorities

The priority for the region decode is as follows:

1. GCR (highest priority)

2. Custom GCR

3. CPC

4. GIC

5. Programmed MMIO regions

6. Programmed memory regions

7. *CM2_DEFAULT_TARGET* (lowest priority)

The above priority allows for large memory regions to be defined with small IOCU regions carved out. Note that these regions can overlap as described in Section 8.2.5.8, "Overlapping Regions".

8.2.5.4 Defining the Base Address Location and Size for Each Region

The address map is programmable through a set of registers located in the GCR as summarized below. Up to 8 variable-size programmable regions can be implemented. When an IOCU is present (i.e., hardware I/O Coherence is implemented), these regions determine if requests are routed to memory or to the IOCU via the CM2's MMIO port. The regions can also be used with or without an IOCU for the CCA Override feature as described in Section 8.2.14 "Setting the Cache Coherency Attributes for Default Memory Transfers".

- The *GCR Base Register* defines the address base of the GCR region. The GCR region has a fixed size of 32 KB (see Table 8.19), hence no corrresponding Mask register is required. Note that this region must reside on a 32 KB boundary.

- The *Cluster Power Controller Base Address Register* defines the address base of the CPC address region. This CPC region may be disabled via the *CPC_EN* bit in that register. When enabled, the CPC address region has a fixed size of 32 KB (see Table 8.32), hence no corrresponding Mask register is required. Note that this region must reside on a 32KB boundary.

- The *Global Interrupt Controller Base Address Register* defines the address base of the GIC address region. This GIC region may be disabled via the *GIC_EN* bit in that register. When enabled, the GIC address region has a fixed size of 128 KB (see Table 8.31), hence no corrresponding Mask register is required. Note that this region must reside on a 128 KB boundary.

- The *CM2 Region [0-3] Base Address Registers* define the address base for each of the four programmable regions. The regions have a programmable base address and a programmable size that is selected via the corresponding Mask register.

- The *CM2 Region [0-3] Address Mask Registers* define the size for each of the four programmable regions. These registers work in conjunction with the corresponding *CM2 Region [0-3] Base Address Registers* to configure a given region.

- The *Custom GCR Base Register* defines the address base of the Custom GCR region. This region defines the location of registers that are implemented by the user. This region may be disabled via the *GGU_EN* bit in the *Custom GCR Base Register*. When enabled, the Custom GCR region has a fixed size of 64 KB (see Table 8.28), hence no corrresponding Mask register is required. Note that this region must reside on a 64 KB boundary.

As described above, the base of each region is defined in the corresponding CM2 *Region [0,1,2,3] Address Base Register* (see Table 8.33), and the size of the region is defined in the corresponding *CM2 Region [0,1,2,3] Address Mask Register* (see Table 8.34). Because a base/mask scheme is used, the base must be located on a boundary of its size. A region can be sized from 64K to the entire 32-bit address space.

Table 8.6 Setting the Base Address for the CM2 Peripheral Devices

Block	Register Name	Offset Address	Field Name	Bits	Description
GCR	GCR_BASE	0x0008	GCR_BASE_ADDR	31:15	Sets the base address of the GCR registers. This field has a fixed size of 32 KB.
Custom GCR	GCR_CUSTOM_BASE	0x0060	CUSTOM_ BASE	31:16	Sets the base address of the Customer GCR registers. This field has a fixed size of 64 KB.
GIC	GCR_GIC_BASE	0x0080	GIC_BASE_ADDR	31:17	Sets the base address of the GIC. This field has a fixed size of 128 KB.
CPC	GCR_CPC_BASE	0x0088	CPC_BASE_ADDR	31:15	Sets the base address of the CPC. This field has a fixed size of 32 KB.
Region 0	GCR_REG0_BASE	0x0090	REGION0_BASE_ADDR	31:16	Sets the base address of region 0 in memory. Minimum size is 64 KB.
	GCR_REG0_MASK	0x0098	REGION0_BASE_MASK	31:16	Sets the size of region 0 in memory.
Region 1	GCR_REG1_BASE	0x00A0	REGION1_BASE_ADDR	31:16	Sets the base address of region 1 in memory. Minimum size is 64 KB.
	GCR_REG1_MASK	0x00A8	REGION1_BASE_MASK	31:16	Sets the size of region 1 in memory.
Region 2	GCR_REG2_BASE	0x00B0	REGION2_BASE_ADDR	31:16	Sets the base address of region 2 in memory. Minimum size is 64 KB.
	GCR_REG2_MASK	0x00B8	REGION2_BASE_MASK	31:16	Sets the size of region 2 in memory.
Region 3	GCR_REG3_BASE	0x00C0	REGION3_BASE_ADDR	31:16	Sets the base address of region 3 in memory. Minimum size is 64 KB.
	GCR_REG3_MASK	0x00C8	REGION3_BASE_MASK	31:16	Sets the size of region 3 in memory.

As described above, some of the blocks are a fixed size, hence there is no corresponding Mask register. Since the GCR, GIC, and CPC blocks each contain a dedicated Base Address register, the Region 0 - 3 registers are used to access the memory and IOCU peripherals.

8.2.5.5 Defining the Target Device

Each *CM2 Region Address Mask* register contains a field that determines how the CM2 routes requests whose address matches the corresponding region. As defined in the *CM2_REGION_TARGET* field, the transaction may be routed to memory or to an I/O device via the CM2's MMIO port and IOCU. A region may be disabled by setting the *CM2_REGION_TARGET* in the corresponding *CM2 Region Address Mask* register to 0.

The *CM2_DEFAULT_TARGET* field in the *GCR Base Register* determines how to route the requests that don't match any of the defined regions. Refer to Section 8.2.13, "Handling of Addresses Not Mapped to a Defined Region" for more information.

8.2.5.6 Setting the Cache Coherency Attributes for Region Memory Transfers

As described in Section 8.2.4 "Setting the CM2 Register Block Base Address", the proAptiv core provides a CCA override capability that allows the CCA's for the L2 cache to be different from those of the L1 data cache.

This capability can be achieved via the CCA override feature in the CM2 Region Address Map Registers listed in Table 8.6. Software can establish up to 4 address map regions by programming the *CM2 Region Base Register 0-3* and *CM2 Region Mask Register 0-3*.

Programming the CCA

Each region has the *CCA_Override_Enable* and *CCA_Override_Value* fields which can be used to set the CCA for transactions on the system memory OCP port. If the *CCA_Override_Enable* field is set to 1 for a given region and the corresponding *CM2_TARGET* field in bits 1:0 is set to memory (0x1), then transactions that map to that region and proceed to the system memory port will have a CCA value set to the corresponding *CCA_Override_Value* for that region. This field also determines the CCA value driven to system memory.

Any valid CCA value can be programmed into *CCA_Override_Value*, but because the L2 does not process coherent CCAs, a value of CWB (5) or CWBE (4) is automatically changed to WB (3) by the CM2 before being driven on the system memory OCP port. The encoding of the *CCA_Override_Value* field is identical to that shown in Table 8.8.

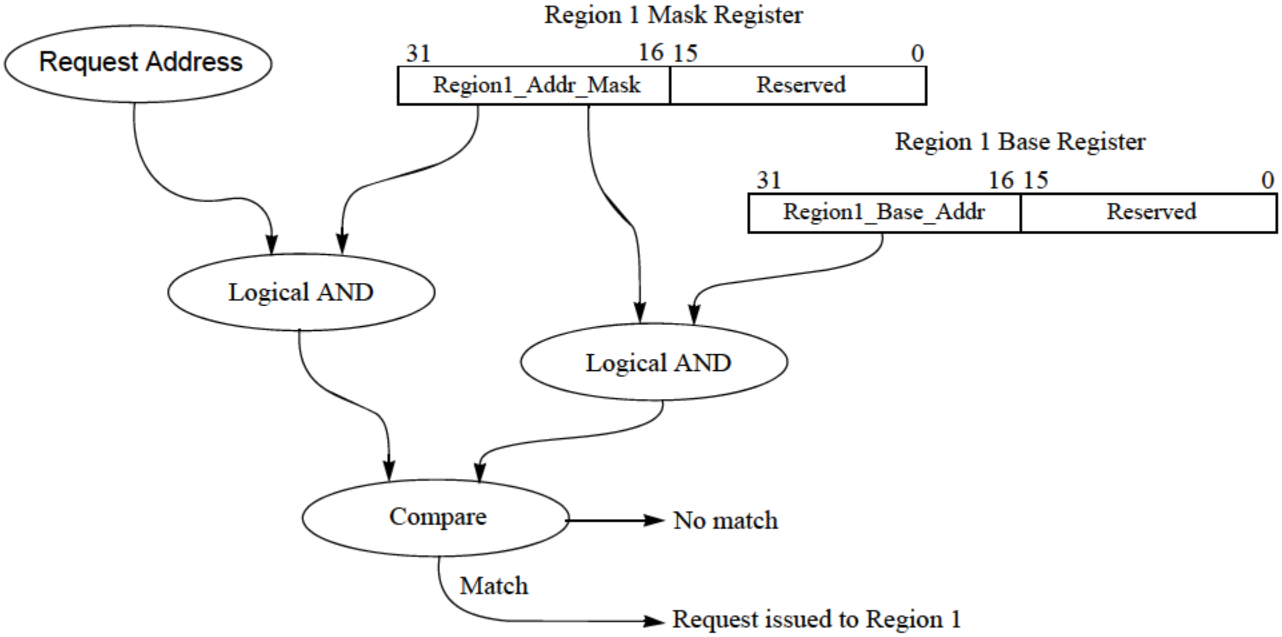
8.2.5.7 Issue Request Protocol and Region Masking

The CM2 contains four region mask registers used to set the size of a given region. These mask registers work in conjunction with their corresponding base address registers as shown in Table 8.6. The requesting address is logically ANDed with the value in the selected *Region Address Mask* register. At the same time, the value in the corresponding *REGION_BASE_ADDR* field is compared to the value in the *Region Address Mask* register. If both outputs match, the request is routed to this region.

For example, if the requesting address is compared to the value in the *CM2_REGION1_BASE_ADDR* and the *CM2_REGION1_ADDR_MASK* registers and there is a match, then the requesting address is routed to region 1. This concept is shown in Figure 8.4.

The only allowed values in this register are contiguous sets of leading 0x1’s. An 0x1 preceded by a 0x0 is not allowed (e.g., the value of 0xFFF0 is allowed, but the value 0xFFEF is not allowed).

Figure 8.4 Mapping a Request to Region 1

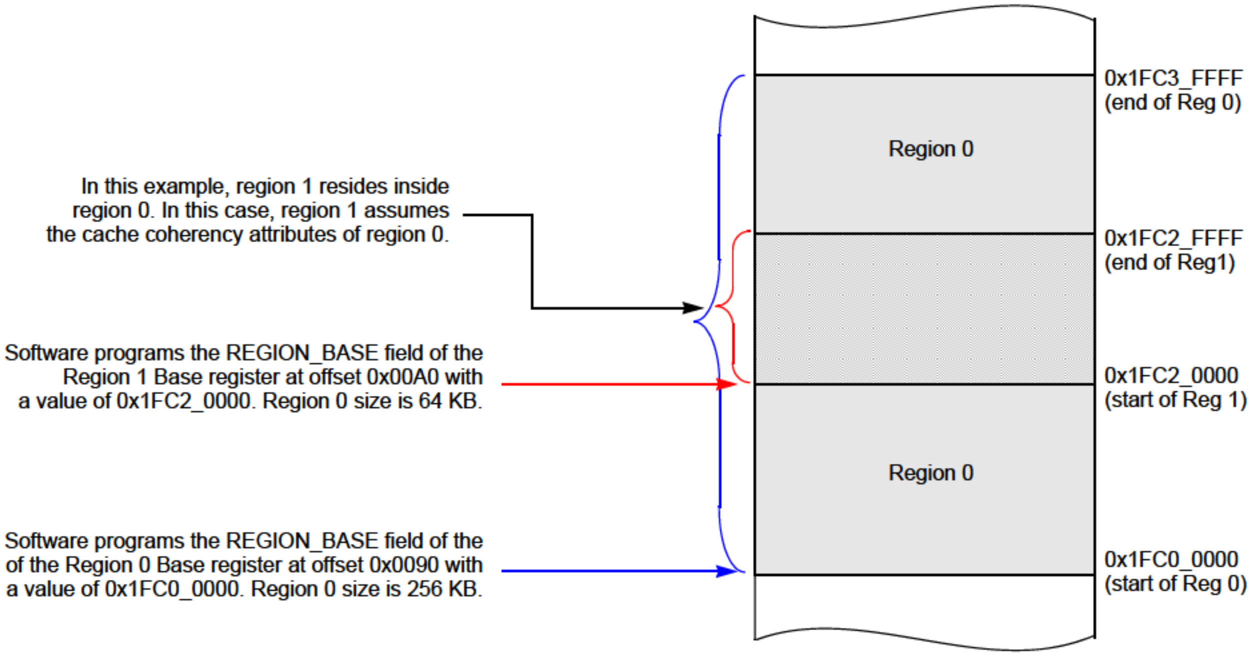


8.2.5.8 Overlapping Regions

Since overlapping regions are supported, it is possible that an address maps to more than one region. In this case, the CCA override enable and value are used from the lowest numbered region mapped to memory. For example, if an address matches both *CM2 Region Base/Mask Register 0* and *CM2 Region Base/Mask Register 1*, and both regions 0 and 1 are mapped to Memory (*CM2_REGION_TARGET* is set to 1 in both *CM2 Region Mask Register 0* and *1*), then the values of *CCA_Override_Enable* and *CCA_Override_value* in CM2 Region Mask Register 0 will be used to determine the CCA value driven on the system memory OCP Port.

This concept is shown in Figure 8.5. In this example, region 1 is a 64 KB space located inside the larger 256 KB region 0.

Figure 8.5 Example of Overlapping Regions

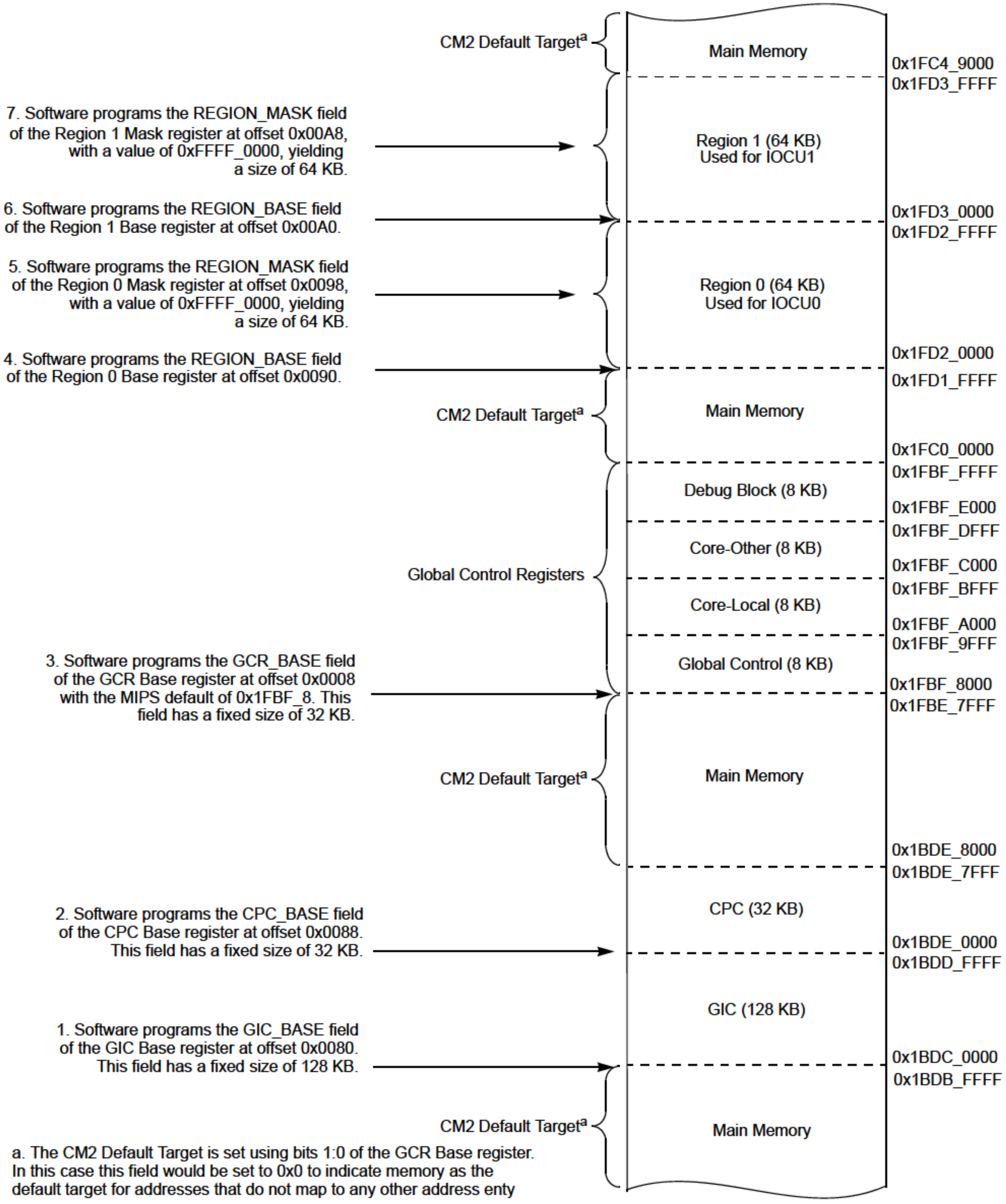

0x1FC3_FFFF
(end of Reg 0)
Region 0
In this example, region 1 resides inside region 0. In this case, region 1 assumes the cache coherency attributes of region 0.
0x1FC2_FFFF
(end of Reg1)
Software programs the REGION_BASE field of the Region 1 Base register at offset 0x00A0 with a value of 0x1FC2_0000. Region 0 size is 64 KB.
0x1FC2_0000
(start of Reg 1)
Region 0
Software programs the REGION_BASE field of the of the Region 0 Base register at offset 0x0090 with a value of 0x1FC0_0000. Region 0 size is 256 KB.
0x1FC0_0000
(start of Reg 0)

When overriding a CCA value, only the CCA driven to the system memory OCP is affected. Otherwise, the functionality of the transaction within the CM2 is based on the original CCA. When the CM2 is programmed to override the CCAs for an address region, all accesses to that region including speculative reads and writebacks (explicit or implicit) from the L1 are overridden. Transactions that are never mapped to regions, such as Legacy Syncs, CohCompletionSyncs or L2/L3 CacheOps are unaffected by the CCA override functionality.

8.2.6 Address Map Programming Example

This subsection provides an example of memory mapping for all of the aforementioned regions at different locations using the MIPS default base address. The memory map for this example is shown in Figure 8.6.

Figure 8.6 Address Map Programming Example



7. Software programs the REGION_MASK field of the Region 1 Mask register at offset 0x00A8, with a value of 0xFFFF_0000, yielding a size of 64 KB.

6. Software programs the REGION_BASE field of the Region 1 Base register at offset 0x00A0.

5. Software programs the REGION_MASK field of the Region 0 Mask register at offset 0x0098, with a value of 0xFFFF_0000, yielding a size of 64 KB.

4. Software programs the REGION_BASE field of the Region 0 Base register at offset 0x0090.

3. Software programs the GCR_BASE field of the GCR Base register at offset 0x0008 with the MIPS default of 0x1FBF_8. This field has a fixed size of 32 KB.

2. Software programs the CPC_BASE field of the CPC Base register at offset 0x0088. This field has a fixed size of 32 KB.

1. Software programs the GIC_BASE field of the GIC Base register at offset 0x0080. This field has a fixed size of 128 KB.

a. The CM2 Default Target is set using bits 1:0 of the GCR Base register. In this case this field would be set to 0x0 to indicate memory as the default target for addresses that do not map to any other address enty

The following programming sequence is used to configure the memory map as shown in Figure 8.6 above.

1. Sofware programs the *GIC_BASE* field of the *GIC Base* register located at offset 0x0080 with a value of 0x1BDC. This sets the base address of the GIC registers. This block has a fixed size of 128 KB. Refer to bits 31:17 in Section 8.3.3.1, "Global Interrupt Controller Base Address Register (GCR_GIC_BASE Offset 0x0080)" for more information. Note that this block must reside on a 128 KB boundary.

2. Sofware programs the *CPC_BASE* field of the *CPC Base* register located at offset 0x0088 with a value of 0x1BDE_0. This sets the base address of the CPC registers. This block has a fixed size of 32 KB. Refer to bits 31:15 in Section 8.3.3.2, "Cluster Power Controller Base Address Register (GCR_CPC_BASE Offset 0x0088)" for more information. Note that this block must reside on a 32 KB boundary.

3. Software programs the *GCR_BASE* field of the *GCR Base* register located at offset 0x0008 with a value of 0x1FBF_8. This sets the base address of the 32 KB block of GCR registers. This block is divided into four 8 KB subblocks that contain the Global, Core-Local, Core-Other, and Debug register blocks. Note that if the MIPS default address of 0x1FBF_8 is selected for the base address of the GCR registers during IP configuration, this field becomes read-only. In this case, hardware writes the default value of 0x1FBF_8 to this field. Refer to bits 31:15 in Section 8.3.2.2, "GCR Base Register (GCR_BASE Offset 0x0008)" for more information.

4. Software programs the *REGION_BASE_ADDR* field of the *CM2 Region 0 Base* register located at offset 0x0090 with a value of 0x1FD2. This sets the base address of region 0 to 0x1FD2_0000. Refer to bits 31:16 in Section 8.3.3.3, "CM2 Region [0 - 3] Base Address Register (GCR_REGn_BASE Offsets 0x0090, 0x00A0, 0x00B0, 0x00C0)" for more information.

5. Software programs the *REGION_ADDR_MASK* field of the *CM2 Region 0 Address Mask* register located at offset 0x0098 with a value of 0xFFFF_0000. This sets the size of region 0 to 64 KB. Refer to bits 31:16 in Section 8.3.3.4, "CM2 Region [0 - 3] Address Mask Register (GCR_REGn_MASK Offsets 0x0098, 0x00A8, 0x00B8, 0x00C8)" for more information. Other values for this field could be 0xFFFE (128 KB), 0xFFFC (256 KB), etc.

6. Software programs the *REGION_BASE_ADDR* field of the *CM2 Region 1 Base* register located at offset 0x00A0 with a value of 0x1FD3. This sets the base address of region 1 to 0x1FD3_0000. Refer to bits 31:16 in Section 8.3.3.3, "CM2 Region [0 - 3] Base Address Register (GCR_REGn_BASE Offsets 0x0090, 0x00A0, 0x00B0, 0x00C0)" for more information.

7. Software programs the *REGION_ADDR_MASK* field of the *CM2 Region 1 Address Mask* register located at offset 0x00A8 with a value of 0xFFFF_0000. This sets the size of region 1 to 64 KB. Refer to bits 31:16 in Section 8.3.3.4, "CM2 Region [0 - 3] Address Mask Register (GCR_REGn_MASK Offsets 0x0098, 0x00A8, 0x00B8, 0x00C8)" for more information. Other values for this field could be 0xFFFE (128 KB), 0xFFFC (256 KB), etc.

8. Software programs the *CM2_DEFAULT_TARGET* field of the *GCR Base* register with a value of 2’b00, indicating that memory is the target device for addresses that do not map to any of the address blocks shown in Figure 8.6. Refer to bits 1:0 in Section 8.3.2.2, "GCR Base Register (GCR_BASE Offset 0x0008)" for more information.

9. Software programs the *CM2_TARGET* field of the *CM2 Region 0 Address Mask* register located at offset 0x0098 with a value of 2’b10. This maps region 0 to IOCU0.

10. Software programs the *CM2_TARGET* field of the *CM2 Region 1 Address Mask* register located at offset 0x00A8 with a value of 2’b11. This maps region 1 to IOCU1.

8.2.7 Core-Local GCRs

The Core-Local GCR block contains the configuration and status registers for a given core. Each core has its own copy of Core-Local registers. A core can access its own Core-Local block to determine the programmable parameters for that core. Parameters include base address assignments for cache coherency attributes, reset exception base, boot exception vector mask, etc.

8.2.8 Core-Other GCRs

The Core-Other GCR block is a single block that all of the cores have access to, and provides a way for one core to access the Core-Local registers of another core. Before a core can access the Core-Other space, the *Core-Other Addressing* register in that cores own Core-Local Control Block must be set with the core number (CORENUM) of the target core. In this case, a particular core would program the *Core-Other Addressing* register in its own Core-Local block with the core number to be accessed. The core would then write the contents of the register to be accessed into the Core-Other address space.

8.2.9 Accessing Another Cores CM2 GCR Registers

As shown in Table 8.1, the CM2 provides two blocks of registers.

- Core-Local (offset range 0x2000 - 0x3FFF)
- Core-Other (offset range 0x4000 - 0x5FFF)

Each core contains a copy of these registers. The Core-Local address space contains the GCR registers for that core. The Core-Other address space allows a core to access the GCR registers for another cores Core-Local GCR block.

As described in Section 8.2.4, these registers can be located anywhere in physical memory if this option is selected during IP configuration. If this option is not selected, the location of these registers are located at the MIPS default address of 0x1FBF_8000. Refer to Section 8.1 "Coherence Manager Address Map" and related subsection for more information on use of the MIPS default memory location.

The Core-Local block represents registers corresponding to that core. If a core wishes to modify the contents of its own set of CM2 GCR registers, it writes to the Core-Local block located at the address range shown in Table 8.1. If a core wishes to program the GCR registers of another core, it selects the core number and writes this value into the Core-Other Addressing register in its own Core-Local block at offset address 0x0018. The actual register in the other core to be written would use the corresponding offset in the Core-Other block shown in Table 8.1.

In a multiprocessor system, it is common for one core to boot up first, then have that core boot the other cores in the system. In the following example, assume core 0 is booted up first. Then core 0 is used to program the GCR registers in core 1. This example examines how core 0 would program the boot exception vector location for core 1. Note that this example uses the MIPS default addressing scheme. The programming sequence would be as follows:

1. Core 0 writes a value of 0x0001 to the *CORENUM* field (bits 31:16) of the *Core-Other Addressing* register located in its own Core-Local block at offset 0x0018 (physical address of 0x1FBF_A018 in Table 8.3). This indicates that the register to be programmed corresponds to core 1. Refer to Section 8.4.1.3, "Core-Other Addressing Register" for more information.

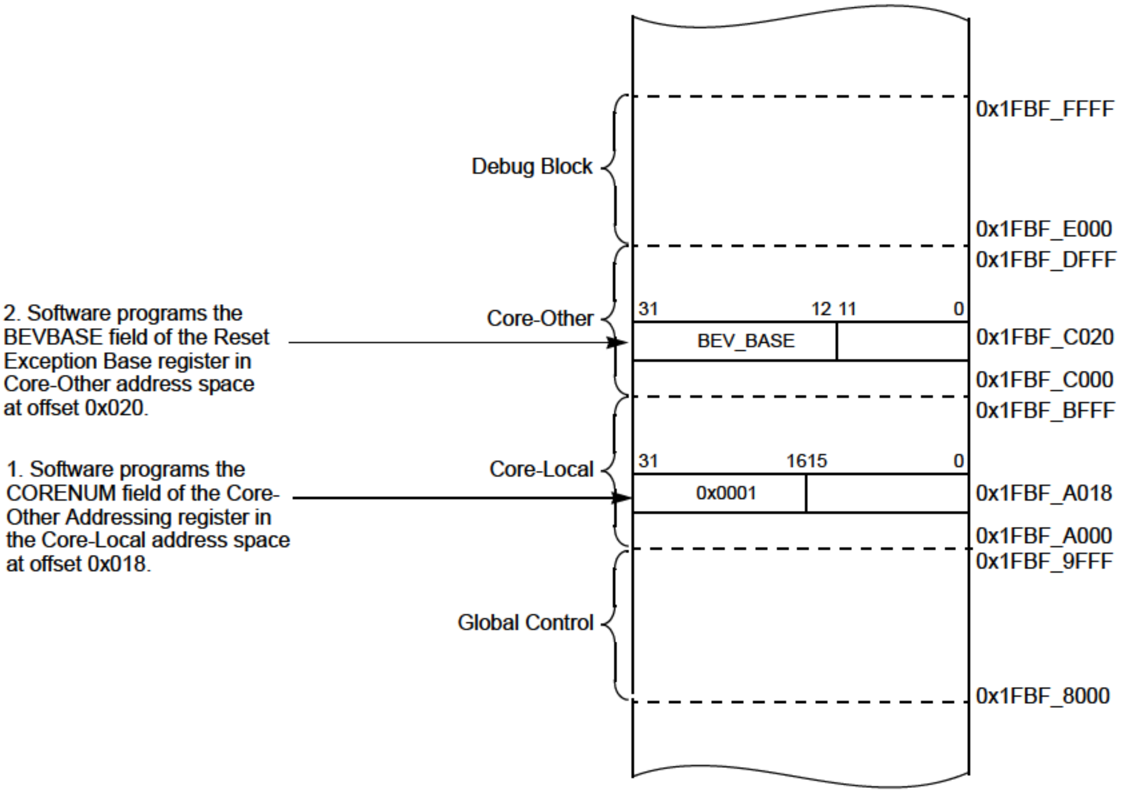
2. Core 0 writes the appropriate value into the *BEVEXCBase* field (bits 31:12) of the *Reset Exception Base* register located in the Core-Other block at offset 0x0020 (physical address of 0x1FBF_C020 in Table 8.4). Because core 0 is setting the BEV base value for core 1, as opposed to its own core, the write is done to the Core-Other address block. Refer to Section 8.4.1.4, "Core Local Reset Exception Base Register (GCR_Cx_RESET_BASE Offset 0x0020)" for more information.

Note that in addition to the *CORENUM* field in the *Core-Other Addressing* register used to indicate the number of the destination core as described in #1 above, a core can determine its own core number by reading the *CORENUM* field in its own *Core-Local Identification* register located at offset 0x0028 in Core-Local address space. Refer to Section 8.4.1.5, "Core Local Identification Register (GCR_Cx_ID Offset 0x0028)" for more information.

Whenever one core read or writes to the registers associated with another core, the number of the core to be written is programmed into that cores local CORENUM field as described in step 1 above. The actual register to be programmed is accessed via the Core-Other block as described in step 2 above.

Since there is only one Core-Other block in Table 8.1, this means that when one core wants to access any of the other cores in the system, the register to be accessed always resides in the Core-Other block, regardless of the number of cores in the system. The state of the CORENUM field in the *Core-Other Addressing* register in that cores own Core-Local space determines which core the data will be written to. This concept is shown in Figure 8.7.

Figure 8.7 Core 0 Accessing the BEV_BASE GCR of Core 1


Debug Block
Core-Other
Core-Local
Global Control
2. Software programs the BEVBASE field of the Reset Exception Base register in Core-Other address space at offset 0x020.
1. Software programs the CORENUM field of the Core-Other Addressing register in the Core-Local address space at offset 0x018.
31
12 11
0
BEV_BASE
31
1615
0
0x0001
0x1FBF_FFFF
0x1FBF_E000
0x1FBF_DFFF
0x1FBF_C020
0x1FBF_C000
0x1FBF_BFFF
0x1FBF_A018
0x1FBF_A000
0x1FBF_9FFF
0x1FBF_8000

8.2.10 Boot Exception Vector Configuration

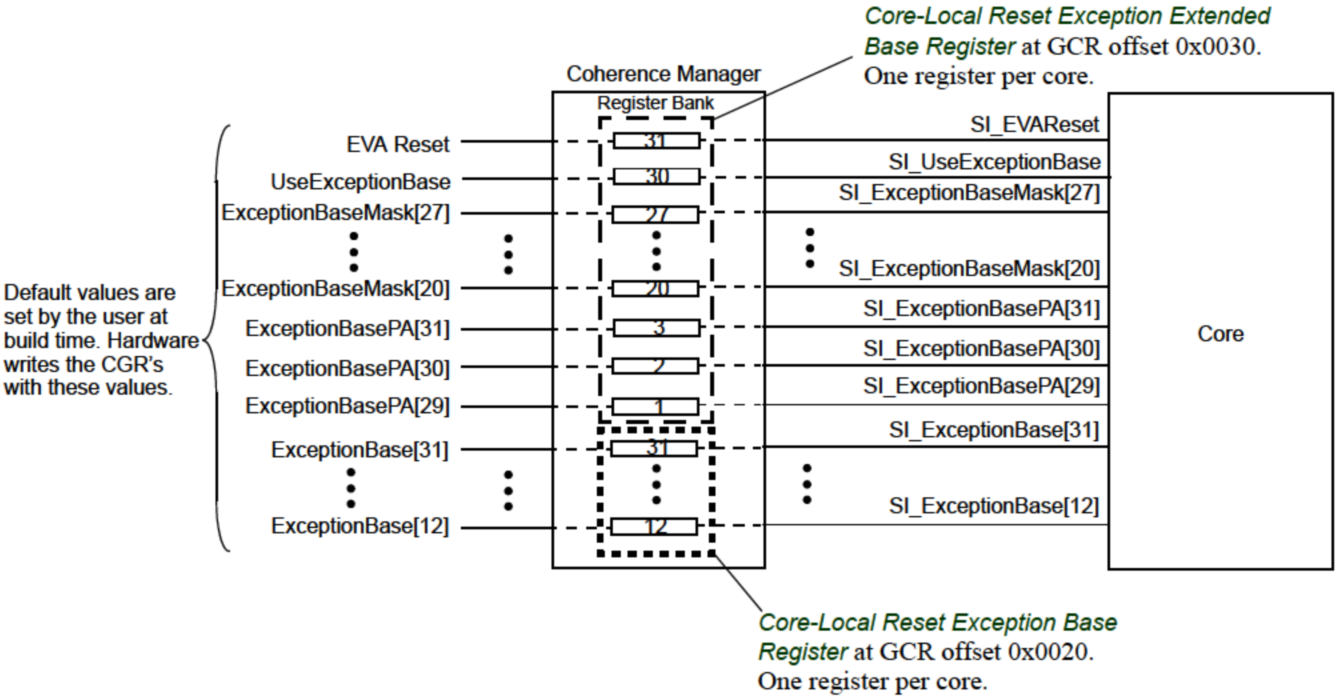
To facilitate the BEV overlay scheme, a number of pins were added to the proAptiv core that allow the user to select the boot overlay parameters at build time. In addition, the CM2 provides two registers in the Core-Local address space that allow the boot exception vector for each core to be located anywhere in physical memory.

The initial state of the default values selected by the user at build time are registered inside the Coherence Manager (CM2) block using two Core-Local Configuration Registers. As shown in Figure 8.8, there are two GCR registers used per core. Each core has its own pair of these GCR registers and its own set of BEV related pins. This allows each core to be programmed in a different manner and independently from one another. Refer to Section 8.4.1.4, "Core Local Reset Exception Base Register (GCR_Cx_RESET_BASE Offset 0x0020)" and Section 8.4.1.6, "Core Local Reset Exception Extended Base Register (GCR_Cx_RESET_EXT_BASE Offset 0x0030)" for more information on these two registers.

The CM2 drives these values to the proAptiv cores at reset. Note that the two CGR registers are loaded only on a cold boot and are programmed with the values selected by the user at build time. Each of these pins is described in Table 8.22.

Figure 8.8 shows the boot exception vector pins for a single proAptiv core. Each additional core would have an identical set of CM2 registers and set of BEV related pins shown in the figure. For more information, refer to Section 3.7 in the MMU chapter.

Figure 8.8 Registered Boot Exception Vector Relocation Pins — One Core



As noted in the figure above, there is one pair of GCR registers for each core. This allows each proAptiv core to be powered up in a different memory mode and independently from one another.

The boot exception vector relocation pins are described in Table 8.7.

Table 8.7 proAptiv Boot Exception Vector Pins

Pin Name	Field Size in Bits	CM2 GCR Register Mapping	Description
SI_EVAReset	1	Bit 31 of the *Core-Local Reset Exception Extended Base Register* (offset = 0x0030)	If this pin is asserted at reset, the proAptiv core comes up in the EVA configuration. In this case the *CONFIG5*.$_K$ bit becomes read-only with a fixed value of 1 to indicate EVA as the addressing scheme. In addition, the *SegCtl0* - *SegCtl2* registers are configured with values that correspond to the EVA mapping. If this pin is not asserted at reset, the proAptiv core comes up in the legacy setting. In this case the *CONFIG5*.$_K$ bit becomes read-write with an initial value of 0 to indicate legacy mode. This bit is modified by software when switching from legacy mode to EVA mode. This pin is used in both the legacy and EVA settings. There is one *SI_EVAReset* pin per core.
SI_UseExceptionBase	1	Bit 30 of the *Core-Local Reset Exception Extended Base Register* (offset = 0x0030)	In the legacy configuration, if the *SI_UseExceptionBase* pin is not asserted, then the BEV location defaults to 0xBFC0_0000. If the *SI_UseExceptionBase* pin is asserted, address bits *SI_ExceptionBase[31:30]* are forced to a value of 2'b10 to force the BEV location into the KSEG0/ KSEG1 space. This pin is only used in the legacy configuration. There is one *SI_UseExceptionBase* pin per core.
SI_ExceptionBaseMask[27:20]	8	Bits 27:20 of the *Core-Local Reset Exception Extended Base Register* (offset = 0x0030)	Used to determine the size of the boot exception vector overlay region from 1 MB to 256 MB in powers of two. These pins are used in both the legacy and EVA configurations. There is one set of *SI_ExceptionBaseMask* pins per core.
SI_ExceptionBasePA[31:29]	3	Bits 3:1 of the *Core-Local Reset Exception Extended Base Register* (offset = 0x0030)	Upper physical address bits. The size of the overlay region defined by *SI_ExceptionBaseMask[27:20]* is remapped to a location in physical address space pointed to by the *SI_ExceptionBasePA[31:29]* pins. This allows the overlay region to be placed into one of the 512 MB segments in physical memory. These pins are used in both the legacy and EVA configurations. There is one set of *SI_ExceptionBasePA* pins per core.

Table 8.7 proAptiv Boot Exception Vector Pins *(continued)*

Pin Name	Field Size in Bits	CM2 GCR Register Mapping	Description
SI_ExceptionBase[31:12]	20	Bits 31:12 of the *Core-Local Reset Exception Base Register* (offset = 0x0020)	The *SI_ExceptionBase[31:12]* pins define the boot address in virtual address space which is used to define the overlay region. These pins, along with the *SI_ExceptionBaseMask[27:20]* pins, determine the size and location of the BEV region within virtual address space. Note that the *CONFIG5.$_K$* CP0 register bit is used to determine which pins of the *SI_ExceptionBase[31:12]* address are used to calculate the overlay. These pins are used in the EVA setting and can also be used in the legacy setting. There is one set of *SI_ExceptionBase* pins per core.

8.2.11 Coherency Domains

The CM2 provides the *COH_DOMAIN_EN* field in *Core-Local Coherence Control* register at offset 0x0008 for managing the coherency aspects of each requestor in the system. There is one register per core. A requestor can be either a core or an IOCU.

In the 8-bit *COH_DOMAIN_EN* field, each bit corresponds to one requestor. Setting a given bit in the *COH_DOMAIN_EN* field for the GCR local register corresponding to a given core puts that core into coherent mode. If the same bit in the *COH_DOMAIN_EN* is 0 for the GCR local register corresponding to a given core, then that core is not in coherence mode and will never issue a coherent request.

For example, if bit 1 of this field is set, then interventions from core 1 to core 0 are enabled and can occur. Note that changing the coherence mode for a local core from 0x1 to 0x0 can only be done after flushing and invalidating all the cache lines in the core; otherwise, the system behavior is UNDEFINED.

Also note that if bit 1 of the *COH_DOMAIN_EN* field is set for the GCR local register corresponding to core 0, then software should also set bit 0 of the *COH_DOMAIN_EN* field for the GCR local register corresponding to core 1.

There is no need to program *COH_DOMAIN_EN* for the GCR local register corresponding to IOCUs.

Section 7.1.2, "Operating Level Transitions" in Chapter 7 of this manual provides examples of how this field is used to transition between coherency domains.

Figure 8.9 Encoding of COH_DOMAIN_EN Field — 2 or 4 Core Package

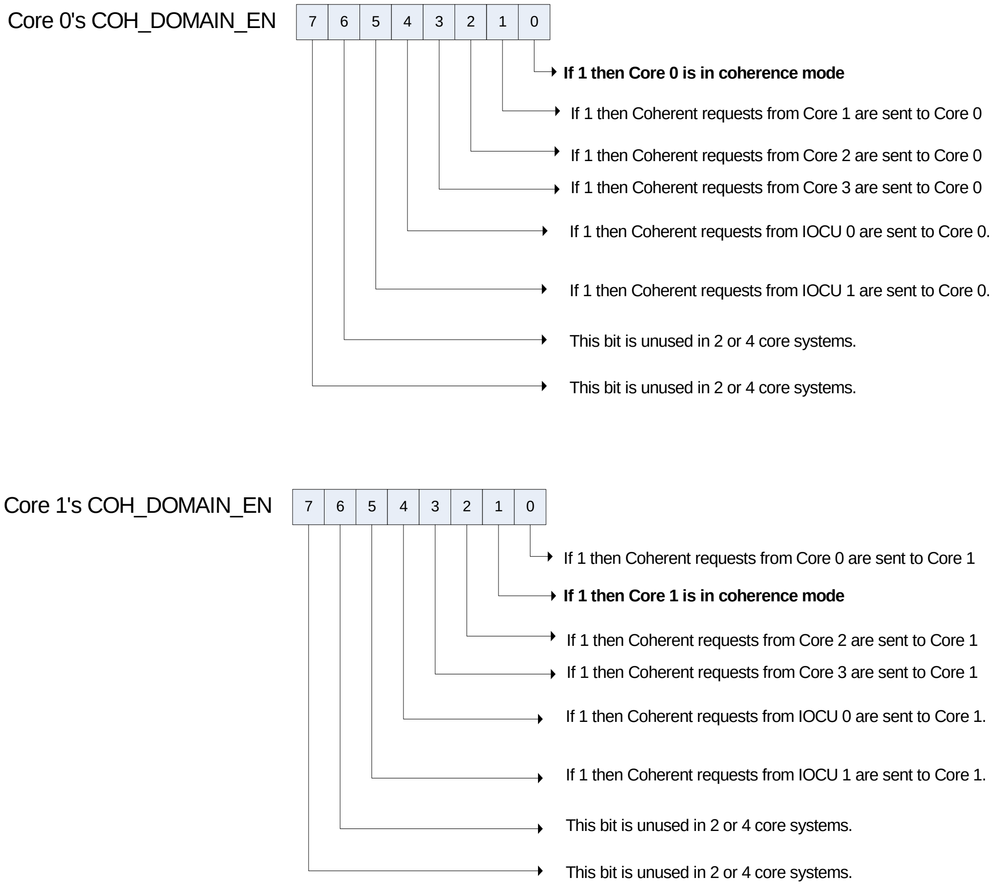
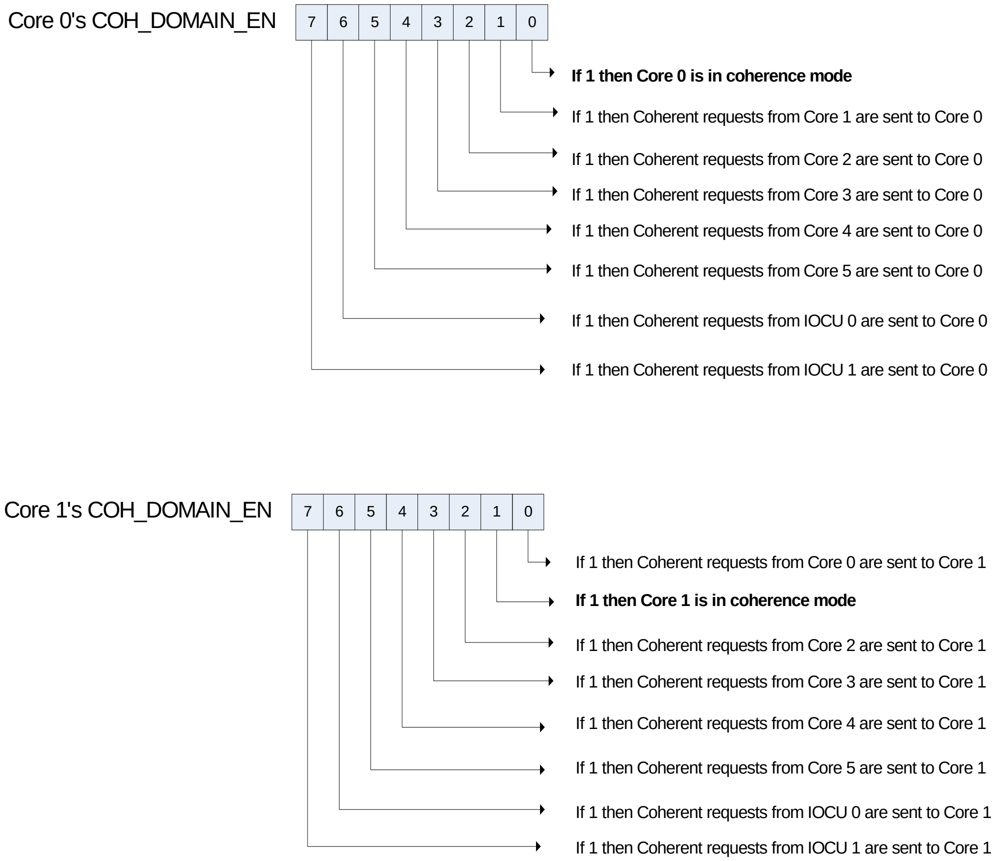

Core 0's COH_DOMAIN_EN
7
6
5
4
3
2
1
0
If 1 then Core 0 is in coherence mode
If 1 then Coherent requests from Core 1 are sent to Core 0
If 1 then Coherent requests from Core 2 are sent to Core 0
If 1 then Coherent requests from Core 3 are sent to Core 0
If 1 then Coherent requests from IOCU 0 are sent to Core 0.
If 1 then Coherent requests from IOCU 1 are sent to Core 0.
This bit is unused in 2 or 4 core systems.
This bit is unused in 2 or 4 core systems.
Core 1's COH_DOMAIN_EN
7
6
5
4
3
2
1
0
If 1 then Coherent requests from Core 0 are sent to Core 1
If 1 then Core 1 is in coherence mode
If 1 then Coherent requests from Core 2 are sent to Core 1
If 1 then Coherent requests from Core 3 are sent to Core 1
If 1 then Coherent requests from IOCU 0 are sent to Core 1.
If 1 then Coherent requests from IOCU 1 are sent to Core 1.
This bit is unused in 2 or 4 core systems.
This bit is unused in 2 or 4 core systems.

Figure 8.10 Encoding of COH_DOMAIN_EN Field — 6 Core Package


Core 0's COH_DOMAIN_EN
7
6
5
4
3
2
1
0
If 1 then Core 0 is in coherence mode
If 1 then Coherent requests from Core 1 are sent to Core 0
If 1 then Coherent requests from Core 2 are sent to Core 0
If 1 then Coherent requests from Core 3 are sent to Core 0
If 1 then Coherent requests from Core 4 are sent to Core 0
If 1 then Coherent requests from Core 5 are sent to Core 0
If 1 then Coherent requests from IOCU 0 are sent to Core 0
If 1 then Coherent requests from IOCU 1 are sent to Core 0
Core 1's COH_DOMAIN_EN
7
6
5
4
3
2
1
0
If 1 then Coherent requests from Core 0 are sent to Core 1
If 1 then Core 1 is in coherence mode
If 1 then Coherent requests from Core 2 are sent to Core 1
If 1 then Coherent requests from Core 3 are sent to Core 1
If 1 then Coherent requests from Core 4 are sent to Core 1
If 1 then Coherent requests from Core 5 are sent to Core 1
If 1 then Coherent requests from IOCU 0 are sent to Core 1
If 1 then Coherent requests from IOCU 1 are sent to Core 1

8.2.12 L2-Only SYNC Operation

In previous generation MIPS processors, the execution of a SYNC instruction would cause the entire core pipeline to stall until all read/write requests were completed. This included the L2 pipeline. After all instructions had been completed, a signal was sent to the L2 cache to continue. This caused a sometimes unnecessary stalling of the L2 cache.

The proAptiv core provides a way to perform a SYNC operation on only the L2 cache. The core defines a fixed 4 KB address space for performing L2 only SYNC operations. The base address for the location of this fixed 4 KB segment is programmed using bits 31:12 of the *L2-Only Sync Base* register located at offset 0x0070.

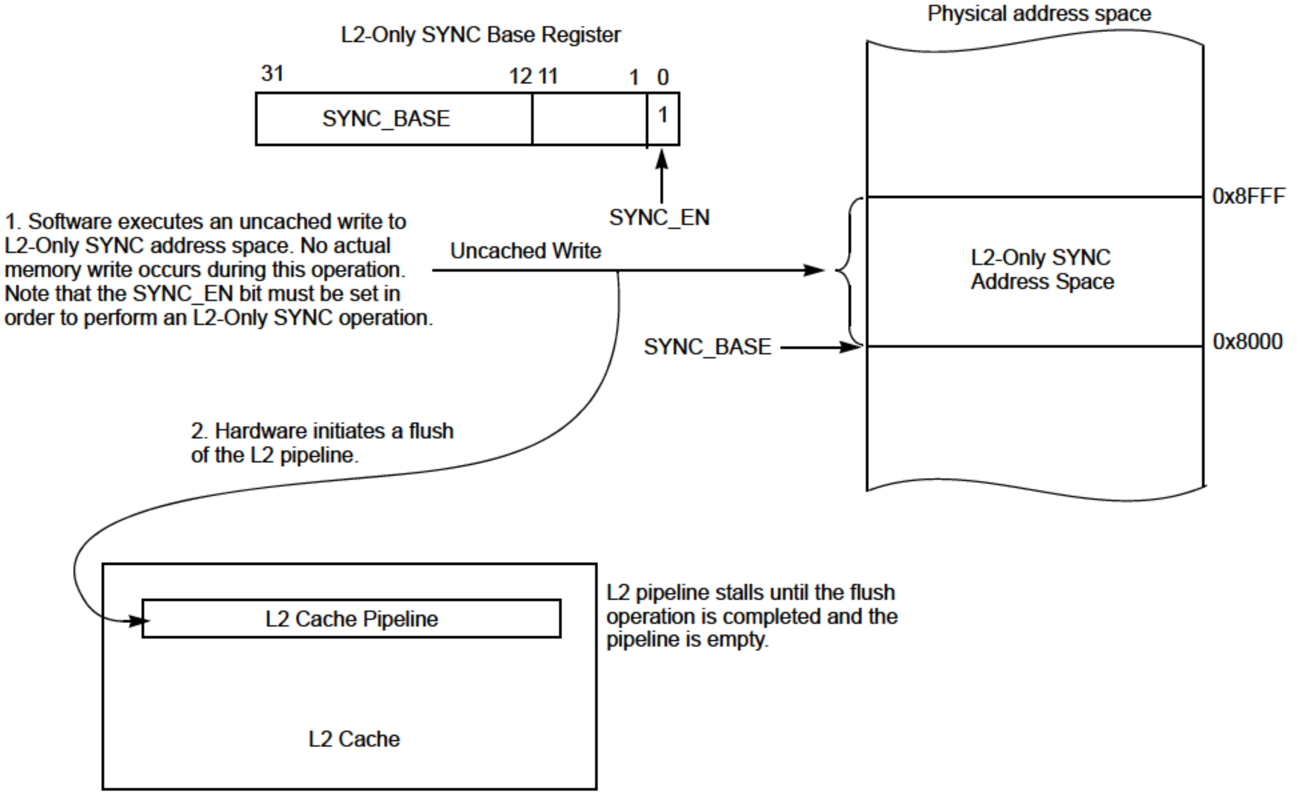
Bit 0 of the *L2-Only Sync Base* register enabled the L2-only SYNC function. If this bit is set, the CM2 treats an uncached write to anywhere within the 4 KB block as an L2-only SYNC. This operation does not write anything to memory, but rather just initiates the L2-only SYNC.

The L2-only SYNC provides a way for the software to ensure that subsequent uncached loads and stores from a core will not pass previous L2 cache operations, such as L2 cacheops.

Note that the L2-Only SYNC is not required, but it can be useful for optimizing performance. Since the L2-Only SYNC operation does not synchronize to the L1 caches, care should be taken to ensure correct system functionality.

As an example of how this operation works, assume the 4 KB block is located at offset address 0x8000 as shown in Figure 8.11.

Figure 8.11 Example of an L2-Only SYNC Operation


L2-Only SYNC Base Register
31
12 11
1 0
SYNC_BASE
1
SYNC_EN
Physical address space
0x8FFF
L2-Only SYNC Address Space
0x8000
SYNC_BASE
Uncached Write
1. Software executes an uncached write to L2-Only SYNC address space. No actual memory write occurs during this operation. Note that the SYNC_EN bit must be set in order to perform an L2-Only SYNC operation.
2. Hardware initiates a flush of the L2 pipeline.
L2 Cache Pipeline
L2 Cache
L2 pipeline stalls until the flush operation is completed and the pipeline is empty.

8.2.13 Handling of Addresses Not Mapped to a Defined Region

The CM2 handles transactions between the core and several devices as described in Figure 8.2.

For addresses that do not map to any of the defined address regions, these transactions can be mapped to either memory or one of the IOCU's as determined by the *CM2_DEFAULT_TARGET* field in bits 1:0 of the *GCR Base* register located at offset 0x0008. The default state of this field is determined by the value of the *SI_CM_Default_Target[1:0]* pins at reset, but can be changed by software at any point. Refer to Section 8.3.2.2, "GCR Base Register (GCR_BASE Offset 0x0008)" for more information on the CM2_TARGET field.

Because programmable regions of the address map are disabled at reset, the value of *SI_CM_Default_Target[1:0]* determines whether the initial boot code upon power-up is fetched from the L2/Memory port or the MMIO port. For systems without an IOCU, *SI_CM_Default_Target[1:0]* should be set to 0 (memory) so that all non-coherent requests are routed to memory.

8.2.14 Setting the Cache Coherency Attributes for Default Memory Transfers

In previous generation MIPS processors, the cache coherency attributes (CCA) for the L1 and L2 caches were configured as one, and the CCA for the L2 cache could not be different from the CCA for the L1 data cache. The proAptiv core provides a CCA override capability that allows the CCA's for the L2 cache to be different from those of the L1 data cache. For example, it may be useful to treat a line as cached in the L1, but uncached in the L2.

The default region determined by the *GCR Base Address* register described in Section 8.2.4 above contains a mechanism for modifying the cache coherency attributes of the base region relative to that of the L1 cache. The attributes are programmed using the *CCA_Override_Enable* (bit 4) and *CCA_Override_Value* (bits 7:5) fields in the *CM2 GCR Base Address Register*. Addresses that do not map to any other region are mapped to the default region.

Any valid CCA value can be programmed into *CCA_Override_Value*, but because the L2 does not process coherent CCAs, a value of CWB (0x5) or CWBE (0x4) is automatically changed to WB (0x3) by the CM2 before being driven on the system memory OCP port.

The various coherency options are shown in Table 8.8. Note that the CCA overrides shown below only affect the L2 cache and not the L1 cache.

Table 8.8 Cache Coherency Attributes

Encoding	Name	Descriptions
0x0	WT	Write through.
0x1	—	Reserved.
0x2	UC	Uncached.
0x3	WB	Writeback, cacheable, non-coherent.
0x4	CWBE	Coherent writeback exclusive. Since the CM2 does not process coherent CCA’s, this encoding automatically maps to WB (0x3).
0x5	CWB	Coherent writeback. Since the CM2 does not process coherent CCA’s, this encoding automatically maps to WB (0x3).
0x6	—	Reserved.
0x7	UCA	Uncached accelerated.

The *CCA_Override_Enable* (bit 4) must be set in order for the *CCA_Override_Value* field to have meaning.

When overriding a CCA value, the CCA used withing the L2 cache and driven to the system memory OCP interface is affected. Otherwise, the functionality of the transaction within the CM2 is based on the original CCA. Transactions that are not routed to the system memory OCP port, such as accesses to GCRs, GIC, CPC, or MMIO are also unaffected by the CCA Override.

8.2.15 In-Flight L1 and L2 Cache Operations

A core has the ability to issue a steady stream of cache operations and can potentially saturate the CM2 resources. To mitigate the possibility of this happening, the CM2 provides a mechanism to limit the number of successive cache transactions by a particular core. This limits a single core from issuing cache operations in rapid succession. The CM2 provides limits for both the L1 cache and the L2 cache via the *Global CM2 Control2* register located at offset address 0x0018. The default limit for successive L2 cache operations is four, meaning that a given core can execute a maximum of four cache operations (bits 19:16). For the L1 cache the limit is six cache operations (bits 3:0).

Setting a value of 0x0 in either of these fields disables this limitation. In this case the CM2 will not limit the number of successive cache operations that can be issued by a single core.

8.2.16 MIPS System Trace

The MIPS System trace is a new feature to the proAptiv Multiprocessing System Multiprocessing System and allows the SoC designer to place signals from their non-probe SoC logic directly into the trace funnel for PDTrace to capture. The logic and registers that controls System Trace are handled by the CM2. Refer to Chapter 8 of the proAptiv Multiprocessing System Multiprocessing System Hardware User's Manual for more information on MIPS System Trace.

8.2.17 Error Processing

The CM2 detects, reports, and handles several types of errors that may be caused by errant software or hardware soft or hard errors. Table 8.9 lists the errors detected by the CM2. The first 7 errors are invalid requests to the GCR, GIC, or MMIO. There are two errors for invalid intervention responses due to inconsistent L1 cache states. And there are 3 errors due to L2 RAM parity errors.

When an error is detected, information that may be useful in debugging the error is captured in the *Global CM2 Error Cause Register* and *Global CM2 Error Address Register*. Refer to Section 8.3.2.8, "Global CM2 Error Cause Register (GCR_ERROR_CAUSE Offset 0x0048)" and Section 8.3.2.9, "Global CM2 Error Address Register (GCR_ERROR_ADDR Offset 0x0050)" for more information.

If these registers already have valid error information and a second error is detected, the error type of the second error is captured in the *CM2 Error Multiple Register*. However, an L2 ram correctable error is overwritten by a 2nd error that is not a second L2 ram correctable error. Refer to Section 8.3.2.10, "Global CM2 Error Multiple Register (GCR_ERROR_MULT Offset 0x0058)" for more information. Note that for the second error, only the error type is captured, not the associated error address.

When the *Global CM2 Error Cause Register* is loaded, an interrupt may be generated if the corresponding bit for that type of error is set in the *Global CM2 Error Mask Register* (see Table 8.24). If the error was generated by a request that requires a response and the corresponding *Global CM2 Error Mask Register* bit is 0, then the CM2 issues an ERROR response. However, if the corresponding *Global CM2 Error Mask Register* bit is 1, then the CM2 issues a normal response and an interrupt will be generated instead.

Table 8.9 CM2 Error Types

CM2_ERROR_ TYPE	Error Name	Description	Action
0	-	Reserved	-
1	*GC_WR_ERR*	Non-Coherent Write of length > 1 to GCR or GIC	Drop Write Signal Interrupt if *CM_ERROR_MASK[1]* = 1
2	*GC_RD_ERR*	Non_Coherent Read of length > 1 to GCR or GIC	No GCR access Return SResp = ERROR if *CM_ERROR_MASK[2]* = 0 Signal Interrupt if CM2_ERROR_MASK[2] = 1
3	*COH_WR_ERR*	Coherent Writeback, Cacheop, or CohWriteInvalidate to GIC, GCR, MMIO	Intervention occurs Signal Interrupt if *CM_ERROR_MASK[3]* = 1
4	*COH_RD_ERR*	Coherent Read to GIC, GCR, MMIO	Intervention occurs After intervention, return SResp = ERROR to the original requestor if *CM_ERROR_MASK[4]* = 0 Signal Interrupt if *CM_ERROR_MASK[4]* = 1

Table 8.9 CM2 Error Types *(continued)*

CM2_ERROR_ TYPE	Error Name	Description	Action
5	*MMIO_WR_ERR*	Write to MMIO from the IOCU (only occurs if *CM_DISABLE_MMIO_LIMIT* = 0)	Drop Write Signal Interrupt if *CM_ERROR_MASK[5]* = 1
6	*MMIO_RD_ERR*	Write to MMIO from the IOCU (only occurs if *CM_DISABLE_MMIO_LIMIT* = 0)	Return SResp = ERROR if *CM_ERROR_MASK[6]* = 0 Signal Interrupt if *CM_ERROR_MASK[6]* = 1
17	*INTVN_WR_ERR*	Request does not require a response and: One core responded with M and one or more cores responded with E, or S or One core responded with E and one or more cores responded with S or Multiple cores responded with data	If multiple M or E responses then data from core with lowest port ID is used. Signal Interrupt if *CM_ERROR_MASK[17]* = 1
18	*INTVN_RD_ERR*	Request requires a response and: One core responded with M and one or more cores responded with E, or S or One core responded with E and one or more cores responded with S or Multiple cores responded with data	If multiple M or E responses then data from core with lowest port ID is used. Return SResp = ERROR if *CM_ERROR_MASK[18]* = 0 Signal Interrupt if *CM_ERROR_MASK[18]* = 1
24	*L2_RD_UNCORR*	Request requires a response and: an uncorrectable parity/ECC error occurred during an access to an L2 RAM	Signal Interrupt if *CM_ERROR_MASK[24]* = 1
25	*L2_WR_UNCORR*	Request does not require a response and: an uncorrectable parity/ECC error occurred during an access to an L2 RAM	Signal Interrupt if *CM_ERROR_MASK[25]* = 1
26	*L2_CORR*	A correctable parity/ECC error occurred during an access to an L2 RAM	Signal Interrupt if *CM_ERROR_MASK[26]* = 1

When an error occurs, hardware updates the read-only CM2_ERROR_TYPE field in bits 31:27 of the Global Config register with one of the values listed in Table 8.9 above. Refer to Section 8.3.2.1 “Global Config Register (GCR_CONFIG Offset 0x0000)” for more information. When this field is written, hardware also updates the 27-bit ERROR_INFO field that provides additional information about the error. The organization of this field varies depending on the value in the CM2_ERROR_TYPE field.

8.2.17.1 Error Codes 1 - 15

If the decimal value in the CM2_ERROR_TYPE field is between 1 and 15, the ERROR_INFO field in the *Global CM2 Error Cause* register is organized as shown in Table 8.10.

Table 8.10 State of ERROR_INFO Field for Error Types 1 through 15

Bits	Meaning
26:18	Reserved.
17:15	CCA
14:12	Target Region (0: MEM, 1:GCR, 2: GIC, 3: MMIO, 5: CPC)
11:7	OCP MCmd (see Table 8.11)
6:3	Source TagID
2:0	Source Port

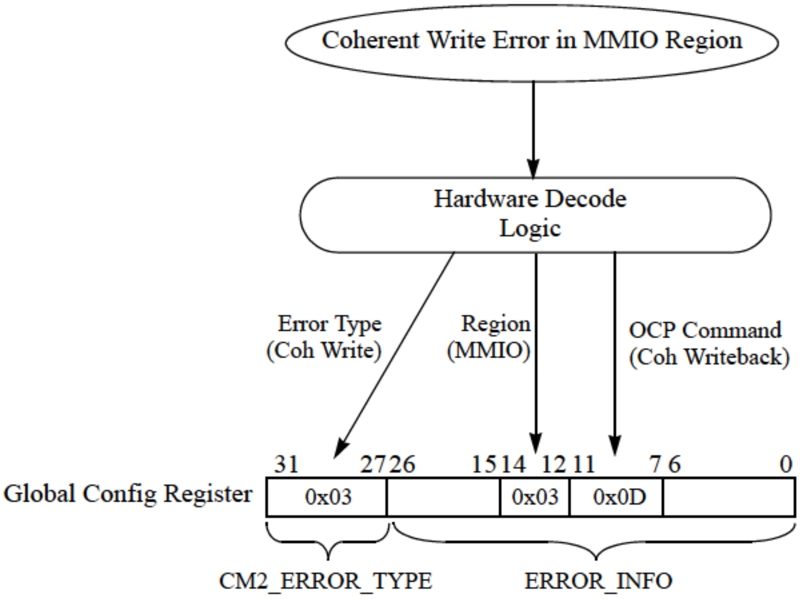
As shown in the above table, the *OCP MCmd* field in bits 11:7 is further encoded as shown in Table 8.11 below.

Table 8.11 MCmd (Bits 11:7) Encoding for CM2_ERROR_INFO

MCmd Encoding	Description
0x01	Legacy Write
0x02	Legacy Read
0x08	Coherent Read Own
0x09	Coherent Read Share
0x0A	Coherent Read Discard
0x0B	Coherent Ready Share Always
0x0C	Coherent Upgrade
0x0D	Coherent Writeback
0x10	Coherent Copyback
0x11	Coherent Copyback Invalidate
0x12	Coherent Invalidate
0x13	Coherent Write Invalidate
0x14	Coherent Completion Sync

Consider the example where a coherent write error occurs to the MMIO region during a coherent writeback operation. In this case, the *Global Config* register would be programmed by hardware as follows:

Figure 8.12 Example of a Coherent Write Error to MMIO



8.2.17.2 Error Codes 16 - 23

If the decimal value in the CM2_ERROR_TYPE field is between 16 and 23, the ERROR_INFO field in the *Global Config* register is organized as shown in Table 8.12.

Table 8.12 State of ERROR_INFO Field for Error Types 16 through 23

Bit	Meaning
26:21	Reserved
20:19	Coherent state from core 3 (see Table 8.13)
18	Intervention SResp from core 3 (see Table 8.14)
17:16	Coherent state from core 2 (see Table 8.13)
15	Intervention SResp from core 2 (see Table 8.14)
14:13	Coherent state from core 1 (see Table 8.13)
12	Intervention SResp from core 1 (see Table 8.14)
11:10	Coherent state from core 0 (see Table 8.13)
9	Intervention SResp from core 0 (see Table 8.14)
8	Request was from a Store Conditional
7:3	OCP MCmd (see Table 8.11)
2:0	Source port

Note that for each of the coherent state errors in Table 8.12 (bits 20:19, 17:16, 14:13, and 11:10), the encoding for these fields is shown in Table 8.13.

Table 8.13 Coherent State Values for Error Types 16 through 23

Encoding	Meaning
0	Invalid
1	Shared
2	Modified
3	Exclusive

For each of the Intervention SResp errors in Table 8.12 (bits 18, 15, 12, and 9), the encoding for these bits is shown in Table 8.14.

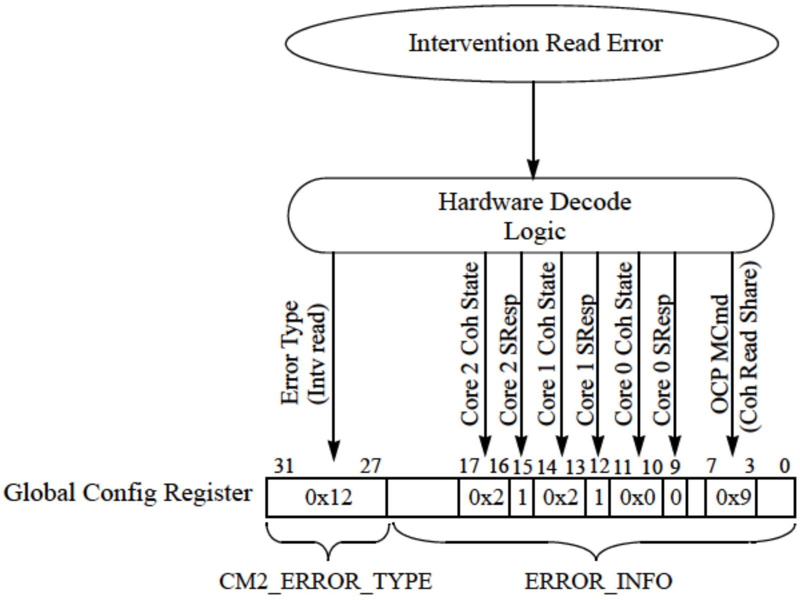
Table 8.14 Intervention SResp Values for Error Type 16 to 23

Encoding	Meaning
0	OK
1	Data (DVA)

Bits 7:3 of the ERROR_INFO field are encoded the same as those shown in Table 8.11.

Consider the example where a core issues a coherent read, and both cores 1 and 2 respond with modified data. In this case, the *Global Config* register would be programmed by hardware as follows:

Figure 8.13 Example of a Intervention Read Error to MMIO



8.2.17.3 Error Codes 24 - 26

If the decimal value in the *CM2_ERROR_TYPE* field is between 24 and 26, the *ERROR_INFO* field in the *Global Config* register is organized as shown in Table 8.15.

Table 8.15 State of ERROR_INFO Field for Error Types 24 to 26

Bit	Meaning
26:24	Reserved (zero)
23	Multiple Uncorrectable
22:18	Instruction[4:0] associated with the error see Table 8.16
17:16	Array type[1:0]: 00 = None 01 = Tag RAM single/double ECC error 10 = Data RAM single/double ECC error 11 = WS RAM uncorrectable dirty parity
15:12	DWord[3:0] with error, Array type = 2 only
11:9	Way[2:0] associated with the error
8	Multi-way error for Tag or WS RAM
7:0	Syndrome associated with Tag or WS way, or Syndrome associated with Data DWord

For each of the errors types 24 - 26 listed in Table 8.9, the instruction associated with the error is encoded into bits 22:18 of the ERROR_INFO field as shown in Table 8.15. The encoding for these bits is shown in Table 8.16 below.

Table 8.16 Instructions for Error Type 24 to 26

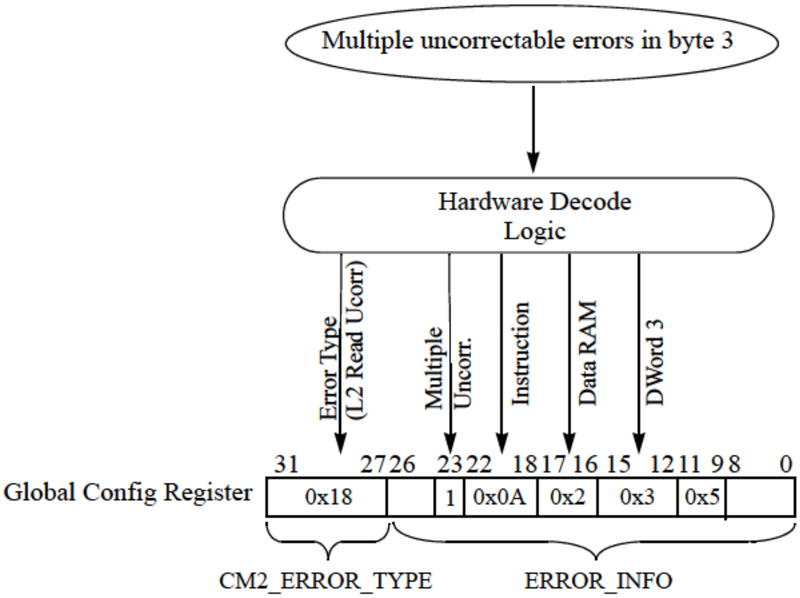
Bit	Meaning
0x00	L2_NOP
0x01	L2_ERR_CORR
0x02	L2_TAG_INV
0x03	L2_WS_CLEAN
0x04	L2_RD_MDYFY_WR
0x05	L2_WS_MRU
0x06	L2_EVICT_LN2
0x08	L2_EVICT
0x09	L2_REFL
0x0A	L2_RD
0x0B	L2_WR
0x0C	L2_EVICT_MRU
0x0D	L2_SYNC
0x0E	L2_REFL_ERR
0x10	L2_INDX_WB_INV
0x11	L2_INDX_LD_TAG

Table 8.16 Instructions for Error Type 24 to 26 *(continued)*

Bit	Meaning
0x12	L2_INDX_ST_TAG
0x13	L2_INDX_ST_DATA
0x14	L2_INDX_ST_ECC
0x18	L2_FTCH_AND_LCK
0x19	L2_HIT_INV
0x1A	L2_HIT_WB_INV
0x1B	L2_HIT_WB

Consider the example of multiple uncorrectable errors in DWord 3, way 5 of the Data RAM during an *L2 Read* instruction. In this case, the *Global Config* register would be programmed by hardware as follows:

Figure 8.14 Multiple Uncorrectable Errors to Byte 3 of the Data RAM During an L2 Hit Writeback Instruction



8.2.18 Custom GCR Implementation

The CM2 provides the ability for the user to implement a 64 KB block of custom registers that can be used to control system level functions. These registers are defined by the user and then instantiated into the design. The CM2 provides two global registers to handle the implementation of customer registers: the *Global Custom Base* register at offset 0x0060, and the *Global Custom Status* register located at offset 0x0068.

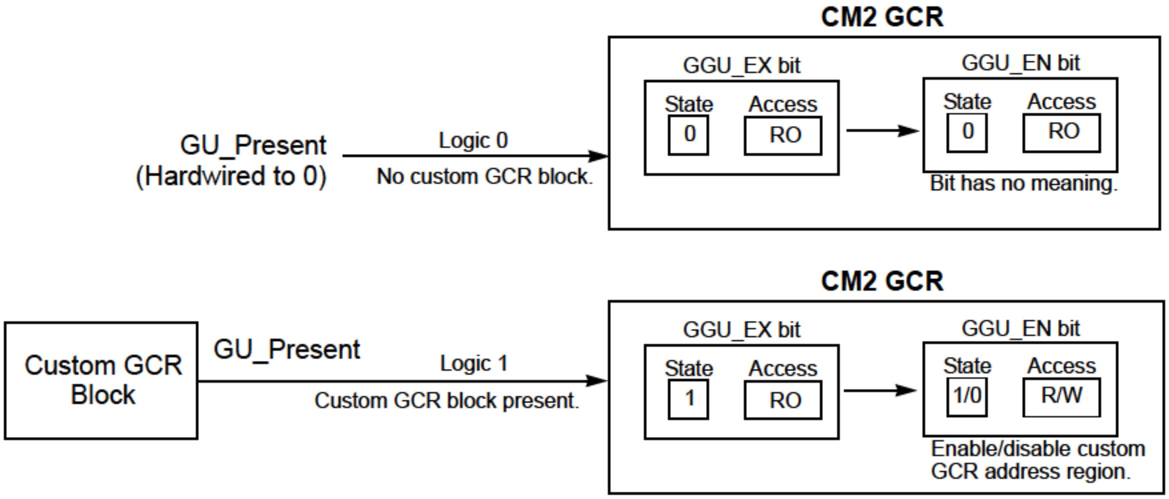
The existence of a custom GCR implementation in the system is selected during IP Configuration. If this option is selected, custom GCR hardware must drive the internal *GU_Present* pin to the CM2. The state of this pin is loaded into the GGU_EX bit in the *Global Custom Status* register. This bit indicates that a custom GCR block is connected to the CM2. Note that *GU_Present* is an internal signal that is an output of the Custom GCR and is connected to the CM2 logic.

If a custom block is implemented, the starting address in memory of the 64 KB block is determined using the 16-bit CUSTOM_BASE field in the *Global Custom Base* register. Note that unlike the configuration of the CM2 Global control registers described in Section 8.2.4, the CUSTOM_BASE field does not have a default base address and this field is undefined at reset. Therefore, it is software's responsibility to program the base address into this field during boot time if a custom GCR block is implemented.

In addition, the selected address region where the registers will reside must be enabled by setting the GGU_EN bit in the *Global Custom Base* register. Note that the accessibility of this bit by software depends on the state of the the *GGU_EX* bit described above. If *GGU_EX* is cleared (zero), indicating that no custom GCR is connected to the CM2, then the *GGU_EN* bit becomes RO and is not accessible by software. If this bit is set, indicating that a custom GCR is connected to the CM2, then the *GGU_EN* bit becomes R/W and is accessible by software.

This concept is described in Figure 8.15 below.

Figure 8.15 Relationship Between the CM_Present Signal and the GGU_EX and GGU_EN Bits at Reset



Note that, depending on the user's implementation, the custom GCR may handle 64-bit reads/writes (unlike the normal GCR which only handles 32-bit accesses). For more information on this feature, contact MIPS Customer Support.

8.2.19 Attribute-Only Regions

The CM2 provides four standard variable-size regions as described in Section 8.2.5, "Address Regions", as well as four additional attribute-only regions. The attribute only regions allows the cache coherency attributes for that region to be modified, but they cannot be used to select between memory and I/O as the target.

In a situation where all of the standard variable size regions have been allocated, the attribute-only regions can be used to override the cache coherency attributes for that memory region. For example, all four attribute-only regions can be mapped to a single IOCU.

The CM2 uses four sets of base/mask registers to manage up to four attribute-only regions. The Base registers described in Section 8.3.5.1, "CM2 Attribute-Only Region [0 - 3] Base Address Registers (GCR_REGn_ATTR_BASE Offsets 0x0190, 0x01A0, 0x0210, 0x0220)" contain the base address in memory for each region. The Mask registers described in Section 8.3.5.2, "CM Attribute-Only Region[0 - 3] Address Mask

Registers (GCR_REGn_ATTR_MASK Offsets 0x0198, 0x1A8, 0x218, 0x228)" contain the size of the region and the CCA override information.

These registers are shown starting at offset address 0x0190 in Table 8.17 below:

8.3 Global Control Block

8.3.1 Global Control Block Address Map

All registers in the Global Control Block are 32 bits wide and should only be accessed using 32-bit uncached load/ stores. Reads from unpopulated registers in the GCR address space return 0x0, and writes to those locations are silently dropped without generating any exceptions.

Table 8.17 Global Control Block Register Map (Relative to Global Control Block offset)

Register Address	Name	Type	Description
0x0000	Global Config Register (*GCR_CONFIG*)	R	Indicates the number of Processor cores, number of interrupts, number of IOCUs, etc.
0x0008	GCR Base Register (*GCR_BASE*)	R/W	Base of the Control Register Space
0x0010	Global CM2 Control Register (*GCR_CONTROL*)	R/W	Control bits for the Coherence Manager
0x0018	Global CM2 Control2 Register (*GCR_CONTROL*2)	R/W	More Control bits for the Coherence Manager
0x0020	Global CSR Access Privilege Register (*GCR_ACCESS*)	R/W	Controls which Cores can modify the GCR Registers
0x0030	GCR Revision Register (*GCR_REV*)	R	RevisionID of the GCR hardware
0x0040	Global CM2 Error Mask Register (*GCR_ERROR_MASK*)	R/W	Controls what Errors are reported as Interrupts
0x0048	Global CM2 Error Cause Register (*GCR_ERROR_CAUSE*)	R/W	Captures info when an Error occurs within the CM2
0x0050	Global CM2 Error Address Register (*GCR_ERROR_ADDR*)	R/W	Captures address which caused the CM2 error.
0x0058	Global CM2 Error Multiple Register (*GCR_ERROR_MULT*)	R/W	Captures information for subsequent CM2 errors.
0x0060	GCR Custom Base Register (*GCR_CUSTOM_BASE*)	R/W	Base address of the custom user-defined 64KB control register space.
0x0068	GCR Custom Status Register (*GCR_CUSTOM_STATUS*)	R/W	Existence and status of the custom user-defined GCR
0x0070	Global L2 only Sync Register (*GCR_L2_ONLY_SYNC_BASE*)	R/W	Base address of the L2 only Sync 4KB address space
0x0080	Global Interrupt Controller Base Address Register (*GCR_GIC_BASE*)	R/W	GIC Base Address

Table 8.17 Global Control Block Register Map (Relative to Global Control Block offset)

Register Address	Name	Type	Description
0x0088	Cluster Power Controller Base Address Register (*GCR_CPC_BASE*)	R/W	CPC Base Address
0x0090	CM2 Region0 Base Address Register (*GCR_REG0_BASE*)	R/W	Address Region0 Base Address This register is present only when the IOCU is present
0x0098	CM2 Region0 Address Mask Register (*GCR_REG0_MASK*)	R/W	Address Region0 Size and Destination This register is present only when the IOCU is present
0x00A0	CM2 Region1 Base Address Register (*GCR_REG1_BASE*)	R/W	Address Region1 Base Address This register is present only when the IOCU is present
0x00A8	CM2 Region1 Address Mask Register (*GCR_REG1_MASK*)	R/W	Address Region1 Size and Destination This register is present only when the IOCU is present
0x00B0	CM2 Region2 Base Address Register (*GCR_REG2_BASE*)	R/W	Address Region2 Base Address This register is present only when the IOCU is present
0x00B8	CM2 Region2 Address Mask Register (*GCR_REG2_MASK*)	R/W	Address Region2 Size and Destination This register is present only when the IOCU is present
0x00C0	CM2 Region3 Base Address Register (*GCR_REG3_BASE*)	R/W	Address Region3 Base Address This register is present only when the IOCU is present
0x00C8	CM2 Region3 Address Mask Register (*GCR_REG3_MASK*)	R/W	Address Region3 Size and Destination This register is present only when the IOCU is present
0x00D0	Global Interrupt Controller Status Register (*GCR_GIC_STATUS*)	R	Existence and status of GIC
0x00E0	Cache Revision Register (*GCR_CACHE_REV*)	R	Revision of cache attached to the coherent Cluster.
0x00F0	Cluster Power Controller Status Register (*GCR_CPC_STATUS*)	R	Existence and status of CPC.
0x0190	CM Attribute-Only Region0 Base Address Register (*GCR_REG0_ATTR_BASE*)	R/W	Attribute Only Region.
0x0198	CM Attribute-Only Region0 Address Mask Register (*GCR_REG0_ATTR_MASK*)	R/W	Attribute Only Region.
0x01A0	CM Attribute-Only Region1 Base Address Register (*GCR_REG0_ATTR_BASE*)	R/W	Attribute Only Region.
0x01A8	CM Attribute-Only Region1 Address Mask Register (*GCR_REG1_ATTR_MASK*)	R/W	Attribute Only Region.
0x0200	IOCU Revision Register (*GCR_IOCU1_REV*)	R	Revision of IOCU
0x0210	CM Attribute-Only Region2 Base Address Register (*GCR_REG2_ATTR_BASE*)	R/W	Attribute Only Region.
0x0218	CM Attribute-Only Region2 Address Mask Register (*GCR_REG2_ATTR_MASK*)	R/W	Attribute Only Region.

Table 8.17 Global Control Block Register Map (Relative to Global Control Block offset)

Register Address	Name	Type	Description
0x0220	CM Attribute-Only Region3 Base Address Register (*GCR_REG3_ATTR_BASE*)	R/W	Attribute Only Region.
0x0228	CM Attribute-Only Region3 Address Mask Register (*GCR_REG3_MASK*)	R/W	Attribute Only Region.
All Others	RESERVED	-	For Future Extensions

8.3.2 CM2 Configuration Registers

This section describes the CM2 configuration registers, including control, error and mask, revision, and custom-GCR registers.

8.3.2.1 Global Config Register (GCR_CONFIG Offset 0x0000)

This register provides information on the overall system configuration. These fields are read-only and their reset state is determined at IP configuration time. Refer to Section 8.2.5, "Address Regions" for more information on how the address regions are used.

Figure 8.16 Global Configuration Register Format

31 20	19 16	15 12	11 8	7 0
R	ADDR_REGIONS	R	NUMIOCU	PCORES

Table 8.18 Global Config Register Descriptions

<table>
<tr><th>Name</th><th>Bits</th><th>Description</th><th>Read/ Write</th><th>Reset State</th></tr>
<tr><td>RESERVED</td><td>31:20</td><td>Reserved, Read as 0x0. Writes ignored. Must be written with a value of 0x0.</td><td>R</td><td>-</td></tr>
<tr><td><i>ADDR_REGIONS</i></td><td>19:16</td><td>Number of address regions. Total number of CM2 Address Regions. Note: only 0, 4, 6, or 8 address regions are currently supported. All other encoded values not listed below are reserved.
.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0x0</td><td>0 Address Regions - no IOCU</td></tr>
<tr><td>0x4</td><td>4 Address Regions - standard</td></tr>
<tr><td>0x6</td><td>6 Address Regions - 4 standard + 2 Attribute Only</td></tr>
<tr><td>0x8</td><td>8 Address Regions - 4 standard + 4 Attribute Only</td></tr>
</table>
</td><td>R</td><td>IP Configuration Value</td></tr>
<tr><td>RESERVED</td><td>15:12</td><td>Read as 0x0. Writes ignored. Must be written with a value of 0x0.</td><td>R</td><td>-</td></tr>
</table>

Table 8.18 Global Config Register Descriptions

<table>
<tr><th>Name</th><th>Bits</th><th>Description</th><th>Read/ Write</th><th>Reset State</th></tr>
<tr><td><em>NUMIOCU</em></td><td>11:8</td><td>Total number of IOCUs in the system. Note: only 0, 1, or 2 IOCU’s are currently supported.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0x0</td><td>0 IOCU</td></tr>
<tr><td>0x1</td><td>1 IOCUs</td></tr>
<tr><td>0x2</td><td>2 IOCUs</td></tr>
<tr><td>0x3 - 0xF</td><td>Reserved</td></tr>
</table></td><td>R</td><td>IP Configuration Value</td></tr>
<tr><td><em>PCORES</em></td><td>7:0</td><td>Total number of proAptiv cores in the system <em>not</em> including the IOCUs. All values not shown are reserved.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0x00</td><td>1 core</td></tr>
<tr><td>0x01</td><td>2 cores</td></tr>
<tr><td>0x02</td><td>3 cores</td></tr>
<tr><td>0x03</td><td>4 cores</td></tr>
<tr><td>0x04</td><td>Reserved</td></tr>
<tr><td>0x05</td><td>6 cores</td></tr>
<tr><td colspan="2">No 5-core option</td></tr>
</table></td><td>R</td><td>IP Configuration Value</td></tr>
</table>

8.3.2.2 GCR Base Register (GCR_BASE Offset 0x0008)

Within the physical address space, the location of the GCR is set by the *GCR_BASE* register. The MIPS default power-up value produces the physical address 0x1FBF_8000. A different default value may be specified at IP configuration time.

Refer to Section 8.2.4, "Setting the CM2 Register Block Base Address" and Section 8.2.14, "Setting the Cache Coherency Attributes for Default Memory Transfers" for more information on how this register is used.

Figure 8.17 GCR Base Register Format

31 15	14 8	7 5	4	3 2	1 0
GCR_BASE	R	CCA	CCAEN	R	CM2_TARGET

Table 8.19 GCR Base Register Descriptions

<table>
<tr><th>Name</th><th>Bits</th><th>Description</th><th>Read/ Write</th><th>Reset State</th></tr>
<tr><td><i>GCR_BASE</i></td><td>31:15</td><td>This field sets the base address of the 32KB GCR block of the proAptiv MPS.
This register has a fixed value after reset if configured as Read-Only (an IP Configuration Option).</td><td>R or R/W (IP Configuration)</td><td>IP Configuration Value
MIPS Default: 0x1FBF_8</td></tr>
<tr><td>RESERVED</td><td>14:8</td><td>Reads as 0x0. Must be written with a value of 0x0.</td><td>R</td><td>0</td></tr>
<tr><td><i>CCA</i></td><td>7:5</td><td><i>CCA default override value.</i> Used in conjunction with <i>CCAEN</i> to force the Cache Coherence Attribute (CCA) value for transactions on the system memory OCP. See <i>CCAEN</i> field.
<table>
<tr><th>Encoding</th><th>Name</th><th>Description</th></tr>
<tr><td>0x0</td><td>WT</td><td>Write Through</td></tr>
<tr><td>0x1</td><td>-</td><td>Reserved</td></tr>
<tr><td>0x2</td><td>UC</td><td>Uncached</td></tr>
<tr><td>0x3</td><td>WB</td><td>Writeback, cacheable, noncoherent</td></tr>
<tr><td>0x4</td><td>CWBE</td><td>Mapped to WB</td></tr>
<tr><td>0x5</td><td>CWB</td><td>Mapped to WB</td></tr>
<tr><td>0x6</td><td>-</td><td>Reserved</td></tr>
<tr><td>0x7</td><td>UCA</td><td>Uncached Accelerated</td></tr>
</table>
</td><td>R/W</td><td>0</td></tr>
<tr><td><i>CCAEN</i></td><td>4</td><td>If <i>CCA_DEFAULT_OVERRIDE_ENABLE</i> is set to 1 and <i>CM2_DEFAULT_TARGET</i> is set to Memory, then transactions with addresses that do not map to any region will have a CCA value set to <i>CCA_DEFAULT_OVERRIDE_VALUE</i> when driven to system memory.</td><td>R/W</td><td>0</td></tr>
<tr><td>RESERVED</td><td>3:2</td><td>Read as 0x0. Must be written with a value of 0x0.</td><td>-</td><td>0x0</td></tr>
</table>

Table 8.19 GCR Base Register Descriptions *(continued)*

<table>
<tr><th>Name</th><th>Bits</th><th>Description</th><th>Read/ Write</th><th>Reset State</th></tr>
<tr><td><i>CM2_DEFAULT_ TARGET</i></td><td>1:0</td><td>Determines the target device for addresses which do not match any address map entry.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Memory</td></tr>
<tr><td>1</td><td>Reserved</td></tr>
<tr><td>2</td><td>IOCU 0</td></tr>
<tr><td>3</td><td>IOCU 1</td></tr>
</table>
Only used for hardware I/O-Coherent systems.</td><td>R/W</td><td>Value of signal <i>SI_CM_Default_ Target[1:0]</i></td></tr>
</table>

8.3.2.3 Global CM2 Control Register (GCR_CONTROL Offset 0x0010)

Figure 8.18 Global CM2 Control Register Format

31 17	16
R	SYNCCTL

15 8	7 6	5	4	3	2	1	0
R	U	SYNCDIS	IVU_EN	SHST_EN	PARK_EN	MMIO_LIMIT_DIS	SPEC_READ_EN

Table 8.20 Global CM2 Control Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
RESERVED	31:17	Read as 0x0. Must be written with a value of 0x0.	-	0x0
SYNCCTL	16	Determines SYNC behavior when a SYNC level 0x0 is executed by a core. *SyncCtl* = 1 means Sync0 generates a memory sync *SyncCtl* = 0 means Sync0 generates an intervention sync	RW	0x0
RESERVED	15:8	Read as 0x0. Must be written with a value of 0x0.	R	0x0
UNUSED	7:6	These bits are currently unused. When writing to this register, software should assign a value of 2’b00 to this field.	R/W	0x0
SYNCDIS	5	SYNC transmit disable. Set to 1 to disable the propagation of SYNC transactions on the system memory port. This has the same effect as deasserting *SI_SyncTxEn*. Setting to 0 makes the propagation of SYNC transactions on the system memory port dependent solely on the state of *SI_SyncTxEn*. Refer to the pin descriptions chapter in the proAptiv Hardware User’s Manual for more information on this pin.	RW	0x0

Table 8.20 Global CM2 Control Register Descriptions *(continued)*

Name	Bits	Description	Read/ Write	Reset State
IVU_EN	4	Stall until interventions are completed. Set to 1 to stall serialization when a core's clock is stopping or is being powered down by the CPC until all previous interventions are complete. Set to 0 for no stalling of serialization when a core is going offline.	RW	0x0
SHST_EN	3	Force coherent read data to shared state in L1 data cache. If set to 1 then Coherent Read Data is always installed in the Level 1 cache of the requesting proAptiv Multiprocessing System core in the SHARED state. If set to 0 then Coherent Read Data may be installed in the Level 1 cache in the SHARED state (if the data coexists in other Level 1 caches) or EXCLUSIVE (if the data does not coexist in other Level 1 caches).	RW	0x0
PARK_EN	2	I/O port parking enable. If set to 1 and the *SI<iocu>_CMP_IOC_ParkEn* signal is 1, then I/O Port Parking is enabled for the corresponding IOCU. I/O Port parking is a mechanism where the CM2 only serializes requests from the IOCU for some period of time. If set to 0 or *SI<iocu>_CMP_IOC_ParkEn* signal is 0, then the I/O Port Parking is disabled for the corresponding IOCU. This bit has no effect in systems without an IOCU (i.e., they are not hardware I/O coherent).	RW	0x0
MMIO_LIMIT_DIS	1	Limit requests to memory-mapped I/O. If set to 0, the CM2 avoids deadlock in systems with hardware I/O coherence by limiting requests issued to Memory-Mapped I/O. An MMIO request will be selected for serialization only if the previous request and write data (if applicable) has been accepted by the IOCU. If set to 1, MMIO requests are not limited and therefore deadlock may occur in systems with hardware I/O coherence unless avoided by some other mechanism. This bit has no effect in systems without an IOCU (i.e., they are not hardware I/O coherent) because there are no MMIO ports and therefore the limit does not apply.	RW	0x0

Table 8.20 Global CM2 Control Register Descriptions *(continued)*

Name	Bits	Description	Read/ Write	Reset State
SPEC_READ_EN	0	Speculative coherent read enable. If set to 1, the CM2 may speculatively read memory for a coherent read before the intervention for that read has completed. Performance is improved by reading memory in parallel with the intervention. If set to 0, the CM2 will never issue speculative reads to memory.	R/W	0x1

8.3.2.4 Global CM2 Control2 Register (GCR_CONTROL2 Offset 0x0018)

This register sets limits on how many consecutive cache operations are allowed to the L1 and L2 caches. Refer to Section 8.2.15, "In-Flight L1 and L2 Cache Operations" for more information on how this register is used.

Figure 8.19 Global CM2 Control2 Register Format

31 20	19 16	15 4	3 0
R	L2_CACEOP_LIMIT	R	L1_CACEOP_LIMIT

Table 8.21 Global CM2 Control2 Register

Name	Bits	Description	Read/ Write	Reset State
RESERVED	31:20	Read as 0x0. Writes ignored. Must be written with a value of 0x0.	-	0x0
L2_CACHEOP_LIMIT	19:16	L2 CacheOp transaction limit. The total number of L2 CacheOp transactions allowed by the CM2 serialization arbiter to be simultaneously in-flight. An L2 CacheOp is defined as any transaction with MAddrSpace = 0b001 or 0b010. In this context, an L2 CacheOp transaction is considered in-flight when it is selected for serialization by the CM2 until the request is issued on the CM2's system memory OCP Port. Setting a value of 0x0 disables the limit (i.e., the CM2 serialization arbiter will not explicitly limit the number of in-flight L12 CacheOps). Setting a value of 0x1 allows only a single in-flight L2 CacheOp. Setting a value of 0x2 allows two in-flight L2 CacheOps, etc... The purpose of this limit is to avoid the case where one or more cores substantially impact the performance of other cores by issuing a rapid succession of L2 CacheOps.	R/W	0x4
RESERVED	15:4	Read as 0x0. Writes ignored. Must be written with a value of 0x0.	-	0x0

Table 8.21 Global CM2 Control2 Register *(continued)*

Name	Bits	Description	Read/ Write	Reset State
L1_CACHEOP_LIMIT	3:0	L1 CacheOp transaction limit. The total number of L1 CacheOp transactions allowed by the CM2 serialization arbiter to be simultaneously in-flight. A L1 CacheOp is defined as a transaction with MAddrSpace = 0b011 or 0b1xx. In this context, a transaction is considered in-flight when it is selected for serialization by the CM2 until its intervention response is processed by the CM2 (if the cacheOp did not receive a DVA intervention response) or until all intervention data has been received (if the cacheOp received a DVA intervention response). Setting a value of 0x0 disables the limit (i.e., the CM2 serialization arbiter will not explicitly limit the number of in-flight L1 CacheOps). Setting a value of 0x1 allows only a single in-flight L1 CacheOp. Setting a value of 0x2 allows two in-flight L1 CacheOps, etc... The purpose of this limit is to avoid the case where one or more cores substantially impact the performance of other cores by issuing a rapid succession of L1 CacheOps that receive an intervention response of DVA.	R/W	0x6

8.3.2.5 Global CSR Access Privilege Register (GCR_ACCESS Offset 0x0020)

A request can be initiated by either a core or an IOCU. The CM2 allows for a maximum of eight requestors; six cores and two IOCU's. However, these requestors do not have unrestricted access to the CM2 register set and must be granted permission by software via this register. Refer to Section 8.2.2, "Requestor Access to GCR Registers" for more information on how this register is used.

Figure 8.20 Global CSR Access Privilege Register Format

31 8	7 0
R	CM2_ACCESS_EN

Table 8.22 Global CSR Access Privilege Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
RESERVED	31:8	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0x0000_00

Table 8.22 Global CSR Access Privilege Register Descriptions*(continued)*

Name	Bits	Description	Read/ Write	Reset State
CM2_ACCESS_EN	7:0	Requester access to global control registers. Each bit in this field represents a coherent requester. If the bit is set, that requester is able to write to the GCR registers (this includes all registers within the Global, Core-Local, Core-Other, and Global Debug control blocks. The GIC is always writable by all requestors). If the bit is clear, any write request from that requestor to the GCR registers (Global, Core-Local, Core-Other, or Global Debug control blocks) will be dropped.	R/W	0xFF

8.3.2.6 CM2 Revision Register (GCR_REV Offset 0x0030)

Figure 8.21 GCR Revision Register Format

31 16	15 8	7 0
R	MAJOR_REV	MINOR_REV

Table 8.23 GCR Revision Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
RESERVED	31:16	Reads as 0x0. Must be written with a value of 0x0.	R	0x0000
MAJOR_REV	15:8	CM2 Major revision number. This field reflects the major revision of the GCR block. A major revision might reflect the changes from one product generation to another. This value changes based on the processor revision. Refer to the errata sheet of the proAptiv core for the exact value of this field.	R	Preset
MINOR_REV	7:0	CM2 Minor revision number. This field reflects the minor revision of the GCR block. A minor revision might reflect the changes from one release to another. This value changes based on the processor revision. Refer to the errata sheet of the proAptiv core for the exact value of this field.	R	Preset

8.3.2.7 Global CM2 Error Mask Register (GCR_ERROR_MASK Offset 0x0040)

This register is used in conjunction with the *Global CM2 Error Cause* and *Global CM2 Error Address* registers to determine the type of error and the address which caused the error. Refer to Section 8.2.17, "Error Processing" for more information on how this register is used.

Figure 8.22 Global CM2 Error Mask Register Format

31	0
CM2_ERROR_MASK	

Table 8.24 Global CM2 Error Mask Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
CM2_ERROR_MASK	31:0	CM2 Error Mask field. Each bit in this field represents an Error Type. If the bit is set, an interrupt is generated if an error of that type is detected. If the bit is set, the transaction for Read-Type Errors completes with OK response to avoid double reporting of the error. The Error Types that can be captured are implementation-specific.	R/W	0x000A_002A (write errors cause interrupts; read errors provide error response)

8.3.2.8 Global CM2 Error Cause Register (GCR_ERROR_CAUSE Offset 0x0048)

This register is used in conjunction with the *Global CM2 Error Mask* and *Global CM2 Error Address* registers to determine the type of error and the address which caused the error. Refer to Section 8.2.17, "Error Processing" for more information on how this register is used.

Figure 8.23 Global CM2 Error Cause Register Format

31 27	26 0
CM2_ERROR_TYPE	ERROR_INFO

Table 8.25 Global CM2 Error Cause Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
CM2_ERROR_TYPE	31:27	Indicates type of error detected. When *CM2_ERROR_TYPE* is zero, no errors have been detected. When *CM2_ERROR_TYPE* is non-zero, another error will not be reloaded until a power-on reset or this field is written to 0.	R/W	0

Table 8.25 Global CM2 Error Cause Register Descriptions*(continued)*

Name	Bits	Description	Read/ Write	Reset State

ERROR_INFO	26:0	Information about the error.
		If *CM2_ERROR_TYPE* = 1 through 15, see Table 8.10
		if *CM2_ERROR_TYPE* = 16 through 23, see Table 8.12
		if *CM2_ERROR_TYPE* = 24 through 26, see Table 8.15
			R/W	Undefined

8.3.2.9 Global CM2 Error Address Register (GCR_ERROR_ADDR Offset 0x0050)

This register is used in conjunction with the *Global CM2 Error Cause* and *Global CM2 Error Mask* registers to determine the type of error and the address which caused the error. Refer to Section 8.2.17, "Error Processing" for more information on how this register is used.

Figure 8.24 Global CM2 Error Address Register Format

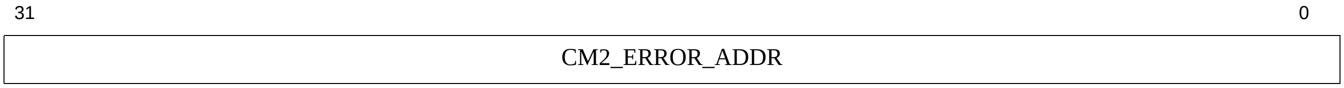


Table 8.26 Global CM2 Error Address Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
CM2_ERROR_ADDR	31:0	Request address which caused error. Loaded when the *Global Error Cause Register* is loaded. Bits 2:0 should always be 0.	R/W	Undefined

8.3.2.10 Global CM2 Error Multiple Register (GCR_ERROR_MULT Offset 0x0058)

The *Global CM2 Error Cause*, *Global CM2 Error Address*, and *Global CM2 Error Mask* registers described above provide information on the type of error, and the address which caused the error. In addition to this information, the proAptiv core also provides a way to determine the type of error should an secondary error occur. However, for the secondary error, only the type of error is logged, not the associated address. This register is used to log the type of secondary error. Refer to Section 8.2.17, "Error Processing" for more information on how this register is used.

Figure 8.25 Global CM2 Error Multiple Register Format

31 5	4 0
R	ERROR_2ND

Table 8.27 Global CM2 Error Multiple Register

Name	Bits	Description	Read/ Write	Reset State
RESERVED	31:5	Reads as 0x0. Must be written with a value of 0x0.	R	0x0000_000
CM2_ERROR_2ND	4:0	Type of second error. Loaded when the *Global CM2 Error Cause Register* has valid error information and a second error is detected.	R/W	5’b0

8.3.2.11 GCR Custom Base Register (GCR_CUSTOM_BASE Offset 0x0060)

This register allows for the implementation of custom registers that are designed by the customer and instantiated into the design at build time. Refer to Section 8.2.18, "Custom GCR Implementation" for more information on how this register is used.

Figure 8.26 Global Custom Base Register Format

31 16	15 1	0
CUSTOM_BASE	R	GGU_EN

Table 8.28 GCR Custom Base Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
CUSTOM_BASE	31:16	This field sets the base address of the 64KB GCR custom user-defined block of the proAptiv Multiprocessing System.	R/W	Undefined
RESERVED	15:1	Reads as 0x0. Must be written with a value of 0x0.	R	0x0000
GGU_EN	0	If this bit is set, the address region for the Custom GCR is enabled. This bit cannot be set to 1 if *GGU_EX* = 0, indicating that a custom GCR is not attached to the CM.	R/W (if *GGU_EX* = 1) R (if *GGU_EX* = 0)	0

8.3.2.12 GCR Custom Status Register (GCR_CUSTOM_STATUS Offset 0x0068)

Refer to Section 8.2.18, "Custom GCR Implementation" for more information on how this register is used.

Figure 8.27 Global Custom Status Register Format

31 1	0
R	GGU_EX

Table 8.29 GCR Custom Status Register Descriptions

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
RESERVED	31:1	Reads as 0x0. Must be written with a value of 0x0.	R	0x0
GGU_EX	0	If this bit is set, the Custom GCR is connected to the CM2. The state of this bit is set based on whether or not this block is implemented at build time as determined by the state of the *GU_Present* signal. If a Custom GCR block is not present, the *GU_Present* pin is driven to 0. If there is a custom GCR block present, then the user must drive GU_Present = 1 inside their custom GCR module.	R	Build time option

8.3.2.13 L2-Only Sync Base Register (GCR_L2_ONLY_SYNC_BASE Offset 0x0070)

The proAptiv core provides a mechanism to execute a SYNC operation to only the L2 cache, without affecting the core. Refer to Section 8.2.12, "L2-Only SYNC Operation" for more information on how this register is used.

Figure 8.28 L2-Only Sync Base Register Format

31 12	11 1	0
SYNC_BASE	R	SYNC_EN

Table 8.30 L2-Only Sync Base Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
SYNC_BASE	31:12	L2-only SYNC base address. This field sets the base address of the 4KB GCR L2 only Sync of the proAptiv MPS.	R/W	Undefined
RESERVED	11:1	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0x0
SYNC_EN	0	L2-only SYNC enable. If this bit is set, the CM2 treats an uncached write request as an L2 only Sync. If set to 0, the CM2 treats the uncached write as a regular uncached request.	R/W	0x0

8.3.3 CM2 Region Address Map Registers

8.3.3.1 Global Interrupt Controller Base Address Register (GCR_GIC_BASE Offset 0x0080)

Figure 8.29 Global Interrupt Controller Base Address Register Format

31 17	16 1	0
GIC_BASE_ADDR	R	GIC_EN

Table 8.31 Global Interrupt Controller Base Address Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
GIC_BASE_ADDR	31:17	Global Interrupt Controller Base Address. This field sets the base address of the 128KB Global Interrupt Controller.	R/W	Undefined
RESERVED	16:1	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0
GIC_EN	0	Global Interrupt Controller Enable. If this bit is set, the address region for the GIC is enabled. This bit can not be set to 1 if *GIC_EX* = 0, indicating that a GIC is not attached to the CM2.	R/W (if *GIC_EX* = 1) R (if *GIC_EX* = 0)	0

8.3.3.2 Cluster Power Controller Base Address Register (GCR_CPC_BASE Offset 0x0088)

Figure 8.30 Cluster Power Controller Base Address Register Format

31 15	14 1	0
CPC_BASE_ADDR	R	CPC_EN

Table 8.32 Cluster Power Controller Base Address Register

Name	Bits	Description	Read/ Write	Reset State
CPC_BASE_ADDR	31:15	This field sets the base address of the 32K Cluster Power Controller.	R/W	Undefined
RESERVED	14:1	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0
CPC_EN	0	If this bit is set, the address region for the CPC is enabled. This bit can not be set if 1 *CPC_EX* = 0, indicating that a CPC is not attached to the CM2.	R/W (if *CPC_EX* = 1) R (if *CPC_EX* = 0)	0

8.3.3.3 CM2 Region [0 - 3] Base Address Register (GCR_REGn_BASE Offsets 0x0090, 0x00A0, 0x00B0, 0x00C0)

Some or all of these registers may be removed during IP configuration. When an IOCU is present, there may be 4 CM2 Address Mask Registers implemented. When no IOCU is present, there may be 0 or 4 CM2 Address Mask Registers. When a register is not present, it is defined as Reserved and Read-Only of 0.

Figure 8.31 CM2 Region [0 - 3] Base Address Register Format

31 16	15 14 0
CM2_REGION_BASE_ADDR	R

Table 8.33 CM2 Region [0 - 3] Base Address Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
CM2_REGION_BASE_ADDR	31:16	CM2 region base address. This field sets the base physical address of the memory region.	R/W	Undefined
RESERVED	15:0	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0

8.3.3.4 CM2 Region [0 - 3] Address Mask Register (GCR_REGn_MASK Offsets 0x0098, 0x00A8, 0x00B8, 0x00C8)

Some or all of these registers may be removed during IP configuration. When an IOCU is present, there may be 4 CM2 Address Mask Registers implemented. When no IOCU is present, there may be 0 or 4 CM2 Address Mask Registers. When a register is not present, it is defined as Reserved and Read-Only of 0.

Figure 8.32 CM2 Region [0-3] Address Mask Register Format

31 15	14 8	7 5	4	3	2	1 0
CM2_REGION_ADDR_MASK	R	CCA_Override_Value	CCA_Override_Enable	R	DROP_L2	CM2_TARGET

Table 8.34 CM2 Region [0 - 3] Address Mask Register Descriptions

<table>
<tr><th>Name</th><th>Bits</th><th>Description</th><th>Read/ Write</th><th>Reset State</th></tr>
<tr><td><i>CM2_REGION_ADDR_MASK</i></td><td>31:16</td><td>This field is used to set the size of the CM2 Region.
This field is used along with its equivalent <i>CM2 Region Base Address Register</i>.
The request address is logically ANDed with the value of this register. The value of the associated <i>Base Address Register</i> is also logically ANDed with the value of this register. If both outputs match, then the request is routed to the CM2 region.
The only allowed values in this register are contiguous sets of leading 0x1's. An 0x1 preceded by a 0x0 is not allowed (e.g., the value of 0xFFF0 is allowed, but the value 0xFFEF is not allowed).</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>RESERVED</td><td>15:8</td><td>Reads as 0x0. Must be written with a value of 0x0.</td><td>R</td><td>0</td></tr>
<tr><td><i>CCA_Override_Value</i></td><td>7:5</td><td>Used with <i>CCA_Override_Enable</i> to force the Cache Coherence Attribute (CCA) value for transactions on the system memory OCP. See <i>CCA_Override_Enable</i> field.
<table>
<tr><th>Encoding</th><th>Name</th><th>CCA</th></tr>
<tr><td>0x0</td><td>WT</td><td>Write Through</td></tr>
<tr><td>0x1</td><td>-</td><td>Reserved</td></tr>
<tr><td>0x2</td><td>UC</td><td>Uncached</td></tr>
<tr><td>0x3</td><td>WB</td><td>WriteBack cacheable, non-coherent,</td></tr>
<tr><td>0x4</td><td>CWBE</td><td rowspan="2">Mapped to WB</td></tr>
<tr><td>0x5</td><td>CWB</td></tr>
<tr><td>0x6</td><td>-</td><td>Reserved</td></tr>
<tr><td>0x7</td><td>UCA</td><td>Uncached Accelerated</td></tr>
</table></td><td>R/W</td><td>0</td></tr>
<tr><td><i>CCA_Override_Enable</i></td><td>4</td><td>If <i>CCA_Override_Enable</i> is set and the <i>CM2_TARGET</i> field is set to Memory (0x1), then transactions with addresses that map to this region will have a CCA value set to <i>CCA_Override_Value</i> when driven to system memory.</td><td>R/W</td><td>0</td></tr>
<tr><td><i>Reserved</i></td><td>3</td><td>Reads as 0x0. Must be written with a value of 0x0.</td><td>R</td><td>0</td></tr>
<tr><td><i>DROP_L2</i></td><td>2</td><td>Drop L2 CacheOp write.
If this bit is set, the CM2 drops the L2 CacheOp write after it has been serialized.
If this bit is cleared, the L2 CacheOp writes behave like a regular L2 CacheOp request.</td><td>R/W</td><td>0</td></tr>
</table>

Table 8.34 CM2 Region [0 - 3] Address Mask Register Descriptions *(continued)*

<table>
<tr><th>Name</th><th>Bits</th><th>Description</th><th>Read/
Write</th><th>Reset State</th></tr>
<tr><td><i>CM2_TARGET</i></td><td>1:0</td><td>Maps this region to the specified device. The IOCU can only be mapped to regions 0 - 3, while memory can be mapped to all regions. .
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0x0</td><td>Disabled</td></tr>
<tr><td>0x1</td><td>Memory</td></tr>
<tr><td>0x2</td><td>IOCU 0</td></tr>
<tr><td>0x3</td><td>IOCU 1</td></tr>
</table></td><td>R/W</td><td>0</td></tr>
</table>

8.3.4 CM2 Status and Revision Registers

This section contains the status registers for the GIC and CPC, and the revision information for the L2 cache.

8.3.4.1 Global Interrupt Controller Status Register (GCR_GIC_STATUS Offset 0x00D0)

Figure 8.33 Global Interrupt Controller Status Register Format

31 1	0
R	GIC_EX

Table 8.35 Global Interrupt Controller Status Register

Name	Bits	Description	Read/ Write	Reset State
RESERVED	31:1	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0
GIC_EX	0	GIC to CM2 connection. If this bit is set, the GIC is connected to the CM2.	R	1

8.3.4.2 Cache Revision Register (GCR_CACHE_REV Offset 0x00E0)

Figure 8.34 Cache Revision Register Format

31 16	15 8	7 0
R	MAJOR_REV	MINOR_REV

Table 8.36 Cache Revision Register

Name	Bits	Description	Read/ Write	Reset State
RESERVED	31:16	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0x0
MAJOR_REV	15:8	This field reflects the major revision of the Cache block inside the CM2.	R	Preset
MINOR_REV	7:0	This field reflects the minor revision of the Cache block inside the CM2.	R	Preset

8.3.4.3 Cluster Power Controller Status Register (GCR_CPC_STATUS Offset 0x00F0)

Figure 8.35 Cluster Power Controller Status Register Format

31 1	0
R	CPC_EX

Table 8.37 Cluster Power Controller Status Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
RESERVED	31:1	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0
CPC_EX	0	This bit is always 1 in the proAptiv core as the CPC is always connected to the CM2.	R	1

8.3.4.4 IOCU Revision Register (GCR_IOCU1_REV Offset 0x0200)

This register gives the existence and revision information for an IOCU which might be connected to the CM2.

Figure 8.36 IOCU Revision Register Format

31 16	15 8	7 0
R	MAJOR_REV	MINOR_REV

Table 8.38 IOCU Revision Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
RESERVED	31:16	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0x0

Table 8.38 IOCU Revision Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
MAJOR_REV	15:8	This field reflects the major revision of the IOCU attached to the CM2. A major revision might reflect the changes from one product generation to another. The value of 0x0 means that no IOCU is attached.	R	Preset
MINOR_REV	7:0	This field reflects the minor revision of the IOCU attached to the CM2. A minor revision might reflect the changes from one release to another.	R	Preset

8.3.5 CM2 Attribute-Only Region Address Map Registers

This section contains the base address and address mask registers for CM2 attribute-only regions 0 through 3. These register have the same functionality as the normal region registers, except they can not be used to map to MMIO vs. memory.

8.3.5.1 CM2 Attribute-Only Region [0 - 3] Base Address Registers (GCR_REGn_ATTR_BASE Offsets 0x0190, 0x01A0, 0x0210, 0x0220)

Some or all of these registers may be removed during IP configuration. These registers are similar to the CM2 Region Address Register except the attribute-only regions can not be used to determine if a request is routed to memory or the IOCU.

Figure 8.37 CM2 Attribute-Only Region [0 - 3] Register Format

31 16	15 0
CM2_REGION_BASE_ADDR	R

Table 8.39 CM2 Attribute-Only Region [0 - 3] Base Address Register Format

Name	Bits	Description	Read/ Write	Reset State
CM2_REGION_BASE_ADDR	31:16	This field sets the base physical address of the memory region.	R/W	Undefined
RESERVED	15:0	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0

8.3.5.2 CM Attribute-Only Region[0 - 3] Address Mask Registers (GCR_REGn_ATTR_MASK Offsets 0x0198, 0x1A8, 0x218, 0x228)

These registers may be removed during IP Configuration. These registers are similar to the CM Region Address Mask registers except they may not be used to route requests to memory or the IOCU.

Figure 8.38 CM2 Attribute Only Region [0-3] Address Mask Register Format

31 15	14 8	7 5	4	3	2	1 0
CM2_REGION_ADDR_MASK	R	CCA_Override_Value	CCA_Override_EN	R	DROP_L2	R

.

Table 8.40 CM Attribute-Only Region [0 - 3] Address Mask Register Descriptions

<table>
<tr><th colspan="2">Register Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/
Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>CM2_REGION_ADDR_MASK</td><td>31:16</td><td>This field is used to set the size of the CM Region.
This field is used along with its equivalent CM Region Base Address Register.
The request address is logically ANDed with the value of this register. The value of the associated Base Address Register is also logically ANDed with the value of this register. If both outputs match, then the request is routed to the CM region.
The only allowed values in this register are contiguous sets of leading 0x1’s. An 0x1 preceded by a 0x0 is not allowed (e.g., the value of 0xfff0 is allowed, but the value 0xffef is not allowed).</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>RESERVED</td><td>15:8</td><td>Reads as 0x0. Must be written with a value of 0x0.</td><td>R</td><td>0</td></tr>
<tr><td>CCA_Override_Value</td><td>7:5</td><td>Used with CCA_Override_Enable to force the Cache Coherence Attribute (CCA) value for transactions on the system memory OCP. See CCA_Override_Enable field.
<table>
<tr><th>Encoding</th><th>Name</th><th>CCA</th></tr>
<tr><td>0x0</td><td>WT</td><td>Write Through</td></tr>
<tr><td>0x1</td><td>-</td><td>Reserved</td></tr>
<tr><td>0x2</td><td>UC</td><td>Uncached</td></tr>
<tr><td>0x3</td><td>WB</td><td>WriteBack cacheable, non-coherent</td></tr>
<tr><td>0x4</td><td>CWBE</td><td rowspan="2">Mapped to WB</td></tr>
<tr><td>0x5</td><td>CWB</td></tr>
<tr><td>0x6</td><td>-</td><td>Reserved</td></tr>
<tr><td>0x7</td><td>UCA</td><td>Uncached Accelerated</td></tr>
</table></td><td>R/W</td><td>0</td></tr>
<tr><td>CCA_Override_Enable</td><td>4</td><td>If set CCA_Override_Enable is set to 1 and CM_TARGET is set to Memory, then transactions with addresses that map to this region will have a CCA value set to CCA_Override_Value when driven to system memory.</td><td>R/W</td><td>0</td></tr>
<tr><td>RESERVED</td><td>3</td><td>Reads as 0x0. Writes ignored. Must be written with a value of 0x0.</td><td>R</td><td>0</td></tr>
</table>

Table 8.40 CM Attribute-Only Region [0 - 3] Address Mask Register Descriptions *(continued)*

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
DROP_L2	2	Set to 1 for the CM to drop L2 CacheOp writes after it has been serialized. If set to 0, the L2 CacheOp writes behaves like a regular L2 CacheOp request.	R/W	0x0
RESERVED	1:0	Reads as 0x0. Must be written with a value of 0x0. Since the attribute-only registers can not be used to map to MMIO vs. memory, this field is not needed and is reserved.	R/W	0x0

8.4 Core-Local and Core-Other Control Blocks

8.4.1 Core-Local and Core-Other Control Blocks Address Map

A set of these registers exists for each proAptiv core in the proAptiv Multiprocessing System. These registers can also be accessed from other cores by first writing the *Core Other Addressing Register* (in the Core-Local Control Block) with the proper core number and then accessing these registers using the Core Other Register block.

All registers are 32 bits wide and should only be accessed using 32-bit uncached load/stores. Reads from unpopulated registers in the GCR address space return 0x0, and writes to those locations are silently dropped without generating any exceptions.

Table 8.41 Core Local and Core Other Block Register Map (Relative to Core-Local/Core-Other CB Offset)

Register Offset	Name	Type	Description
0x0000	Reserved	-	Reserved
0x0008	Core Local Coherence Control Register (*GCR_CL_COHERENCE* *GCR_CO_COHERENCE*)	R/W	Controls which coherent intervention transactions apply to the local core.
0x0010	Core Local Config Register (*GCR_CL_CONFIG* *GCR_CO_CONFIG*)	R	Contains configuration parameters for the Core-Local address space.
0x0018	Core Other Addressing Register (*GCR_CL_OTHER* *GCR_CO_OTHER*)	R/W	Used to access the registers of another core.
0x0020	Core Local Reset Exception Base Register (*GCR_CL_RESET_BASE* *GCR_CO_RESET_BASE*)	R/W	Sets the Reset Exception Base for the local core.
0x0028	Core Local Identification Register (*GCR_CL_ID* *GCR_CO_ID*)	R	Indicates the proAptiv Multiprocessing System Number of the local core.
0x0030	Core Local Reset Exception Extended Base *(GCR_CL_RESET_EXT_BASE GCR_CO_RESET_EXT_BASE)*	R/W	Extends the capabilities of the Core Local Reset Exception Base Register.

Table 8.41 Core Local and Core Other Block Register Map (Relative to Core-Local/Core-Other CB Offset)*(continued)*

Register Offset	Name	Type	Description
0x0040	Core Local TCID_0_PRIORITY Register *(GCR_CL_TCID_0_PRIORITY GCR_CO_TCID_0_PRIORITY)*	R/W	TCID 0 Priority value (2 bits) if IOCU_TYPE=0 in GCR_Cx_CONFIG.
All Others	RESERVED	-	Reserved for future expansion.

8.4.1.1 Core Local Coherence Control Register (GCR_Cx_COHERENCE Offset 0x0008)

This register allows each core to respond to intervention requests from only a subset of the coherent masters within the proAptiv Multiprocessing System (MPS). Software can control entry and exit from the coherence domain by setting the *COH_DOMAIN_EN* bit in this register for:

- Initialization during (asynchronous) boot
- Power control for shutting down and bringing up a core

Table 8.42 Core Local Coherence Control Register

Name	Bits	Description	Read/ Write	Reset State
RESERVED	31:8	Reads as 0. Writes ignored. Must be written with a value of 0x0.	W	0x0
COH_DOMAIN_EN	7:0	Each bit in this field represents a coherent requester within the MPS. Setting a bit within this field will enable interventions to this Core from that requester. The requestor bit which represents the local core is used to enable or disable coherence mode in the local core. Changing the coherence mode for a local core from 0x1 to 0x0 can only be done after flushing and invalidating all the cache lines in the core; otherwise, the system behavior is UNDEFINED. Refer to Section 8.2.11, "Coherency Domains" for more information on the encoding of this field.	R/W	0x0

8.4.1.2 Core Local Config Register

Figure 8.39 Core Local Config Register Format

31 12	11 10	9 0
R	IOCU_TYPE	PVPE

Table 8.43 Core Local Config Register (GCR_Cx_CONFIG Offset 0x0010)

<table>
<tr><th>Name</th><th>Bits</th><th>Description</th><th>Read/ Write</th><th>Reset State</th></tr>
<tr><td>RESERVED</td><td>31:12</td><td>Reads as 0x0. Writes ignored. Must be written with a value of 0x0.</td><td>R</td><td>-</td></tr>
<tr><td><i>IOCU_TYPE</i></td><td>11:10</td><td><table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0x0</td><td>This is a proAptiv core and not an IOCU[1]. Only the proAptiv core can access priority values in the GCR_Cx_TCID_n_PRIORITY registers.</td></tr><tr><td>0x1</td><td>This is a non-caching IOCU (no intervention port). The IOCU does not access the GCR_Cx_TCID_n_PRIORITY registers.</td></tr><tr><td>0x2</td><td>This is a caching IOCU (not currently implemented by MIPS).</td></tr><tr><td>0x3</td><td>Reserved</td></tr></table>[1] Note that the first encoding is redundant information for convenience. It is possible for the system to determine if a core is an IOCU or not by reading the <i>Global Config</i> register.</td><td>R</td><td>IP Configurable Value</td></tr>
<tr><td><i>PVPE</i></td><td>9:0</td><td>Number of VPE's in the system. Note that in the proAptiv core, the term VPE is analogous to a core since there is one VPE per core.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0x0</td><td>1 VPE</td></tr><tr><td>0x1</td><td>2 VPE's</td></tr><tr><td>0x2</td><td>3 VPE's</td></tr><tr><td>0x3</td><td>4 VPE's</td></tr><tr><td>0x5</td><td>6 VPE's</td></tr></table></td><td>R</td><td>IP Configurable Value</td></tr>
</table>

8.4.1.3 Core-Other Addressing Register

This register must be written with the correct core number before accessing the Core-Other address segment.

Figure 8.40 Core Local Config Register Format

31 16	15 0
CORENUM	R

Table 8.44 Core-Other Addressing Register (GCR_Cx_OTHER Offset 0x0018)

Name	Bits	Description	Read/ Write	Reset State
CORENUM	31:16	Core number of the register set to be accessed in the Core-Other address space.	R/W	0x0
RESERVED	15:0	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.-	R	-

8.4.1.4 Core Local Reset Exception Base Register (GCR_Cx_RESET_BASE Offset 0x0020)

This register is used to drive the *SI_ExceptionBase[31:12]* input to the local core. The value is used for placing the exception vectors within the virtual address map during core boot-up time (e.g., when COP0 $Status_{BEV}$ = 1). The value in this register is reset only on Cold Reset (not Warm Reset).

Figure 8.41 Core Local Reset Exception Base Register Format

31 12	11 0
BEVEXCBASE	R

Table 8.45 Core Local Reset Exception Base Register

Name	Bits	Description	Read/ Write	Cold Reset State
BEVEXCBase	31:12	Bits [31:12] of the virtual address that the local core will use as the exception base in the boot environment (C0P0 $Status_{BEV}$=1).	R/W	IP Configuration Value. MIPS Default Value is 0xBFC00
RESERVED	11:0	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	-

For Core 0, the user can configure the reset location at IP configuration.

Core 0 can write the register to force any of the other cores to use a different reset vector. This register write is done before releasing the other core from reset.

This allows a subset of the processor cores to boot one operating system while another subset of the processor cores boot a different operating system.

8.4.1.5 Core Local Identification Register (GCR_Cx_ID Offset 0x0028)

The aliased memory scheme is normally invisible to software when accessing GCR registers within the Core-Local control block. What actually happens is that an offset is used to make a subset of the GCR registers appear in the Core-Local addressing window.

This register reports the core number that is used as the addressing offset for the Core-Local control block.

Figure 8.42 Core Local Identification Register Format

31	0
CORENUM	

Table 8.46 Core Local Identification Register

Name	Bits	Description	Read/ Write	Reset State
CORENUM	31:0	This number is used as an index to the registers within the GCR when accessing the Core-local control block for this core.	R	-

8.4.1.6 Core Local Reset Exception Extended Base Register (GCR_Cx_RESET_EXT_BASE Offset 0x0030)

This register is an extension to the Core-Local Reset Exception Base Register (see Section 8.4.1.4 “Core Local Reset Exception Base Register (GCR_Cx_RESET_BASE Offset 0x0020)”). It also is used to drive the *SI_ExceptionBase* input to the local core. The value is used for placing the exception vectors within the virtual address map during core boot-up time (e.g., when COP0 $Status_{BEV}$=1). The value in this register is reset only on Cold Reset (not Warm Reset).

Figure 8.43 Core Local Exception Extended Base Register Format

31	30	29 28	27 20	19 8	7 1	0
EVAReset	UEB	R	BEVExceptionBaseMask	R	BEVExceptionBasePA	PRESENT

Table 8.47 Core Local Reset Exception Extended Base Register

Name	Bits	Description	Read/ Write	Cold Reset State
EVAReset	31	Assertion of this bit indicates to the core to come up in the EVA configuration at reset. This bit is originally set based on the state of the EVAReset pin during reset.	R/W	IP Configuration Value. MIPS Default Value is 0
UseExceptionBase	30	UseExceptionBase address. This bit reflects the state of the SI_UseExceptionBase pin at reset. In the legacy configuration, if the *SI_UseExceptionBase* pin is not asserted, then the BEV location defaults to 0xBFC0_0000. If the *SI_UseExceptionBase* pin is asserted, address bits *SI_ExceptionBase[31:30]* are forced to a value of 2’b10 to force the BEV location into the KSEG0/ KSEG1 space. Refer to Section 3.7.2 in Chapter 3 for more information. This pin is only used in the legacy configuration. There is one *SI_UseExceptionBase* pin per core.	R/W	IP Configuration Value. MIPS Default Value is 1
RESERVED	29:28	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	-
BEVExceptionBaseMask	27:20	This field is used to determine the size of the boot exception vector overlay region from 1 MB to 256 MB in powers of two. This field reflects the state of the *SI_ExceptionBaseMask[27:20]* pins at reset. This field is used to mask bits [27:20] of the virtual address that the local core will use as the exception base in the boot environment (C0P0 $Status_{BEV}$ = 1). These pins are used in both the legacy and EVA configurations. There is one set of *SI_ExceptionBaseMask* pins per core. Refer to Section 3.7.2 in Chapter 3 for more information.	R/W	IP Configuration Value. MIPS Default Value is 0x00
RESERVED	19:8	Reads as 0x0. Must be written with a value of 0x0.	R	-

Table 8.47 Core Local Reset Exception Extended Base Register *(continued)*

Name	Bits	Description	Read/ Write	Cold Reset State
BEVExceptionBasePA	7:1	BEV exception base physical address. This field contains the upper bits of the physical address that the local core will use as the exception base in the boot environment (C0P0 $Status_{BEV}$ = 1).and reflects the state of the *SI_ExceptionBasePA[31:29]* pins at reset. The size of the overlay region defined by *SI_ExceptionBaseMask[27:20]* is remapped to a location in physical address space pointed to by the *SI_ExceptionBasePA[31:29]* pins. This allows the overlay region to be placed into one of the 512 MB segments in physical memory. These pins are used in both the legacy and EVA configurations. There is one set of *SI_ExceptionBasePA* pins per core. Note that the bits of this register correspond to upper address bits 35:29. However, in the proAptiv core only the lower three bits (31:29) are used, which correspond to bits 3:1 of this field. The upper four bits are reserved for future cores which implement a 36-bit address. This bit should always be driven with a value of 0x0. Refer to Section 3.7.2 in Chapter 3 for more information.	R/W	IP Configuration Value. MIPS Default Value is 0x00.
PRESENT	0	Reads as 0x1. Writes are ignored	R	1

8.4.1.7 Core Local TCID Registers (GCR_Cx_TCID_PRIORITY Offset 0x0040)

In the proAptiv core, there is one thread context per core. Hence only one TCID register if required.

Figure 8.44 Core Local TCID Register Format

31	2	1	0
Reserved		TCID_PRIORITY	

Table 8.48 Core Local TCID Register Description

Name	Bits	Description	Read/ Write	Reset State
Reserved	31:2	Reads as 0x0. Must be written with a value of 0x0.	R	0x0000_000
TCID_PRIORITY	1:0	TCID priority. This 2-bit value contains the thread context priority level and is encoded as follows: 00: Lowest priority .... 11: Highest priority	R	0x0

8.5 Global Debug Control Block

8.5.1 Global Debug Control Block Address Map

This block holds registers which are used for debugging the CM2 and software which uses the coherence features supplied by the CM2. The registers associated with PDTrace are reset upon assertion of the TAP controller reset. The other registers in this block are reset when the CM2 is reset. TAP reset occurs when *PB_EJ_TRST_N* is asserted or the Test-Logic-Reset TAP state is entered.

Table 8.49 Global Debug Block Register Map (Relative to Global Debug Block Offset)

Register Offset	Name	Type	Reset Source	Description
0x0008	PDTrace TCBControlB Register *(GCR_DB_TCBCONTROLB)*	R/W	TAP	Controls how the TCB deals with the trace information. This register only exists if the CM2 is configured with PDTrace.
0x0010	CM2 PDTrace TCBControlD Register (*GCR_DB_TCBCONTROLD*)	R/W	TAP	Controls CM2 PDTrace. This register only exists if the CM2 is configured with PDTrace.
0x0020	PDTrace TCBControlE Register *(GCR_DB_TCBCONTROLE)*	R/W	TAP	Controls how the TCB deals with trace information. This register only exists if the CM2 is configured with PDTrace.
0x0028	PDTrace TCB Config Register *(GCR_DB_TCBConfig)*	R/W	TAP	Contains trace control block configuration information such as probe width, on-trace memory size, and trace clock ratios.
0x0040	PDTrace TCBSYS Register *(GCR_DB_TCBSYS)*	R/W	TAP	Controls how external logic uses the System Trace interface. Bit 31 is a PRESENT bit and bits [30:0] are completely user defined. The output of this register is available on the TC_Sys_UserCtl pins. This register only exists if the CM2 is configured with PDTrace.
0x0100	CM2 Performance Counter Control Register (*GCR_DB_PC_CTL*)	R/W	CM2	Controls starting/stopping of Performance Counters.
0x0108	PDTrace Trace Word Read Pointer Register *(GCR_DB_TCBRDP)*	R/W	TAP	Pointer into the On-Chip Trace Buffer memory for reads from *GCR_DB_TCBTW_LO* and *GCR_DB_TCBTW_HI* registers. This register only exists if the CM2 is configured with PDTrace.
0x0110	PDTrace Trace Word Write Pointer Register (*GCR_DB_TCBWRP)*	R/W	TAP	Pointer into the On-Chip Trace Buffer memory for the next TraceWord write from *GCR_DB_TCBTW_LO* and *GCR_DB_TCBTW_HI* registers. This register only exists if the CM2 is configured with PDTrace.

Table 8.49 Global Debug Block Register Map (Relative to Global Debug Block Offset)*(continued)*

Register Offset	Name	Type	Reset Source	Description
0x0118	PDTrace Trace Word Start Pointer Register *(GCR_DB_TCBSTP)*	R/W	TAP	Pointer into On-Chip Trace Buffer that is used to determine when all entries in the trace buffer have been filled. This register only exists if the CM2 is configured with PDTrace.
0x0120	CM2 Performance Counter Overflow Status Register (*GCR_DB_PC_OV*)	R/W	CM2	Indicates which performance counters have overflowed.
0x0130	CM2 Performance Counter Event Select Register (*GCR_DB_PC_EVENT*)	R/W	CM2	Selects event type of each performance counter.
0x0180	CM2 Performance Cycle Counter Register (*GCR_DB_PC_CYCLE*)	R/W	CM2	Counts cycles.
0x0190	CM2 Performance Counter 0 Qualifier Register (*GCR_DB_PC_QUAL0*)	R/W	CM2	Performance counter 0 event qualifiers.
0x0198	CM2 Performance Counter 0 Register (*GCR_DB_PC_CNT0*)	R/W	CM2	Performance Counter 0 value.
0x01A0	CM2 Performance Counter 1 Qualifier Register (*GCR_DB_PC_QUAL1*)	R/W	CM2	Performance counter 1 event qualifiers.
0x01A8	CM2 Performance Counter 1 Register (*GCR_DB_PC_CNT1*)	R/W	CM2	Performance Counter 1 value.
0x0200	PDTrace Trace Word Lo Register *(GCR_DB_TCBTW_LO)*	R/W	TAP	Access point to read TraceWords from the On-Chip Trace Buffer memory, Least Significant 32-bits.
0x0208	PDTrace Trace Word Hi Register *(GCR_DB_TCBTW_HI)*	R/W	TAP	Access point to read TraceWords from the On-Chip Trace Buffer memory, Most Significant 32-bits.
All Others	RESERVED			

All registers are 32 bits wide and should only be accessed using 32-bit uncached load/stores. Reads from unpopulated registers in the GCR address space return 0x0 and writes to those locations should be silently dropped without generating any exceptions.

8.5.1.1 CM2 PDTrace TCB ControlB Register (GCR_DB_TCBCONTROLB Offset 0x0008)

The TCB includes a control register, *GCR_DB_TCBCONTROLB* (0x11). This register configures interfaces to the trace buffer. This register only exists if the CM2 is configured with PDTrace.

The format of the *GCR_DB_TCBCONTROLB* register is shown below, and the fields are described in Table 8.50.

Figure 8.45 PDTrace TCB ControlB Register Format

31	30 28	27 26	25 20	19	18	17	16	15	14	13 12	11	10 8	7	6 2	1	0
WE	R	TWSrcWidth	R	STCE	TRPAD	R	RM	TR	BF	TM	R	CR	Cal	R	OfC	EN

Table 8.50 PDTrace TCB ControlB Register

Fields		Description	Read / Write	Reset State
Name	Bits			
WE	31	Write Enable. Only when set to 1 will the other bits of this register be written. This bit will always read 0.	R	0
Reserved	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. The value for the CM2 is always 2’b10, indicating a four bit source field width.	R	2’b10
Reserved	25:20	This field is used by EJTAG to access other PDTtrace registers. Although the field is R/W via core accesses, this field has no function for core accesses.	R/W	0
STCE	19	System Trace capture enable. When asserted, the System Trace port of the Funnel is enabled to capture System Trace stream data. When not asserted, System Trace stream data is not captured regardless of *TC_Sys_Valid[1:0]* input pin state.	R/W	0
TRPAD	18	Trace RAM access disable bit. When set, core reads and writes to the on-chip trace RAM using GCR accesses are inhibited. If TRPAD is set, memory-mapped writes to the GCR_DB_TCBTW_LO and GCR_DB_TCBTW_HI registers have no effect, and memory-mapped reads from GCR_DB_TCBTW_LO and GCR_DB_TCBTW_HI do not access the Trace RAM and 0 is returned. Also, when TRPAD is set, then memory-mapped writes to the following registers are inhibited: *TCBTW* *TCBRDP* *TCBWRP* *TCBSTP*	R/W	0
Reserved	17	Reserved. Must be written as zero; returns zero on read.	R	0
RM	16	Read on-chip trace memory. When this bit is set, the read address-pointer of the on-chip memory in register *TCBRDP* is set to the value held in *TCBSTP*. Subsequent access to the *TCBTW* register (through the *TCBDATA* register), will automatically increment the read pointer in register *TCBRDP* after each read. 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*.	R/W	0

Table 8.50 PDTrace TCB ControlB Register *(continued)*

<table>
<thead>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
</thead>
<tbody>
<tr><td><i>TR</i></td><td>15</td><td>Trace memory reset.
When written to one, the address pointers for the on-chip trace memory <i>TCBSTP, TCBRDP and TCBWRP</i> are reset to zero. Also the RM and BF bits are reset to 0.
This bit is automatically reset back to 0, when the reset specified above is completed.</td><td>R/W1</td><td>0</td></tr>
<tr><td><i>BF</i></td><td>14</td><td>Buffer Full indicator that the TCB uses to communicate to external software that the on-chip trace memory is full. This bit is cleared when writing a 1 to the TR bit.
This bit has no function if on-chip memory is not implemented.</td><td>R</td><td>0</td></tr>
<tr><td><i>TM</i></td><td>13:12</td><td>Trace Mode. This field determines how the trace memory is filled when using the simple-break control in the PDtrace™ IF to start or stop trace.
<table>
<thead><tr><th>TM</th><th>Trace Mode</th></tr></thead>
<tbody>
<tr><td>00</td><td>Trace-To</td></tr>
<tr><td>01</td><td>Trace-From</td></tr>
<tr><td>10</td><td>Reserved</td></tr>
<tr><td>11</td><td>Reserved</td></tr>
</tbody>
</table>
In Trace-To mode, the on-chip trace memory is filled, continuously wrapping around, 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 (when the write pointer address is the same as the start pointer address). If a <i>TCBTRIGx</i> trigger control register is used to start/stop tracing, then this field should be set to Trace-To mode.
These bits have no function if on-chip memory is not implemented.</td><td>R/W</td><td>0</td></tr>
<tr><td>0</td><td>11</td><td>Read as Zero. Writes ignored. Must be written with a value of 0x0.</td><td>R</td><td>0</td></tr>
<tr><td><i>CR</i></td><td>10:8</td><td>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 8.51.
<b>Note:</b> 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.
These bits have no function if off-chip memory is not implemented.</td><td>R/W</td><td>3’b100</td></tr>
</tbody>
</table>

Table 8.50 PDTrace TCB ControlB Register *(continued)*

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td><i>Cal</i></td><td>7</td><td>Calibrate off-chip trace interface.
If set, 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.
<table>
<tr><th></th><th colspan="4">Calibrations pattern</th></tr>
<tr><th></th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="13">This pattern is replicated for every 4 bits of <i>TR_DATA</i> pins.</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>1</td></tr>
</table>
<b>Note:</b> The clock source of the TCB and PIB must be running.
These bits have no function if off-chip memory is not implemented.</td><td>R/W</td><td>0</td></tr>
<tr><td>Reserved</td><td>6:2</td><td>Read as Zero. Writes ignored. Must be written with a value of 0x0.</td><td>R</td><td>0</td></tr>
<tr><td><i>OfC</i></td><td>1</td><td>If set to 1, trace is sent to off-chip memory using <i>TR_DATA</i> pins.
If not set, trace info is sent to on-chip memory.
This bit is read only if one of these options exists.</td><td>R/W</td><td>Preset</td></tr>
<tr><td>EN</td><td>0</td><td>Funnel Trace Enable. When this bit is set, the trace funnels accepts trace information from the CM2, cores, and/or system trace and writes the information to off-chip or on-chip memory.
When this bit is cleared, the trace funnel drops all new trace information from the those sources. The trace information already accepted by the trace funnel is sent to the off-chip or on-chip memory, but new trace information is dropped and not written out.</td><td>R/W</td><td>0</td></tr>
</table>

Table 8.51 Clock Ratio Encoding of the CR Field

Encoding of CR Field	Trace Clock:Core Clock Ratio
3’b000	1:20
3’b001	1:16
3’b010	1:12
3’b011	1:10
3’b100	1:2
3’b101	1:4
3’b110	1:6
3’b111	1:8

8.5.1.2 CM2 PDTrace TCB ControlD Register (GCR_DB_TCBCONTROLD Offset 0x0010)

Figure 8.46 PDTrace TCB ControlD Register Format

31 30	29 28	27 26	25 24	23 22	21 20	19 18	17 16	15 12	11 8	7	6	5	4 3	2	1	0
R	P6_Ctl	P5_Ctl	P4_Ctl	P3_Ctl	P2_Ctl	P1_Ctl	P0_Ctl	R	TWSrcVal	WB	STEn	IO	TLev	AE	GCE	CME

Table 8.52 CM2 PDTrace TCB ControlD Register Descriptions

<table>
<tr><th>Name</th><th>Bits</th><th>Description</th><th>Read/ Write</th><th>Reset State</th></tr>
<tr><td>RESERVED</td><td>31:30</td><td>Reserved.</td><td>R/W</td><td>0x0</td></tr>
<tr><td><i>P6_Ctl</i></td><td>29:28</td><td>Provides specific control over tracing transactions on Port 6 of the CM. (the IOCU on 6 core configurations).
<table>
<tr><th>Encoding</th><th>Description</th></tr>
<tr><td>00</td><td>Tracing Enabled, no Address Tracing</td></tr>
<tr><td>01</td><td>Tracing Enabled with Address Tracing</td></tr>
<tr><td>10</td><td>Reserved</td></tr>
<tr><td>11</td><td>Tracing Disabled</td></tr>
</table></td><td>R/W</td><td>0x0</td></tr>
<tr><td><i>P5_Ctl</i></td><td>27:26</td><td>Provides specific control over tracing transactions on Port 5 of the CM2 (core 5). See encoding for <i>P6_Ctl</i>.</td><td>R/W</td><td>0x0</td></tr>
<tr><td><i>P4_Ctl</i></td><td>25:24</td><td>Provides specific control over tracing transactions on Port 4 of the CM2 (core 4 on 6 core configurations or the IOCU on 4 core or less configurations). See encoding for <i>P6_Ctl</i>.</td><td>R/W</td><td>0x0</td></tr>
<tr><td><i>P3_Ctl</i></td><td>23:22</td><td>Provides specific control over tracing transactions on Port 3 of the CM2 (core 3). See encoding for <i>P6_Ctl</i>.</td><td>R/W</td><td>0x0</td></tr>
</table>

Table 8.52 CM2 PDTrace TCB ControlD Register Descriptions *(continued)*

<table>
<tr><th>Name</th><th>Bits</th><th>Description</th><th>Read/
Write</th><th>Reset State</th></tr>
<tr><td><i>P2_Ctl</i></td><td>21:20</td><td>Provides specific control over tracing transactions on Port 2 of the CM2 (core 2). See encoding for <i>P6_Ctl</i>.</td><td>R/W</td><td>0x0</td></tr>
<tr><td><i>P1_Ctl</i></td><td>19:18</td><td>Provides specific control over tracing transactions on Port 1 of the CM2 (core 1). See encoding for <i>P6_Ctl</i>.</td><td>R/W</td><td>0x0</td></tr>
<tr><td><i>P0_Ctl</i></td><td>17:16</td><td>Provides specific control over tracing transactions on Port 0 of the CM2 (core 0). See encoding for <i>P6_Ctl</i>.</td><td>R/W</td><td>0x0</td></tr>
<tr><td>RESERVED</td><td>15:12</td><td>Reads as 0x0. Writes ignored. Must be written with a value of 0x0.</td><td>R</td><td>0x0</td></tr>
<tr><td><i>TwSrcVal</i></td><td>11:8</td><td>The source ID inserted into the Trace Word by the CM.
NOTE: When disabling trace by setting <i>Global_CM_En</i> to 0, the value in TWSrcVal continues to be used until all trace messages have been flushed from the CM. Therefore, when writing to this register to disabled, the correct value must still be written into the <i>TWSrcVal</i> field.</td><td>R/W</td><td>0xF</td></tr>
<tr><td><i>WB</i></td><td>7</td><td>When this bit is set, Coherent Writeback requests are traced. If this bit is not set, all Coherent Writeback requests are suppressed from the CM2 PDTrace Stream.</td><td>R/W</td><td>0x0</td></tr>
<tr><td><i>ST_En</i></td><td>6</td><td>System Trace Enable. Driven to the CM2 output pin <i>TC_Sys_Enable</i>. External logic can use this output to control generation of the System Trace stream.</td><td>R/W</td><td>0x0</td></tr>
<tr><td><i>IO</i></td><td>5</td><td>Inhibit Overflow on the CM2 PDTrace FIFO full condition. When set to 0, the CM2 will drop a new PDTrace message if the internal PDTrace FIFOs are full.
When set to 1, the CM2 will not drop PDTrace messages, but may stall transactions within the CM2 when the internal PDTrace FIFOs are full.</td><td>R/W</td><td>0x0</td></tr>
<tr><td><i>TLev</i></td><td>4:3</td><td>This defines the current trace level being used by CM2 PDtrace:
<table>
<tr><th>Encoding</th><th>Description</th></tr>
<tr><td>00</td><td>No Timing Information</td></tr>
<tr><td>01</td><td>Include Stall Times, Causes</td></tr>
<tr><td>10</td><td>Reserved</td></tr>
<tr><td>11</td><td>Reserved</td></tr>
</table></td><td>R/W</td><td>0x0</td></tr>
<tr><td><i>AE</i></td><td>2</td><td>When set to 1, address tracing is always enabled for the CM. When set to 0, address tracing may be enabled on a per-port basis through the P<x>_Ctl bits.</td><td>R/W</td><td>0x0</td></tr>
<tr><td>Global_CM_En</td><td>1</td><td>Setting this bit to 1 enables tracing from the CM2 as long as the CM_EN bit is also enabled.</td><td>R/W</td><td>0x0</td></tr>
<tr><td><i>CM_EN</i></td><td>0</td><td>This is the master trace enable for the CM. When zero, tracing from the CM2 is always disabled. When set to one, tracing is enabled from whenever the other enabling functions are also true.</td><td>R/W</td><td>0x0</td></tr>
</table>

See Section 8.5.1.2, "CM2 PDTrace TCB ControlD Register (GCR_DB_TCBCONTROLD Offset 0x0010)," for more information about how this register is used.

This register only exists if the CM2 is configured with PDTrace.

8.5.1.3 CM2 PDTrace TCB ControlE Register (GCR_DB_TCBCONTROLE Offset 0x0020)

Figure 8.47 PDTrace TCB ControlE Register Format

31 … 9	8	7	… 1	0
R	Tridle	WB	R	PeC

Table 8.53 TCBCONTROLE Register

Name	Bits	Description	Read / Write	Reset State
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. TrIdle is set when the system traces on all cores, and the CM2, have disabled PDTrace and the trace funnel has written all outstanding trace information to the off-chip or on-chip memory.	R	1
0	7:1	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
PeC	0	Performance Control Tracing is not implemented.	R	0

This register only exists if the CM2 is configured with PDTrace.

See Section 8.5.1.3, "CM2 PDTrace TCB ControlE Register (GCR_DB_TCBCONTROLE Offset 0x0020)," for more information about how this register is used.

8.5.1.4 CM2 PDTrace TCB Config Register (GCR_DB_TCBConfig Offset 0x0028)

This register is also accessible by EJTAG via the TCBDATA instruction as described in the EJTAG Debug Support chapter.

Figure 8.48 PDTrace TCB Config Register Format

31	30 … 21	20 … 17	16 … 14	13 … 11	10 … 9	8 … 6	5	4	3 … 0
CF1	R	SZ	CRMax	CRMin	PW	R	OnT	OfT	REV

Table 8.54 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 reads zero.	R	0

Table 8.54 TCBCONFIG Register Field Descriptions *(continued)*

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td><i>Reserved</i></td><td>30:21</td><td>Read as Zero. Writes ignored. Must be written with a value of 0x0.</td><td>R</td><td>0</td></tr>
<tr><td><i>SZ</i></td><td>20:17</td><td>On-chip trace memory size. This field holds the encoded size of the on-chip trace memory.
The size in bytes is given by $2^{(SZ+8)}$. i.e., the lowest value is 256 bytes, and the highest is 8 MB.
This bit is reserved if on-chip memory is not implemented.</td><td>R</td><td>Preset</td></tr>
<tr><td><i>CRMax</i></td><td>16:14</td><td>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 8.51.
This bit is reserved if off-chip trace option is not implemented.</td><td>R</td><td>Preset</td></tr>
<tr><td><i>CRMin</i></td><td>13:11</td><td>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 8.51.
This bit is reserved if off-chip trace option is not implemented.</td><td>R</td><td>Preset</td></tr>
<tr><td><i>PW</i></td><td>10:9</td><td>Probe Width: Number of bits available on the off-chip trace interface <i>TR_DATA</i> pins. The number of TR_DATA pins is encoded, as shown in the table.
<table>
<tr><th>PW</th><th>Number of bits used on TR_DATA</th></tr>
<tr><td>00</td><td>4 bits</td></tr>
<tr><td>01</td><td>8 bits</td></tr>
<tr><td>10</td><td>16 bits</td></tr>
<tr><td>11</td><td>reserved</td></tr>
</table>
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.</td><td>R</td><td>Preset</td></tr>
<tr><td>Reserved</td><td>8:6</td><td>Read as Zero. Must be written with a value of 0x0.</td><td>R</td><td>0</td></tr>
<tr><td><i>OnT</i></td><td>5</td><td>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.</td><td>R</td><td>Preset</td></tr>
<tr><td><i>OfT</i></td><td>4</td><td>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 (<i>TC_PibPresent</i> asserted).</td><td>R</td><td>Preset</td></tr>
<tr><td><i>REV</i></td><td>3:0</td><td>Revision of TCB.</td><td>R</td><td>0x3</td></tr>
</table>

This register only exists if the CM2 is configured with PDTrace.

8.5.1.5 CM2 Performance Counter Control Register (GCR_DB_PC_CTL Offset 0x0100)

Figure 8.49 CM2 Performance Counter Control Register Format

31	30	29	28	10
R	Perf-Int_En	Perf_OvF_Stop	R	

9	8	7	6	5	4	3	0
P1_Reset	P1_CountOn	P1_Reset	P1_CountOn	Cycl_Cnt_Reset	Cycl_Cnt_CountOn	Perf_Num_Cnt	

Table 8.55 CM2 Performance Counter Control Register

Name	Bits	Description	Read/ Write	Reset State
Reserved	31	Read as Zero. Must be written with a value of 0x0.	R	0x0
Perf_Int_En	30	Enable Interrupt on counter overflow. If set to 1, a CM2 performance counter interrupt is generated when any enabled CM2 performance counter overflows.	R/W	0x0
Perf_Ovf_Stop	29	Stop Counting on overflow. If set to 1, all CM2 Performance counters stop counting when any enabled CM2 performance counter overflows i.e., the counter has reached 0xFFFF_FFFF.	R/W	0x0
Reserved	28:10	Read as Zero. Must be written with a value of 0x0.	R	0x0
P1_Reset	9	If set to 1, CM2 Performance Counter 1 and *P1_Overflow* bit is reset before counting is started. If set to 0 counting is resumed from previous value. This bit is automatically set to 0 when the counter is reset, so *P1_Reset* is always read as 0.	R/W	0x0
P1_CountOn	8	Start Counting. If this bit is set to 1 then CM2 Performance Counter 1 and the *P1_Overflow* bit starts counting the specified event. If this bit is set to 0 then CM2 Performance Counter 1 is disabled. This bit is automatically set to 0 if any counter overflows and *Perf_Ovf_Stop* is set to 1.	R/W	0x0
P0_Reset	7	If set to 1, CM2 Performance Counter 0 and *P0_Overflow* bit is reset before counting is started. If set to 0 counting is resumed from previous value. This bit is automatically set to 0 when the counter is reset, so *P0_Reset* is always read as 0.	R/W	0x0
P0_CountOn	6	Start/Stop Counting. If this bit is set to 1 then CM2 Performance Counter 0 starts counting the specified event. If this bit is set to 0 then CM2 Performance Counter 0 is disabled. This bit is automatically set to 0 if any counter overflows and *Perf_Ovf_Stop* is set to 1.	R/W	0x0
Cycl_Cnt_Reset	5	If set to 1, the *CM2 Cycle Counter Register* and the *Cycl_Cnt_Overflow* bit is reset before counting is started. If set to 0 counting is resumed from previous value. This bit is automatically set to 0 when the counter is reset, so *Cycl_Cnt_Reset* is always read as 0.	R/W	0x0
Cycl_Cnt_CountOn	4	Start/Stop the Cycle Counter. If this bit is set to 1 then CM2 Cycle Counter starts counting. If this bit is set to 0 then CM2 Cycle Counter is disabled. This bit is automatically set to 0 if any Counter Overflows and *Perf_Ovf_Stop* is set to 1.	R/W	0x0

Table 8.55 CM2 Performance Counter Control Register

Name	Bits	Description	Read/ Write	Reset State
Perf_Num_Cnt	3:0	The number of performance counters implemented (not including the cycle counter). The CM2 has 2 performance counters.	R	0x2

8.5.1.6 CM2 PDTrace TCB Trace Word Read Pointer Register (*GCR_DB_TCBRDP* Offset 0x0108)

The *TCBRDP* register is an 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 also accessible by EJTAG via the TCBDATA instruction as described in the EJTAG Debug Support chapter.

The format of the *TCBRDP* register is shown below and the fields are described in Table 8.56. 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 8.50 TCBRDP Register Format

31 n+1	n 0
Data	Address

Table 8.56 TCBRDP Register Field Descriptions

Fields		Description	Read / Write	Reset State
Names	Bits			
Data	31:(n+1)	Reserved. Must be written with zero, reads back zero.	0	0
Address	n:0	Byte address of on-chip trace memory word.	R/W	0

8.5.1.7 CM2 PDTrace TCB Trace Word Write Pointer Register (GCR_DB_TCBWRP Offset 0x0110)

The *TCBWRP* register is an 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 also accessible by EJTAG via the TCBDATA instruction as described in the EJTAG Debug Support chapter.

The format of the *TCBWRP* register is shown below and the fields are described in Table 8.57. 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 8.51 TCBWRP Register Format

31 n+1	n 0
Data	Address

Table 8.57 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

8.5.1.8 CM2 PDTrace TCB Trace Word Start Pointer Register (GCR_DB_TCBSTP Offset 0x0118)

The *TCBSTP* register is the start pointer register. This pointer is used to determine when all entries in the trace buffer have been filled (when *TCBWRP* has the same value as *TCBSTP*). 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 also accessible by EJTAG via the TCBDATA instruction as described in the EJTAG Debug Support chapter.

The format of the *TCBSTP* register is shown below and the fields are described in Table 8.58. 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 8.52 TCBSTP Register Format

31 n+1	n 0
Data	Address

Table 8.58 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

8.5.1.9 CM2 PDTrace TCB System Trace User Control Register (GCR_DB_TCBSYS Offset 0x0040)

The *TCBSYS* register contents are driven to the *TC_Sys_UserCtl[31:0]* output signals. This register is also mapped to offset 0x0040 in the Global Debug Block of the CM GCRs. Thus, any change to this register will be reflected in these output signals. The format of the *TCBSYS* register is shown below, and the fields are described in Table 8.59.

This register is also accessible by EJTAG via the TCBDATA instruction as described in the EJTAG Debug Support chapter.

Figure 8.53 TCBSYS Register Format

31	30 0
STA	UsrCtl

Table 8.59 TCBSYS Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
STA	31	System Trace Available. Set to 1 if the System Trace Interface is present. Otherwise it is set to 0.	R	Preset
UsrCtl	30:0	User-defined Control.	R/W	0

8.5.1.10 CM2 Performance Counter Overflow Status Register (GCR_DB_PC_OV Offset 0x120)

Figure 8.54 Performance Counter Overflow Status Register Format

31 3	2	1	0
R	P1_OF	P0_OF	Cycl_Cnt_OF

Table 8.60 Performance Counter Overflow Status Register

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
Reserved	31:3	Reserved. Must be written zero, reads back zero.	R	0x0
P1_OF	2	If this bit is set to 1, *CM2 Performance Counter 1* has overflowed i.e., the counter has reached 0xFFFF_FFFF.	R Write 1 to clear	0x0
P0_OF	1	If this bit is set to 1, *CM2 Performance Counter 0* has overflowed i.e., the counter has reached 0xFFFF_FFFF.	R Write 1 to clear	0x0
Cycl_Cnt_OF	0	If this bit is set to 1, the *CM2 Cycle Counter Register* has overflowed.	R Write 1 to clear	0x0

8.5.1.11 CM2 Performance Counter Event Select Register (GCR_DB_PC_EVENT Offset 0x130)

Figure 8.55 CM2 Performance Counter Event Select Register Format

31 16	15 8	7 0
R	P0_Event	P0_Event

Table 8.61 CM2 Performance Counter Event Select Register

Name	Bits	Description	Read/ Write	Reset State
Reserved	31:16	Reserved. Must be written zero, reads back zero.	R	0x0
P1_Event	15:8	Event Selection for CM2 Performance Counter 1. Event numbers are defined in Table 15.1.	R/W	0x0
P0_Event	7:0	Event Selection for CM2 Performance Counter 0. Event numbers are defined in Table 15.1.	R/W	0x0

8.5.1.12 CM2 Cycle Counter Register

The CM2 Cycle Count Register is a 32-bit register that keeps count of CM2 clock cycles. It is controlled through the *Cycl_Cnt_CountOn* and *Cycl_Cnt_Reset* bits in the CM2 Performance Counter Control Register. An overflow of the cycle counter is indicated by a 1 in the *Cycl_Cnt_Overflow* bit in the CM2 Performance Counter Overflow Status Register.

Figure 8.56 CM2 Cycle Count Register Format

31 0
Cycle_Cnt

Table 8.62 CM2 Cycle Counter Register (GCR_DB_PC_CYCLE Offset 0x180)

Name	Bits	Description	Read/ Write	Reset State
Cycle_Cnt	31:0	32-bit count of CM2 clock cycles.	R/W	0x0

8.5.1.13 CM2 Performance Counter n Qualifier Field Register (GCR_DB_PC_QUALn Offset 0x190, 0x1a0)

Figure 8.57 Performance Counter n Qualifier Field Register Format

31	0
Pn_Qualifier	

Table 8.63 CM2 Performance Counter n Qualifier Field Register Descriptions

Name	Bits	Description	Read/ Write	Reset State
Pn_Qualifier	31:0	CM2 Performance Counter n Event Qualifier. The qualifier corresponds to the event configured through the *Performance Counter 0 Event Select Register*.	R/W	0x0

8.5.1.14 CM2 Performance Counter n Register (GCR_DB_PC_CNTn Offset 0x198, 0x1A8)

Figure 8.58 Performance Counter n Register Format

31	0
Pn_Count	

Table 8.64 CM2 Performance Counter n Register

Name	Bits	Description	Read/ Write	Reset State
Pn_Count	31:0	32-bit Performance Counter. The event counted is specified in the *CM2 Performance Counter Event Select Register* and by the corresponding *Qualifier Register*.	R/W	0x0

8.5.1.15 CM2 PDTrace TCB Trace Word LO Register (*GCR_DB_TCBTW_LO* Offset 0x0200)

Reads to this register access the contents of the On-Chip Trace Buffer entry (least significant 32-bits) which is referenced by the *GCR_DB_TCBRDP* register. Writes to this register modify the On-Chip Trace Buffer entry (least significant 32-bits) which is referenced by the *GCR_DB_TCBWRP* register.

A side effect of reading the *TCBTW_LO* 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. A side effect of writing the *TCBTW_LO* register is that the *TCBWRP* register increments to the next TW in the on-chip trace memory. If *TCBWRP* is at the max size of the on-chip trace memory, the increment wraps back to address zero. The use of load half-word or load byte instructions can lead to unpredictable results, and is not recommended.

This register is also accessible by EJTAG via the TCBDATA instruction as described in the EJTAG Debug Support chapter.

Figure 8.59 TCBTW_LO Register Format

31 0
Data

Table 8.65 TCBTW_LO Register Field Descriptions

Names	Bits	Description	Read / Write	Reset State
Data	31:0	Trace Word, least significant 32-bits.	R/W	0

8.5.1.16 CM2 PDTrace TCB Trace Word HI Register (*GCR_DB_TCBTW_HI* Offset 0x0208)

Reads to this register access the contents of the On-Chip Trace Buffer entry (most significant 32-bits) which is referenced by the *GCR_DB_TCBRDP* register. Writes to this register modify the On-Chip Trace Buffer entry (most significant 32-bits) which is referenced by the *GCR_DB_TCBWRP* register.

To read or write a 64-bit trace word from the Trace Buffer, the *GCR_DB_TCBTW_HI* register must be accessed first before the *GCR_DB_TCBTW_LO* register. The access of the *GCR_DB_TCBTW_LO* register causes the appropriate pointer register to be incremented. The use of load half-word or load byte instructions can lead to unpredictable results, and is not recommended.

This register is also accessible by EJTAG via the TCBDATA instruction as described in the EJTAG Debug Support chapter.

Figure 8.60 TCBTW_HI Register Format

31 0
Data

Table 8.66 TCBTW_HI Register Field Descriptions

Names	Bits	Description	Read / Write	Reset State
Data	31:0	Trace Word, most significant 32-bits.	R/W	0

Chapter 9

Global Interrupt Controller

This chapter describes the optional Global Interrupt Controller (GIC) included in the proAptiv Multiprocessing System. The GIC can control up to 256 external interrupt sources in multiples of 8. This chapter describes how software controls the configuration and use of the GIC.

The GIC handles the distribution of interrupts between and among the CPUs in the cluster. The GIC has the ability to route interrupts to each core independently.

The chapter contains the following sections:

- Section 9.1 "GIC Terminology"
- Section 9.2 "GIC Features"
- Section 9.3 "GIC Address Map Overview"
- Section 9.4 "GIC Programming"
- Section 9.5 "Shared Register Set"
- Section 9.6 "GIC Core-Local and Core-Other Register Set"
- Section 9.7 "GIC User-Mode Visible Section"

9.1 GIC Terminology

In the context of the GIC, the term 'Processor' will be used synonymously to refer to a single processor or a virtual processor in a Core that implements the MT ASE, such as interAptiv.

When there is one VPE per core, such as in proAptiv, the processor numbering is as follows:

Table 9.1 Processor Numbering with One VPE per Core

Processor Number	Core Number	VPE Number
0	0	0
1	1	0
2	2	0
3	3	0

9.2 GIC Features

To provide support for a multiprocessor environment, the GIC design includes the following features:

- Accepts interrupts from up to 256 external sources.

- Supports active-high, active-low, rising-edge triggered, falling-edge triggered, and dual-edge triggered interrupt signaling.
- Distributes/partitions the interrupt sources among the available cores.
- Steers any interrupt source to any core interrupt input (Interrupt pin, NMI).
- Allows any core to interrupt any other core.
- Backward compatible with pre-defined MIPS Technologies interrupt modes (legacy, vectored, and EIC).
- Scalable for both the number of interrupt sources as well as the number of cores in the system.
- Able to integrate interrupt messages from peripherals such as PCI-Express.
- Hardware assist features are configurable be software at run-time.
- Provides interval and watchdog timers.

9.3 GIC Address Map Overview

An proAptiv Multiprocessing System can contain up to four cores and eight cores. To avoid the large address space needed for core-specific register sets, an aliasing address scheme is used.

The GIC address space is accessed with uncached load/store commands. The physical address and the core number of the requester is supplied for each load/store command. The core number is used as an index to reference the appropriate subset of the instantiated control registers. By using the core number information, the hardware writes/reads the correct subset of the control registers pertaining to that core. Software does not need to explicitly calculate the register index for the core in question; it is done entirely by hardware.

In the proAptiv Multiprocessing System, any core can access the registers of any other core by using the *Core-Other* address spaces. Software must write the *Core-Other Addressing Register* before accessing these address spaces. The value of this register is used by hardware to index the appropriate subset of the control registers.

Two address “windows” are made available to the programmer:

- A window for the “Local” core (as specified by the core number information).
- A second window for an “Other” core that allows a core to access the register set belonging to another core. The “Other” core is specified by first writing the *Core-Other Addressing Register* in the “local” core address space.

An additional section called the *User-Mode Visible section* is used to give quick user-mode read access to specific GIC registers. The use of this section is meant to avoid the overhead of system calls to read GIC resources, such as counter registers.

The address map of the GIC is shown in Table 9.3.

Table 9.2 GIC Address Space

Segment	Base Offset	Addressing Method	Address Space Size	Virtual Address Space Type
Shared Section Offset	0x00000	Offset relative to *GCR_GIC_Base*	32 KB	Kernel

Table 9.2 GIC Address Space *(continued)*

Segment	Base Offset	Addressing Method	Address Space Size	Virtual Address Space Type
Core-Local Section Offset	0x08000	Offset relative to *GCR_GIC_Base* + using core number as Index	16 KB	Kernel
Core-Other Section Offset	0x0C000	Offset relative to *GCR_GIC_Base* + using *Core-Other Addressing Register* as Index	16 KB	Kernel
User-Mode Visible Section Offset	0x10000	Offset relative to *GCR_GIC_Base*	64 KB	User

As shown in the table above, the GIC address space is divided into four types:

- A *Shared* section in which the external interrupt sources are registered, masked, and assigned to a particular core and interrupt pin. This section is used by all cores.
- A *Core-Local* section in which interrupts local to a core are registered, masked, and assigned to a particular interrupt pin. If External Interrupt Controller Mode (EIC) mode is used for a particular core, the EIC encoder is instantiated here.
- A *Core-Other* section in which the local core can access the Core-Local section of another core by which the interrupt can be registered, masked, and asigned to a particular interrupt pin of the other core. One core can setup the GIC for all cores in the system using this section.
- A *User Mode Visible* section that contains the GIC Hi/Lo counters accessible in user mode for quick user mode access. The use of this section is meant to avoid the overhead of system calls to read GIC resources, such as counter registers.

In the GIC, the *Shared*, *Core-Local*, and *Core-Other* sections are meant to be located in privileged system virtual address space, in which only kernel mode software can initialize and update the interrupt controller.

A separate 64KB address space is allocated so that it may be mapped to *User Mode* virtual address space. Within this address space are aliases for GIC registers that are read so often that it makes sense to make them available to user-mode programs without requiring a system call. The aliases for these registers are read-only. Currently, the only registers that are aliased into this space are the shared *GIC_SH_CounterLo* and *GIC_SH_CounterHi* registers. Refer to Section 9.7 "GIC User-Mode Visible Section" for more information.

9.3.1 GIC Base Address

The GIC base address is a 17-bit value that is programmed into the GCR_CPE_BASE field of the *GCR CPC Base* register located at offset address 0x0088 in the Global Control Block of the CM2 registers. Refer to the *GCR_CPC_BASE Register in* Chapter 8, *CM2 Global Control Registers* for more information on this register.

9.3.2 Block Offsets Relative to the Base Address

The block offsets for each of the three blocks listed in Table 9.3 above are relative to a GIC base address adescribed above and can be located anywhere in physical memory. To determine the physical address of each block listed in Table 9.4, the base address written to the *GCR_GIC_BASE Register* this value would be added to the GIC block offset ranges to derive the absolute physical address as shown in Table 9.4. Note that an example base address of 0x1BDC_0 is used for these calculations.

Table 9.3 Example Physical Address Calculation of the CPC Register Blocks

Example Base Address		GCR Block Offset		Absolute Physical Address	Size (bytes)	Description
0x1BDC_0	+	0x0000 - 0x7FFF	=	0x1BDE_ 0000 - 0x1BDC_7FFF	32 KB	GIC Shared Control Block.
0x1BDC_0	+	0x8000 - 0xBFFF	=	0x1BDE_ 8000 - 0x1BDC_BFFF	16 KB	GIC Core-Local Control Block.
0x1BDC_0	+	0xC000 - 0xFFFF	=	0x1BDE_ C000 - 0x1BDC_FFFF	16 KB	GIC Core-Other Control Block.
0x1BDD_0	+	0x0000 - 0xFFFF	=	0x1BDD_ 0000 - 0x1BDD_FFFF	64 KB	User-Mode Visible Block.

9.3.3 Register Offsets Relative to the Block Offsets

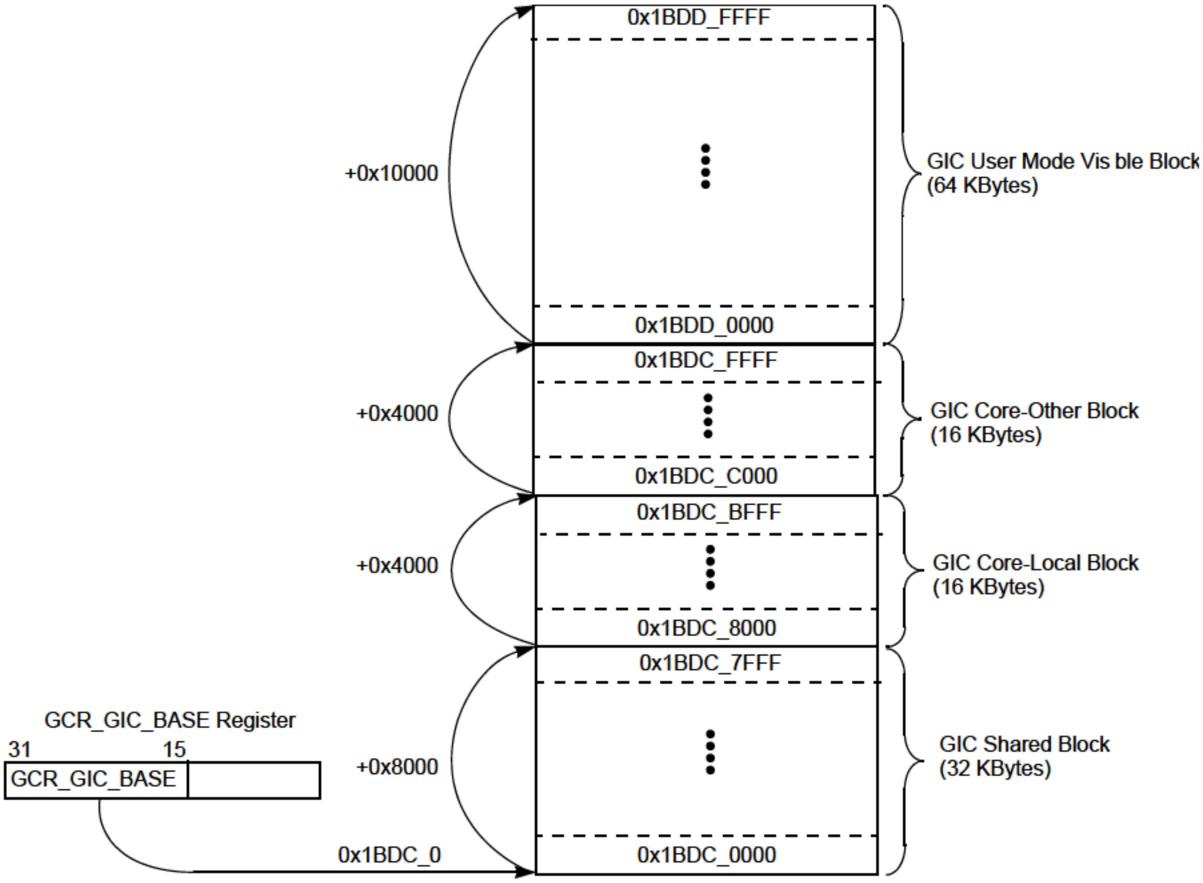
In addition to the block offsets, the register offsets provided in each register description of this chapter are relative to the block offsets shown in Table 9.3 above. To determine the physical address of each register, the base address programmed into the *GCR_GIC_BASE* register is added to the corresponding GIC block offset described above, plus the actual register offset to derive the absolute physical address as shown in Table 9.5.This table shows the physical address for the first few registers of the GIC Shared block. In this table an example base address of 0x1BDC_0 is used.

Table 9.4 Absolute Address of Individual GIC Shared Block Registers

MIPS Default Base		Global Register Block Offset		Global Register Offset		Absolute Physical Address	Global Control Register
0x1BDC_0	+	0x0000	+	0x0000	=	0x1BDC_0000	GIC Config.
0x1BDC_0	+	0x0000	+	0x0010	=	0x1BDC_0010	GIC CounterLo.
0x1BDC_0	+	0x0000	+	0x0014	=	0x1BDC_0014	GIC CounterHi.
0x1BDC_0	+	0x0000	+	0x0020	=	0x1BDC_0020	GIC Revision.
0x1BDC_0	+	0x0000	+	0x0100	=	0x1BDC_0100	CPC Interrupt Polarity 0.
...	+	...	+	...	=	...	...

This concept is described in Figure 9.1 below. In this figure an example base address of 0x1BDE_0 is used.

Figure 9.1 GIC Register Addressing Scheme Using an Example Base Address of 0x1BDC_0



9.4 GIC Programming

This section covers the programming for the following tasks.

- Setting the GIC Base Address and Enabling the GIC
- Configuration of interrupt sources:
- External interrupt source configuration:
  - Level Sensitivity, active high or active low
  - Edge Sensitivity, dual or single edge (falling or Rising)
- Routing of Interrupt external interrupts to specific processors.
- Enabling or Disabling interrupts
- Inter-Processor Interrupts
- Local device interrupt configuration

9.4.1 Setting the GIC Base Address and Enabling the GIC

As described in Section 9.3.1 "GIC Base Address", the base address for the memory mapped registers of the GIC is set using the GIC_BASE_ADDR field of the GCR_GIC_BASE Register. This field is normally programmed by the boot code executing outside of the boot process.

To enable the GIC the GIC_EN bit must be set in this same register.

9.4.2 Configuring Interrupt Sources

The triggering of interrupts is configured through several registers in the GIC that are shared by all processors. All processors can access these registers but in practice these registers are usually programmed at boot time by processor 0. There are three register groups that control the interrupt triggering configuration.

- Trigger type register group
- Edge type register group
- Polarity register group

Each interrupt source is represented by one bit in each register group. Each register in a group is 32 bits so each register controls 32 interrupt sources. The first register in each group would control interrupts 0 - 31, the next 32 - 63 and so on. Since there can be 256 interrupt sources there could be 8 registers in each group. There are enough of these registers in each group to control the number of interrupt sources implemented. The number of interrupt sources is a fixed value configured at core build time. This number can be determined by reading the NUMINTERRUPTS field of the "GIC Configuration Register", GIC_SH_CONFIG. Refer to Section 9.5.3.1 "Global Config Register" for more information.

Each of the interrupt sources can be of either positive (asserted high) or negative (asserted low) polarity. Similarly, any of these sources can be either level-sensitive, single-edge-sensitive, or dual-edge-sensitive. Through the polarity control registers (*GIC_SH_POLx_y*), the trigger type control registers (*GIC_SH_TRIGx_y*) and dual edge control registers (*GIC_SH_DUALx_y*), all of the sources are normalized to positive, level-sensitive signals. This is the interrupt type supported by the CPU interrupt inputs.

For single-edged signaling, the *Polarity* register denotes which edge is used for setting the interrupt register and which edge is ignored. For double-edged signaling, both the rising and falling edges are used to set the interrupt register. These three registers work in conjunction with one another to define the characteristics of each specific interrupt in the system. Each bit of each register corresponds to an interrupt. So for a given bit, the corresponding interrupt characteristics would be defined as shown in Table 9.6. The ‘n’ in the table entries denotes that it can be any bit of a given register, but must be the same bit of each register.

Table 9.5 Selecting Interrupt Polarity, Edge Sensitivity, and Triggering

Polarity (GIC_SH_POL[n])	Trigger (GIC_SH_TRIG[n])	Single/Dual Edge (GIC_SH_DUAL[n])	Description
0	0	x	Interrupt is level sensitive and active low. In this case the contents of the GIC_SH_DUAL have no meaning because level triggering is enabled.
1	0	x	Interrupt is level sensitive and active high. In this case the contents of the GIC_SH_DUAL have no meaning because level triggering is enabled.
0	1	0	Interrupt is single edge triggered on the falling edge of the signal.
1	1	0	Interrupt is single edge triggered on the rising edge of the signal.
x	1	1	Interrupt is dual edge triggered. In this case the contents of the GIC_SH_POL have no meaning because interrupts occur on both the rising and falling edges of the signal.

9.4.2.1 Trigger Type Register Group

The trigger type register group is made up of shared "Global Interrupt Trigger Type Registers", GIC_SH_TRIG. The trigger type can be set to level or edge sensitive. Setting the source bit configures the source to be edge sensitive and clearing it configures it to be level sensitive. For example to set the interrupt source 32 to edge sensitive bit 0 of the second GIC_SH_TRIG Register should be set. Refer to Section 9.5.3.6 “Global Interrupt Trigger Type Registers”, for more information on how to assign this parameter.

9.4.2.2 Edge Type Register Group

The edge type register group is made up of shared "Global Dual Edge Registers", GIC_SH_DUAL. This register group is used if the Trigger type described in the last section is set to edge sensitive and has no effect if the trigger type is level sensitive. The edge type can be either single or dual edge. Setting the source bit configures the source to be dual edge and clearing it configures it to be single edge. For example, to set interrupt source 32 to dual edge sensitive bit 0 of the second Global Dual Edge Registers should be set.

Refer to Section 9.5.3.7 “Global Interrupt Dual Edge Registers” for more information on how to assign this parameters.

9.4.2.3 Polarity Type Register Group

The polarity register group is made up of shared "Global Interrupt Polarity Registers", GIC_SH_POL. This register group is used to determine the polarity sensitivity of the source.

If the interrupt source type is level sensitive then setting the source bit configures the source to be active High, and clearing it configures it to be active low.

If the interrupt is single edge sensitive then setting the source bit configures the source to rising edge toggle and setting clearing it configure it to be falling edge toggle.

This register group has no effect if the edge type was set to dual edge sensitive.

Refer to Section 9.5.3.5 "Global Interrupt Polarity Registers"for more information on how to assign this parameter.

9.4.3 Interrupt Routing

The routing of interrupts to a specific input on a specific processor is controlled by the setting of 2 registers.

- Global Interrupt Map to Processor register, GIC_SH_MAP_VPE — maps the interrupt to a processor.
- Global Interrupt Map to Pin Register, GIC_SH_MAP_PIN — maps interrupt to a specific signal on a processor.

There is one of each of these 32 bit registers for each external interrupt source. The mapping of external interrupt pins and the registers that control them is listed in Table 9.7.

Table 9.6 Mapping of External Interrupts

External Interrupt	Offset	Register Name	External Interrupt	Offset	Register Name
0	0x2000	GIC_SH_MAP0_CORE	248	0x3F00	GIC_SH_MAP248_CORE
	0x0500	GIC_SH_MAP0_PIN		0x08E0	GIC_SH_MAP248_PIN
1	0x2020	GIC_SH_MAP1_CORE	249	0x3F20	GIC_SH_MAP249_CORE
	0x0504	GIC_SH_MAP1_PIN		0x08E4	GIC_SH_MAP249_PIN
2	0x2040	GIC_SH_MAP2_CORE	250	0x3F40	GIC_SH_MAP250_CORE
	0x0508	GIC_SH_MAP2_PIN		0x08E8	GIC_SH_MAP250_PIN
3	0x2060	GIC_SH_MAP3_CORE	251	0x3F60	GIC_SH_MAP251_CORE
	0x050C	GIC_SH_MAP3_PIN		0x08EC	GIC_SH_MAP251_PIN
4	0x2080	GIC_SH_MAP4_CORE	252	0x3F80	GIC_SH_MAP252_CORE
	0x0510	GIC_SH_MAP4_PIN		0x08F0	GIC_SH_MAP252_PIN
5	0x20A0	GIC_SH_MAP5_CORE	253	0x3FA0	GIC_SH_MAP253_CORE
	0x0514	GIC_SH_MAP5_PIN		0x08F4	GIC_SH_MAP253_PIN
6	0x20C0	GIC_SH_MAP6_CORE	254	0x3FC0	GIC_SH_MAP254_CORE
	0x0518	GIC_SH_MAP6_PIN		0x08F8	GIC_SH_MAP254_PIN
7	0x20E0	GIC_SH_MAP7_CORE	255	0x3FE0	GIC_SH_MAP255_CORE
	0x051C	GIC_SH_MAP7_PIN		0x08FC	GIC_SH_MAP255_PIN
8 - 247	0x2100 - 0x3EE0	GIC_SH_MAP8_CORE - GIC_SH_MAP247_CORE			
	0x0520 - 0x08DC	GIC_SH_MAP8_PIN - GIC_SH_MAP247_PIN			

9.4.3.1 Mapping an Interrupt Source to a Processor

There is one shared "Global Interrupt Map to VPE Register", GIC_SH_MAP_VPE for each interrupt source that maps that source to a processor. Bit 0 would map the interrupt source to processor 0; bit 1 would map the interrupt to processor 1 and so on. Refer to Section 9.5.3.14 "Global Interrupt Map to Core Registers" for more information.

9.4.3.2 Mapping and Interrupt Source to a Specific Processor Pin

There is one shared "Global Interrupt Map to Pin Register", GIC_SH_MAP_PIN for each external interrupt source that further maps that source to a specific signal on the processor. There are three bits that control the type of signals that can be assigned to the interrupt source. Refer to Section 9.5.3.13 "Global Interrupt Map to Pin Registers" for more information.

- If set, the MAP_TO_PIN bit will map the external interrupt source to Interrupt Pending bits in the CP0 Cause register of the local processor. The actual Interrupt Pending value is set in the MAP field of this register.
    - Note that in EIC mode, the MAP Field of this register contains the encoded value of the number (0 -63). For example, a value of 0x20 asserts Interrupt 32 (decimal). For vectored interrupt mode, only values of 0x0 through 0x5 should be used. *****Need cross reference to EIC mode*****
- If set, the MAP_TO_NMI bit will map the external interrupt source to the NMI bit in the CP0 Status register. This in essence will cause the processor to soft boot using the boot exception vector as the start of the interrupt routine.

9.4.3.3 Mapping an Interrupt Source to a Register Set

Each processor has one register per interrupt source used when the processor is in "EIC mode" to map the interrupt source to a register set. This is the "EIC Shadow Set Register", GIC_VPEi_EICSS, located in the GIC local and other sections. NOTE: Even through shadow register sets are not present in the proAptiv core, this register must be initialized. Refer to Section 9.6.3.6 "Local EIC Shadow Set Registers" for more information.

The first register corresponds to interrupt source 0; the second to interrupt source 1 and so on. The EIC_SS field is set to the register set number.

9.4.4 Enabling, Disabling, and Polling Interrupts

The Enabling, Disabling and Polling of interrupts is configured through several registers in the GIC that are shared by all processors.

There are 4 shared registers groups for Enabling, Disabling and Polling of interrupts.

- Enabling an interrupt using the "GIC Set Mask Registers", GIC_SH_SMASK
- Disabling an interrupt using the "GIC Reset Mask Registers", GIC_SH_RMASK
- Determining the Enable/Disable state of an interrupt state using "GIC Mask Register", GIC_SH_MASK
- Polling the interrupt active state using the "GIC Pending Register", GIC_PEND_MASK

Like the trigger registers, each interrupt source is represented by one bit in each register group. Each register in a group is 32 bits so each controls 32 interrupt sources. The first register in each group would control interrupts sources 0 - 31, the next 32 - 63 and so on. Since there can be 256 interrupt sources there could be 8 registers in each group. There are enough of these registers in each group to control the number of interrupt sources implemented. The number of interrupt sources is a fixed value configured at core build time. This number can be determined by reading the

NUMINTERRUPTS field of the "GIC Configuration Register", GIC_SH_CONFIG. Refer to Section 9.5.3.1 "Global Config Register" for more information.

9.4.4.1 Enabling External Interrupts

The GIC Set Mask register group is used to enable external interrupts. It is made up of "GIC Set Mask Registers", GIC_SH_SMASK For synchronization purposes this is a write only register. Setting the source bit enables the interrupt. Refer to Section 9.5.3.10 "Global Interrupt Set Mask Registers" for more information.

9.4.4.2 Disabling External Interrupts

The GIC Reset Mask register group is used to disable external interrupts. It is made up of "GIC reset Mask Registers", GIC_SH_RMASK. For synchronization purposes; this is a write only register. Setting the source bit disables the interrupt. Refer to Section 9.5.3.9 "Global Interrupt Reset Mask Registers" for more information.

9.4.4.3 Determining the Enabled or Disabled Interrupt State

The GIC Mask register group is used to determine if an external interrupt is enabled. It is made up of GIC Mask Registers, GIC_SH_MASK. For synchronization purposes; this is a read only register. If a bit is set the corresponding interrupt source is enable. If it is clear the corresponding interrupt is disabled. Refer to Section 9.5.3.11 "Global Interrupt Mask Registers" for more information.

9.4.4.4 Polling for an Active Interrupt

The GIC Pending register group is used to determine if a external interrupt is active. It is made up of GIC Pending Registers, GIC_PEND_MASK. This is a read only register. If a bit is set the corresponding interrupt source is active. If it is clear the corresponding interrupt is inactive. Refer to Section 9.5.3.12 "Global Interrupt Pending Registers" for more information.

9.4.4.5 Programming Example

Incoming interrupts are registered in the *Global Interrupt Pending* registers (*GIC_SH_PENDx_y*). This is the register that software needs to probe to discern the source of the interrupt. The *Global Interrupt Mask* registers (*GIC_SH_MASKx_y*) allow software to temporarily disable any particular interrupt source.

There are separate set (*GIC_SH_SMASKx_y*) and reset (*GIC_SH_RMASKx_y*) mask registers to set/clear individual interrupts to avoid any read-modify-write hazards within the system (multiple cores reading/writing the mask register simultaneously). This mechanism is shown in Figure 9.2 for interrupts 31:0. For interrupts 64:32, a different set of registers is used. Similar for interrupts 95:64, and so on through interrupts 255:224.

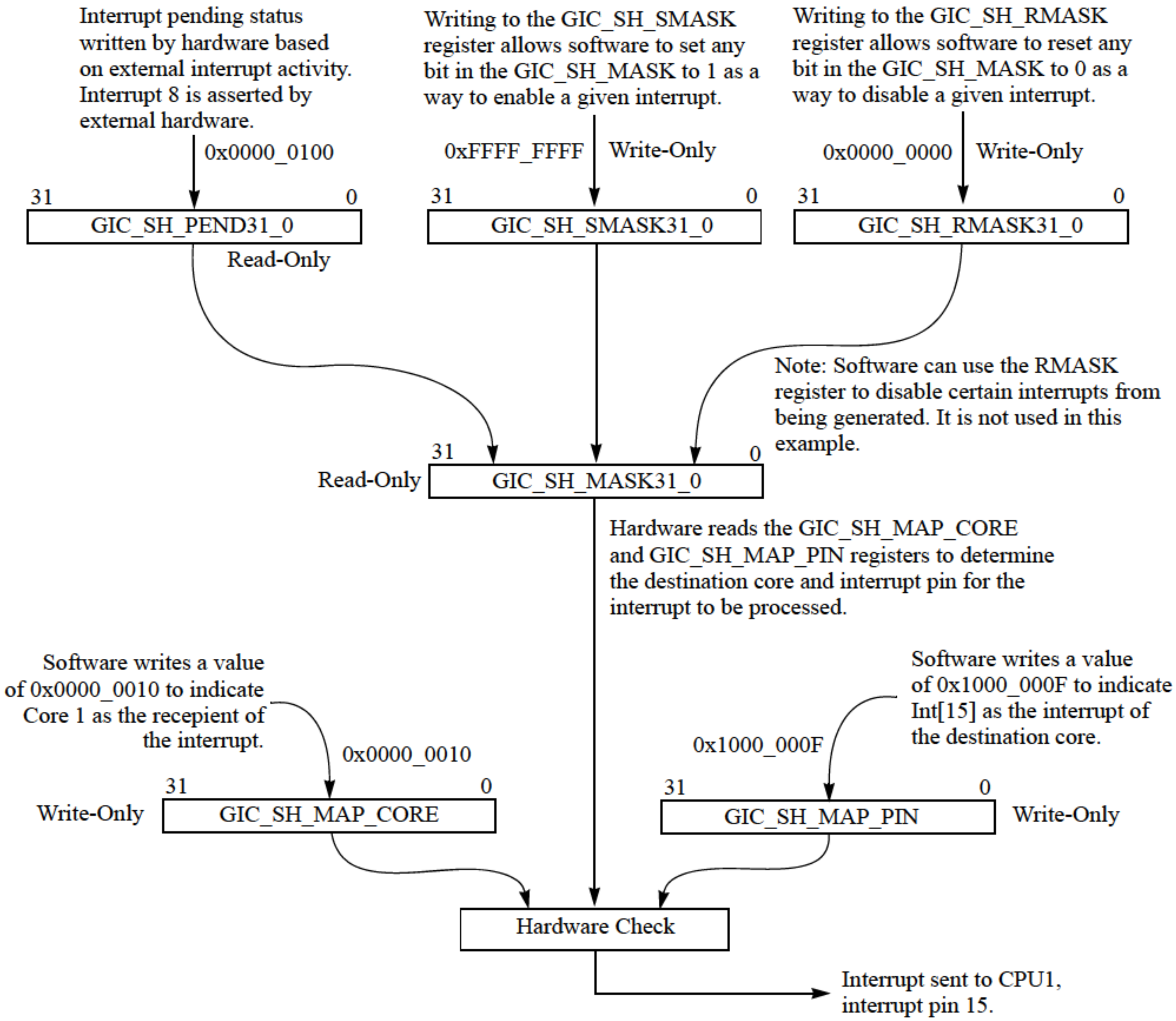
When an interrupt occurs, the corresponding bit in the *GIC_SH_PEND* register is set by hardware. If the corresponding interrupt enable bit in the *GIC_SH_MASK* bit is set, the GIC delivers the interrupt to the appropriate core. The hardware does this by using the *GIC_SH_MAP_CORE* register to send the interrupt to the appropriate core and the *GIC_SH_MAP_PIN* register to set the interrupt pins for that *core*.

In the following example:

- External interrupt 8 is asserted
- All bits of the *GIC_SH_SMASK* register are set, enabling all 32 interrupts.
- The receiving core is #1, and the receiving interrupt is #15.

This example is shown in Figure 9.2 below.

Figure 9.2 Masking and Mapping of Interrupts in the GIC



Interrupt pending status written by hardware based on external interrupt activity. Interrupt 8 is asserted by external hardware.
0x0000_0100
31
0
GIC_SH_PEND31_0
Read-Only
Writing to the GIC_SH_SMASK register allows software to set any bit in the GIC_SH_MASK to 1 as a way to enable a given interrupt.
0xFFFF_FFFF
Write-Only
31
0
GIC_SH_SMASK31_0
Writing to the GIC_SH_RMASK register allows software to reset any bit in the GIC_SH_MASK to 0 as a way to disable a given interrupt.
0x0000_0000
Write-Only
31
0
GIC_SH_RMASK31_0
Note: Software can use the RMASK register to disable certain interrupts from being generated. It is not used in this example.
31
0
Read-Only
GIC_SH_MASK31_0
Hardware reads the GIC_SH_MAP_CORE and GIC_SH_MAP_PIN registers to determine the destination core and interrupt pin for the interrupt to be processed.
Software writes a value of 0x0000_0010 to indicate Core 1 as the recepient of the interrupt.
0x0000_0010
31
0
Write-Only
GIC_SH_MAP_CORE
Software writes a value of 0x1000_000F to indicate Int[15] as the interrupt of the destination core.
0x1000_000F
31
0
GIC_SH_MAP_PIN
Write-Only
Hardware Check
Interrupt sent to CPU1, interrupt pin 15.

9.4.5 Inter-processor Interrupts

Each processor in the system can interrupt any other processor. Each inter-processor interrupt is configured just like an external interrupt using sources not being used by external devices. The interrupt source must be configured to be edge sensitive.

The "Global Interrupt Write Edge Register", GIC_SH_WEDGE is a shared register used to deliver an interrupt to another processor (only one per system). It is also used to clear an interrupt. There are two fields in the GIC_SH_WEDGE register used to do this.

- The RW bit determines if the interrupt is being set (delivered) or cleared. Setting this bit delivers an interrupt and clearing the bit clears the interrupt.
- The Interrupt field should be set to the interrupt number to be set or cleared.

9.4.5.1 WEDGE Register Programming Example

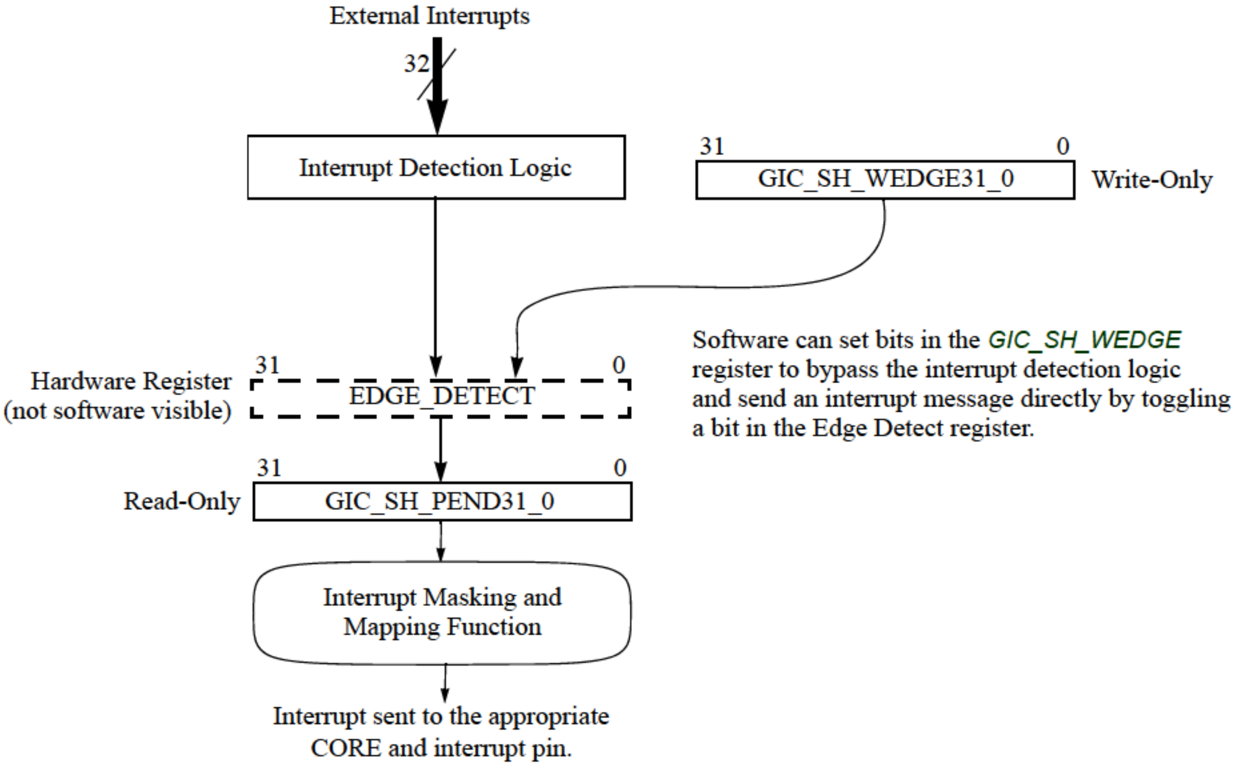
Setting a bit in the *Write Edge* register is treated equivalently to having the edge detection logic see an active edge. Because the programming of the Write Edge register has a direct effect on the state of the internal Edge Detect register, the *Write Edge* register can be used to bypass the edge detection logic. Thus, it does not matter whether the corresponding interrupt is configured to be rising, falling, or dual edge sensitive.

When core 0 wants to interrupt core 1, the number of the interrupt to be used is programmed into the GIC_SH_WEDGE31_0 register. The selected interrupt must be mapped to the target core (core1 in this example) using the GIC_SH_MAPi_CORE register).

For example, assume core 0 wants to toggle interrupt 40. In this case, software writes a value of 0x28 into the GIC_SH_WEDGE31_0 register. Hardware then writes the value in the WEDGE register into the Edge Detect hardware register, effectively bypassing the edge detection logic. Hardware determines that interrupt being toggled belongs to core 1, not core 0. The GIC routing logic then routes interrupt 40 onto the appropriate core 1 interrupt pins.

Figure 9.3 shows how the *Write Edge* register can be used to bypass the interrupt detection logic and assert interrupt directly. Setting a bit in the *Write Edge* register in turn sets the corresponding bit in the internal Edge Detect register, forcing an interrupt to be generated and allowing for inter-processor interrupts within the GIC.

Figure 9.3 Sending Inter-Processor Interrupts in the GIC


External Interrupts
32
Interrupt Detection Logic
31
0
GIC_SH_WEDGE31_0
Write-Only
Software can set bits in the *GIC_SH_WEDGE* register to bypass the interrupt detection logic and send an interrupt message directly by toggling a bit in the Edge Detect register.
Hardware Register (not software visible)
31
0
EDGE_DETECT
31
0
Read-Only
GIC_SH_PEND31_0
Interrupt Masking and Mapping Function
Interrupt sent to the appropriate CORE and interrupt pin.

9.4.5.2 Inter-Processor Interrupt Code Example

Here is an example on how to set up interrupt sources 32 through 39 for inter-processor interrupts. First here is a table of what the #defines are set to.

Table 9.7 Setting Interrupt Sources 32 Through 39

#define	Value	Description
GIC_BASE_ADDR	0xbbdc0000	Virtual Base memory address of the GIC memory mapped registers
GIC_P_BASE_ADDR	0x1bdc0000	Physical Base address of the GIC memory mapped registers
GIC_SH_RMASK63_32	0x0304	Offset into the GIC registers for the GIC Reset Mask Register
GIC_SH_POL63_32	0x0104	Offset into the GIC registers for the GIC Reset Polarity Register
GIC_SH_TRIG63_32	0x0184	Offset into the GIC registers for the GIC Trigger Register
GIC_SH_SMASK63_32	0x0384	Offset into the GIC registers for the GIC Set Mask Register
GCR_CONFIG_ADDR	0xbfbf8000	Base address of the Global Configuration Register
GCR_GIC_BASE	0x0080	Offset int the GCR of the GIC base Address
GIC_SH_MAP0_VPE31_0	0x2000	Offset into the GIC for first map register
GIC_SH_MAP_SPACER	0x20	Spacing between map registers

```
// First load  GIC base address into the GCR and enable the GIC
li    a1, GCR_CONFIG_ADDR + GCR_GIC_BASE // load the address of the GIC Base Address register
li    a0, (GIC_P_BASE_ADDR | 1)          // Physical address + enable
sw    a0, 0(a1)                          // Store the Physical address of the GIC and the enable
                                         // bit to the GCR
// Configure the source pins for inter-processor interrupts
li     a1, GIC_BASE_ADDR                 // load GIC base address
li      a0, 0xff                         // load bits for  interrupts 32..39 lower 8 bits of 2nd  group)
sw     a0, GIC_SH_RMASK63_32(a1)         // (disable  interrupts 32..39)
sw     a0, GIC_SH_TRIG63_32(a1)          // (set source to be edge sensitive for interrupts 32..39)
sw     a0, GIC_SH_POL63_32(a1)     // (set Polarity to rising edge for interrupts32..39)
sw     a0, GIC_SH_SMASK63_32(a1)// (enable   interrupts 32..39)

// Map interrupts to a processor


// The register offset into the GIC for the MAP TO VPE register is obtained by multiplying the
// interrupt number by the spacing size (GIC_SH_MAP_SPACER) and adding the offset for the Global
//  Interrupt Map to VPE Registers (GIC_SH_MAP0_VPE31_0).


li a0, 1          // set bit 0 processor 0
// Map Source 32 processor 0
```

```
sw a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 32)(a1)
sll a0, a0, 1 // set bit 1 for processor 1
// Source 33 to processor 1
sw a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 33)(a1)
sll a0, a0, 1 // set bit 2 for processor 2
// Source 34 to processor 2
sw a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 34)(a1)
sll a0, a0, 1 // set bit 3 for processor 3 or for MT vpe3
 // Source 35 to processor 3
sw a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 35)(a1)
sll a0, a0, 1 // set bit 4 for processor 4
// Source 36 to processor 4
sw a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 36)(a1)
sll a0, a0, 1 // set bit 5 for processor 5
// Source 37 to processor 5
sw a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 37)(a1)
sll a0, a0, 1 // set bit 6 for processor 6
// Source 38 to processor 6
sw a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 38)(a1)
sll a0, a0, 1 // set bit 7 for processor 7
// Source 39 to processor 7
sw a0,GIC_SH_MAP0_VPE31_0+(GIC_SH_MAP_SPACER * 39)(a1)
```

At this point the Map-to-Pin Registers could be used to map each interrupt source to Interrupt Pending bits in the CP0 Cause register of a processor. The default values for the "Map to Pin" registers are the MAP_TO_PIN bit is set and the MAP field is cleared. This example does not change the default values therefore the interrupts are mapped to IP2, Hardware Interrupt 0.

9.4.5.3 Example of Sending an Inter-Processor Interrupt

The following is a C coding example of sending an inter-processor interrupt. First the #defines

Table 9.8

#define	Value	Description
GIC_SH_WEDGE	*((volatile unsigned int*) (0xbbdc0280))	Address of the GIC_WEDGE_REGISTER.
FIRST_IPI	32	Source number for the first IPI.

```
void set_ipi(int cpu_num) {
// Add the enable bit, the first IPI number and the cpu number
// and write it to the GIC_SH_WEDGE register
    GIC_SH_WEDGE = 0x80000000 + FIRST_IPI + cpu_num ;
```

9.4.5.4 Example of Clearing an Inter-Processor Interrupt

Once received, the interrupt routine should do whatever action is intended for the interrupt and clear the interrupt by writing the interrupt number to the GIC_SH_WEDGE register before executing the eret instruction. NOTE: only the interrupt number is set before the write so the R/W bit will be cleared indicating that the interrupt is to be cleared.

```
li      k0, (GIC_SH_WEDGE | GIC_BASE_ADDR)
mfc0    k1, C0_EBASE            // Get cp0 EBase
ext     k1, k1, 0, 10      // Extract CPUNum
addiu   k1, 0x20        // Offset to base of IPI interrupts.
sw      k1, 0(k0)       // Clear this IPI.
```

9.4.6 Local Device Interrupt Configuration

The GIC also controls how devices within the processor and the GIC are configured and mapped locally to the processor.

There are 2 devices that are added as part of the GIC described in this section:

- GIC Interval Timer - a 64 bit timer that compares a local compare registers, GIC_VPE_CompareLo/Hi of a processor with a global counter, GIC_SH_CounterLo/Hi in the GIC and activates an interrupt when they match.
- GIC Watchdog Timer - a 32 bit decrementing counter, GIC_CORE_WD_COUNT that can be used as liveliness signal for a processor.

9.4.6.1 GIC Interval Timer

The interval timer is similar to the CP0 Count/Compare timer within each processor. The difference is the GIC CounterLo/Hi register is global to the CPS so all processors will have the same time reference.

Both the interval count and interval compare values are 8 bytes wide and are made up of 2 (Lo/HI) registers . For each Lo register overflow the Hi register is incremented. If the Hi register overflows, both registers rollover to 0.

Counter Registers

The counter registers, GIC_SH_CounterLo/Hi are in the shared section of the GIC memory map. The counter must be stopped before it is set. This is done by setting the COUNTSTOP bit of the GIC_SH_CONFIG register (link to register reference of GIC_SH_CONFIG). In practical use the counter is usually set by an OS at boot time by one processor. These counter registers are also available (read only) in user mode located at offset 0 of the User Mode Visible Section of the GIC.

The COUNTBITS field of the *GIC_SH_CONFIG* register in Section 9.5.3.1, "Global Config Register" is used to set up the width of the *GIC_SH_CounterHi* register. In the GIC design, this field is fixed at a value of 0x8, indicating a total counter size of 64-bits.

The shared counter registers are defined as follows:

- *GIC_SH_CounterLo* register in Section 9.5.3.2, "GIC CounterLo". Used in conjunction with the *GIC_SH_CounterHi* register. Sets the lower 32-bits of the starting count value.

- *GIC_SH_CounterHi* register in Section 9.5.3.3, "GIC CounterHi". Used in conjunction with the *GIC_SH_CounterLo* register. Sets the upper 32-bits of the starting count value.

Compare Registers

The compare registers, GIC_VPE_CompareLo/Hi are located in the local section of the GIC memory map making the count specific to each processor. These registers can be written at any time. When the count value equals the compare value an Interval Timer interrupt is asserted. The interrupt is cleared (de-asserted) by writing to either GIC_VPE_CompareLo/Hi register. The compare registers are defined as follows:

- *GIC_COREi_CompareLo* register in Section 9.6.3.4, "CompareLo Register". Used in conjunction with the *GIC_COREi_CompareHi* register to set the count value at which an internal interrupt is generated.

- *GIC_COREi_CompareHi* register in Section 9.6.3.5, "Core-Local CompareHi Register". Used in conjunction with the *GIC_COREi_CompareLo* register to set the count value at which an internal interrupt is generated.

Determining the Counter Width

The counter used for GIC internal interrupt generation has a minimum width of 32 bits, meaning that all of the *GIC_SH_CounterLo* register is used. In the GIC design, the width of the *GIC_SH_CounterHi* register is also fixed at 32 bits as indicated by a value of 0x8 in the 4-bit COUNTBITS field in the *GIC_SH_CONFIG* register. To derive the total width of the counter, the following formula isused:

```
32 + COUNTBITS x 4
```

Where:

‘32’ is the width of the *GIC_SH_CounterLo* register and ‘COUNTBITS’ is the value in the COUNTBITS field of the *GIC_SH_CONFIG* register.

Since the COUNTBITS field contains a fixed value of 0x8, the overall width of the counter would be:

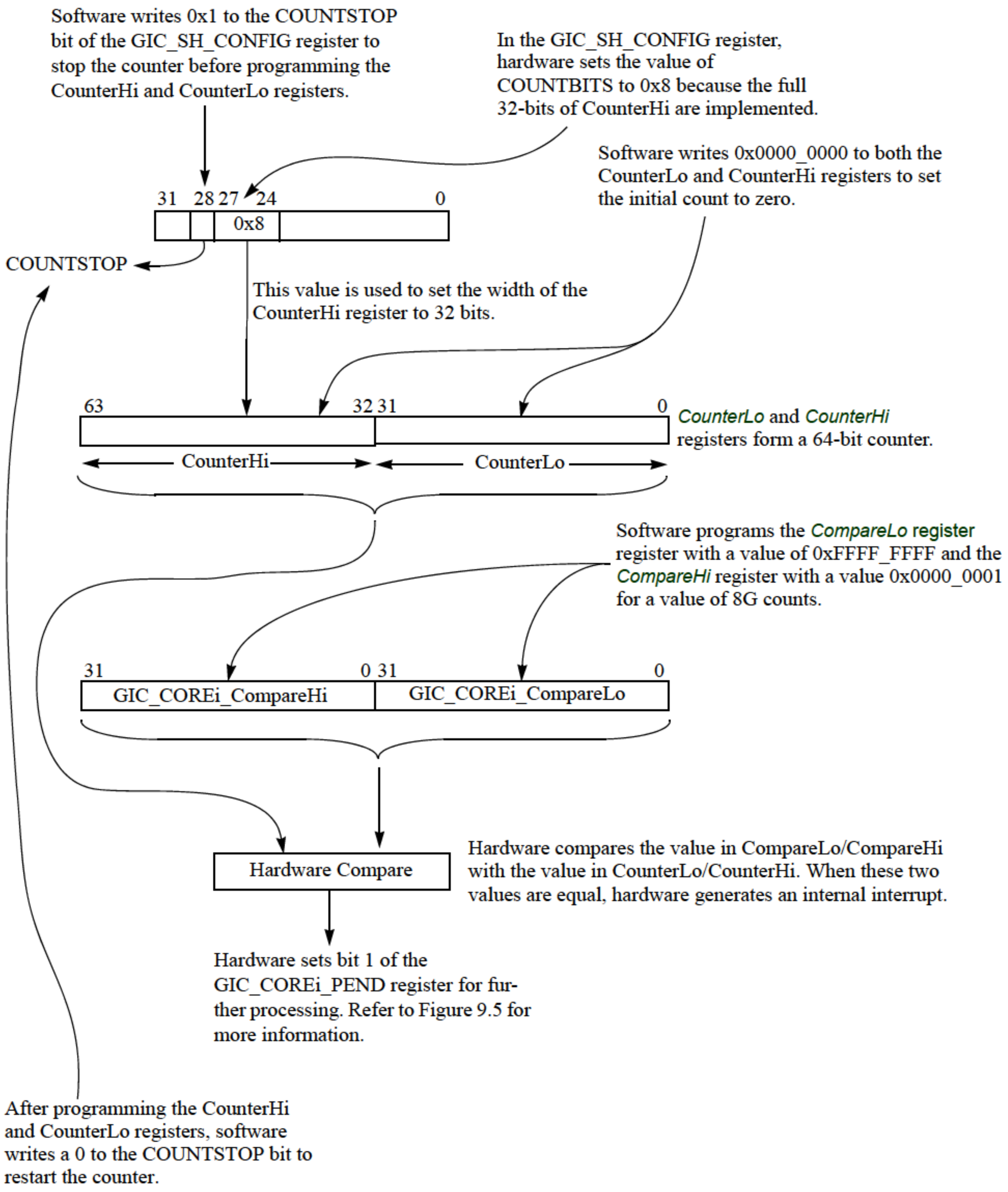
```
32 + 8 x 4 = 64 bits
```

In the GIC design, the COUNTBITS field is fixed at a value of 0x8, indicating a total counter size of 64-bits.

Counter Based Interrupt Example

In the example shown in Figure 9.4, the width of the counter is 64-bits, and the CompareLo/Hi value is 0x1_FFFF_FFFF which corresponds to 8G clock cycles. When this count is reached, hardware generates an internal interrupt.

Figure 9.4 Example of GIC Internal Counter-Based Interrupt Generation



Software writes 0x1 to the COUNTSTOP bit of the GIC_SH_CONFIG register to stop the counter before programming the CounterHi and CounterLo registers.

In the GIC_SH_CONFIG register, hardware sets the value of COUNTBITS to 0x8 because the full 32-bits of CounterHi are implemented.

Software writes 0x0000_0000 to both the CounterLo and CounterHi registers to set the initial count to zero.

This value is used to set the width of the CounterHi register to 32 bits.

CounterLo and *CounterHi* registers form a 64-bit counter.

Software programs the *CompareLo* register register with a value of 0xFFFF_FFFF and the *CompareHi* register with a value 0x0000_0001 for a value of 8G counts.

Hardware compares the value in CompareLo/CompareHi with the value in CounterLo/CounterHi. When these two values are equal, hardware generates an internal interrupt.

Hardware sets bit 1 of the GIC_COREi_PEND register for further processing. Refer to Figure 9.5 for more information.

After programming the CounterHi and CounterLo registers, software writes a 0 to the COUNTSTOP bit to restart the counter.

9.4.6.2 GIC Watchdog Timer

Each core supports a Watchdog timer that is controlled by the following three registers.

- The "GIC Watchdog Timer Configuration Register", GIC_COREi_WD_CONFIG is local to each processor and reports state information and configures the characteristics of the timer.
- The "Watchdog Timer Initial Count Register", GIC_COREi_WD_INITIAL is local to each processor and is used to set the timer interval.
- The "Watchdog Timer Count Register", GIC_COREi_WD_COUNT is a read only register local to each processor that contains the current value of the countdown.

GIC Watchdog Timer Configuration Register

The GIC Watchdog Timer Configuration register contains bits that control the function of the timer.

- Clearing the WAIT bit of GIC_COREi_WD_CONFIG register (default value) will cause the counter stop counting when the processor is executing a wait instruction or is in a low power stats controlled by the Cluster Power Controller. Setting this bit to 1 will cause it to continue counting down in these states. Usually this bit is left unset.
- Clearing the Debug bit (default value) will cause the counter to stop the count when the processor enters debug mode. When set the count will continue counting down. Usually this bit is left unset.
- The TYPE field in bits 3:1 of this register determines what happens when the timer reaches 0.

Table 9.9 GIC Watchdog Timer Modes

Encoding	Mode	Behavior
0x2	One Trip	An interrupt is asserted and the timer stops.
0x1	Second Countdown	An interrupt is asserted and the timer reloads. If the timer expires for the second time before being reloaded again all processors in the CPS will be reset. This mode provides a way to distinguish between a Software hang and a Hardware Hang. Usually the Watchdog Timer Interrupt is routed to NMI. This will cause the processor to soft reboot. In this mode that is what happens when the timer expires the first time so if this was a software hang during the reboot the software should reload the Watchdog Timer thus avoiding the second expiration. If the processor itself does not respond to the interrupt then it is assumed to be a hardware issue so when the count expires the second time a reset signal will be sent to all processors in the system.
0x3	Programmable Interval Timer	An interrupt is asserted, the initial count is reloaded and the time starts counting down again interrupting each time the counter reaches 0. This mode provides a per processor interval timer. This is one mode where the interrupt should not be routed to NMI. It should instead be routed to a normal interrupt where for example the interrupt could be used in a time slicing OS.

Clearing the WDEN bit disables the timer and when it is set it enables the timer. Writing WDEN with a 1 triggers a reloads the GIC_CORE_WD_COUNT register with the value in the GIC_COREi_WD_INITIAL register. Refer to Section 9.6.3.1, "Watchdog Timer Config Register" for more information.

Watchdog Timer Initial Count Register

The "Watchdog Timer Initial Count Register", GIC_COREi_WD_INITIAL is local to each processor and is used to set the timer interval. To start the counter for the first time the counter should be disabled by clearing the WDEN bit in the GIC_COREi_WD_CONFIG register and the countdown value loaded into this register and then the counter enabled by setting the WDEN bit. Refer to Section 9.6.3.3, "Watchdog Timer Initial Count Register" for more information.

Watchdog Timer Count Register

The "Watchdog Timer Count Register", GIC_CORE_WD_COUNT is a read only register local to each processor that contains the current value of the countdown. This register is reloaded with the value in the GIC_COREi_WD_INITIAL register each time the WDEN bit in the GIC_COREi_WD_CONFIG register is set. Refer to Section 9.6.3.2, "Watchdog Timer Count Register" for more information.

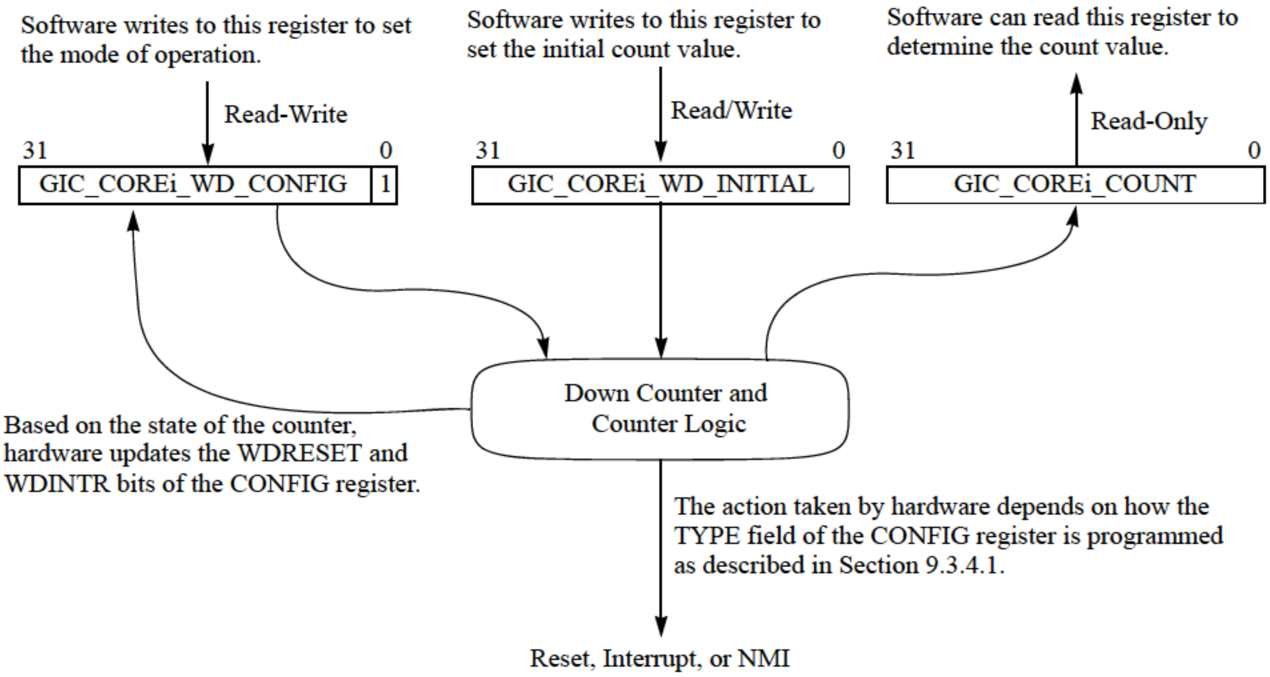
Configuring the Watchdog Timer

Software can configure the WatchDog timer with a starting count value by programming the *WatchDog Timer Initial Count* register (*GIC_COREi_WD_INITIAL*) located at offset address 0x0098. Refer to Section 9.6.3.3 "Watchdog Timer Initial Count Register" for more information.

Software can read the state of the count at any time by reading the the *WatchDog Timer Count* register (*GIC_COREi_WD_COUNT*) located at offset address 0x0094. Refer to Section 9.6.3.2 "Watchdog Timer Count Register" for more information.

Figure 9.7 shows the timer counter configuration process.

Figure 9.5 Local Watchdog Timer Interrupt Count Configuration


Software writes to this register to set the mode of operation.
Read-Write
31
0
GIC_COREi_WD_CONFIG
1
Software writes to this register to set the initial count value.
Read/Write
31
0
GIC_COREi_WD_INITIAL
Software can read this register to determine the count value.
Read-Only
31
0
GIC_COREi_COUNT
Down Counter and Counter Logic
Based on the state of the counter, hardware updates the WDRESET and WDINTR bits of the CONFIG register.
The action taken by hardware depends on how the TYPE field of the CONFIG register is programmed as described in Section 9.3.4.1.
Reset, Interrupt, or NMI

Watchdog Timer Masking and Mapping

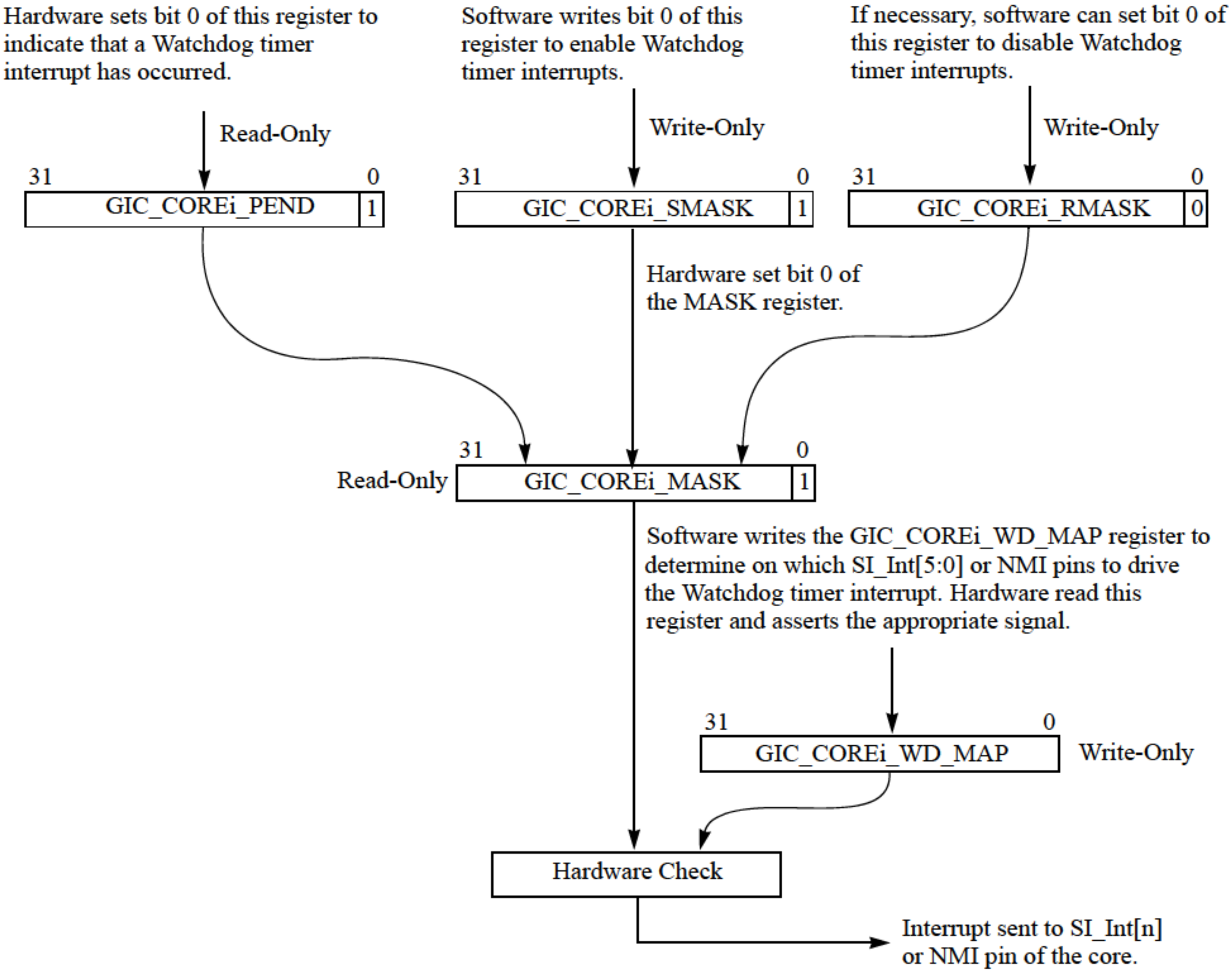
Figure 9.5 above shows the process used to configure the Watchdog timer. Once a Watchdog timer interrupt is generated (output of Figure 9.5), hardware sets bit 0 of the *Local Interrupt Pending* register (*GIC_COREi_PEND*) at offset address 0x0004. Hardware then reads the state of bit 0 in the *Local Interrupt Mask* register (*GIC_COREi_MASK*) at offset address 0x0008 to determine whether the Watchdog timer interrupt has been masked. The *GIC_COREi_MASK* register is a read-only register.

Software can affect the state of this register using the write-only *Local Interrupt Set Mask* register (*GIC_COREi_SMASK*) at offset address 0x0010 and the *Local Interrupt Reset Mask* register (*GIC_COREi_RMASK*) at offset address 0x000C. Software sets bit 0 of the *SMASK* register to enable the Watchdog timer interrupt, or it can set bit 0 of the *RMASK* register to disable Watchdog timer interrupts. Note that when the WatchDog timer is programmed to generate a hardware reset, the reset cannot be masked by the *Local Interrupt Mask* register

Once hardware has determine the masking characteristics of the interrupt, it uses the *Watchdog Timer Map-to-Pin* register at offset address 0x0040 to determine which *SI_Int[5:0]*, or *NMI* pins the interrupt will be driven onto. In non-EIC mode, bits 5:0 of this register are used to select one of 6 core interrupts. For example, if software programs this field with a value of 0x2, then the Watchdog timer interrupt will be driven into *SI_Int[2]*. In non-EIC mode, only encodings 0 - 5 are valid.

In EIC mode, the core encodes this field to support up to 64 interrupts. For example, if software programs this field with a value of 0x20, then the Watchdog timer interrupt corresponds to interrupt 33. This encoded value is then driven onto *SI_Int[5:0]*.

Figure 9.6 Watchdog Timer Interrupt Masking and Mapping in the GIC


Hardware sets bit 0 of this register to indicate that a Watchdog timer interrupt has occurred.
Read-Only
31
0
GIC_COREi_PEND
1
Software writes bit 0 of this register to enable Watchdog timer interrupts.
Write-Only
31
0
GIC_COREi_SMASK
1
If necessary, software can set bit 0 of this register to disable Watchdog timer interrupts.
Write-Only
31
0
GIC_COREi_RMASK
0
Hardware set bit 0 of the MASK register.
31
0
Read-Only
GIC_COREi_MASK
1
Software writes the GIC_COREi_WD_MAP register to determine on which SI_Int[5:0] or NMI pins to drive the Watchdog timer interrupt. Hardware read this register and asserts the appropriate signal.
31
0
GIC_COREi_WD_MAP
Write-Only
Hardware Check
Interrupt sent to SI_Int[n] or NMI pin of the core.

Watchdog Timer and Debug Mode

Under certain conditions, software may want to suspend Watchdog timer operation while the proAptiv Multiprocessing System is in debug mode. This can be accomplished by clearing the DEBUGMODE_CTRL bit of the *Watchdog Timer Config* register located at offset address 0x0090. When this bit is cleared, counting is stopped. Note that the DM bit of the CP0 *Debug* register ($DEBUG_{DM}$) must be set to place the device in debug mode.

If this bit is set by software, entering debug mode has no effect on the Watchdog timer counting process.

Watchdog Timer and Low Power Mode

Under certain conditions, software may want to suspend Watchdog timer operation while the proAptiv Multiprocessing System is in low power mode. This can be accomplished by clearing the WAITMODE_CTRL bit of the *Watchdog Timer Config* register located at offset address 0x0090. When this bit is cleared, counting is stopped (including when low power mode is entered via the WAIT instruction.

If this bit is set by software, entering low power mode has no effect on the Watchdog timer counting process.

9.4.7 Local Interrupt Routing

9.4.7.1 Routability of Local Interrupts

Local interrupts (except for the Watchdog timer, GIC Interval Timer and software interrupts) can be hardwired to local pins when the CPS is configured or can be more flexible and left to software to route the local interrupts to local pins on the processor. The "Local Interrupt Control Register", GIC_COREi_CTL (link to register reference of GIC_COREi_CTL) reports the routable state of the local interrupts. If the bit for the particular interrupt is set then the interrupt is routable within the GIC. The following table describes the behavior if not set.

Bits 4:1 of the *GIC_COREi_CTL* register determines the routing of the following interrupts. In the proAptiv GIC design, these bits are hard-wired to 1. Note that Software Interrupts from the core are routed internally by the CPU in vectored interrupt mode, and are only routed through the GIC when the GIC is in EIC mode, regardless of the *GIC_COREi_CTL* register.

Table 9.10 GIC_COREi_CTL Register Fields

Bit Field Name	Behavior if cleared
FDC_ROUTABLE	The CPU Fast Debug Channel Interrupt is hardwired to one of the SI_Int pins as described by the CPU's COP0 IntCtlIPFDCI register field.
SWINT_ROUTABLE	The CPU SW Interrupts are routed back to the CPU directly.
PERFCOUNT_ROUTABLE	The CPU Performance Counter Interrupt is hardwired to one of SI_Int pins as described by the CPU's COP0 IntCtlIPPCI register field.
TIMER_ROUTABLE	The CPU Timer Interrupt is hardwired to one of the SI_Int pins, as described by the CPU's COP0 IntCtlIPTI register field

9.4.7.2 Routing Local Interrupts

If a local interrupt is routable it can be routed to a local signal of the local processor, much the same as an external interrupt.

There is a Local Interrupt Map to Pin Register (link to register reference of Local WatchDog Timer/Compare/CPU Timer/PerfCount/SWInt0-1 Map to Pin Registers) for each local interrupt source that further maps the local interrupt to a specific input on the processor. There are two bits, MAP_TO_PIN and MAP_TO_NMI that control the type of input that is assigned to the interrupt source. Only one of these bits can be set at any one time.

- If set the MAP_TO_PIN bit will map the local interrupt source to Interrupt Pending bits in the CP0 Cause register of the processor. The actual Interrupt Pending bit is set in the MAP field of this register. The MAP Field of this register contains the encoded value of the number (0 -63). For example, a value of 0x20 asserts Interrupt 32 (decimal). For vectored interrupt mode, only use values of 0x0 to 0x5.
- If set the bit will map the local interrupt source to the NMI bit in the CP0 Status register. This in essence will cause the processor to soft boot using the boot exception vector as the start of the interrupt routine.

Each of these interrupt types is described in the following subsections. Table 9.12 lists the registers and associated bits that would be programmed to facilitate each type of interrupt listed above.

Table 9.11 Local Interrupt Masking and Mapping Register Usage Per Interrupt Type

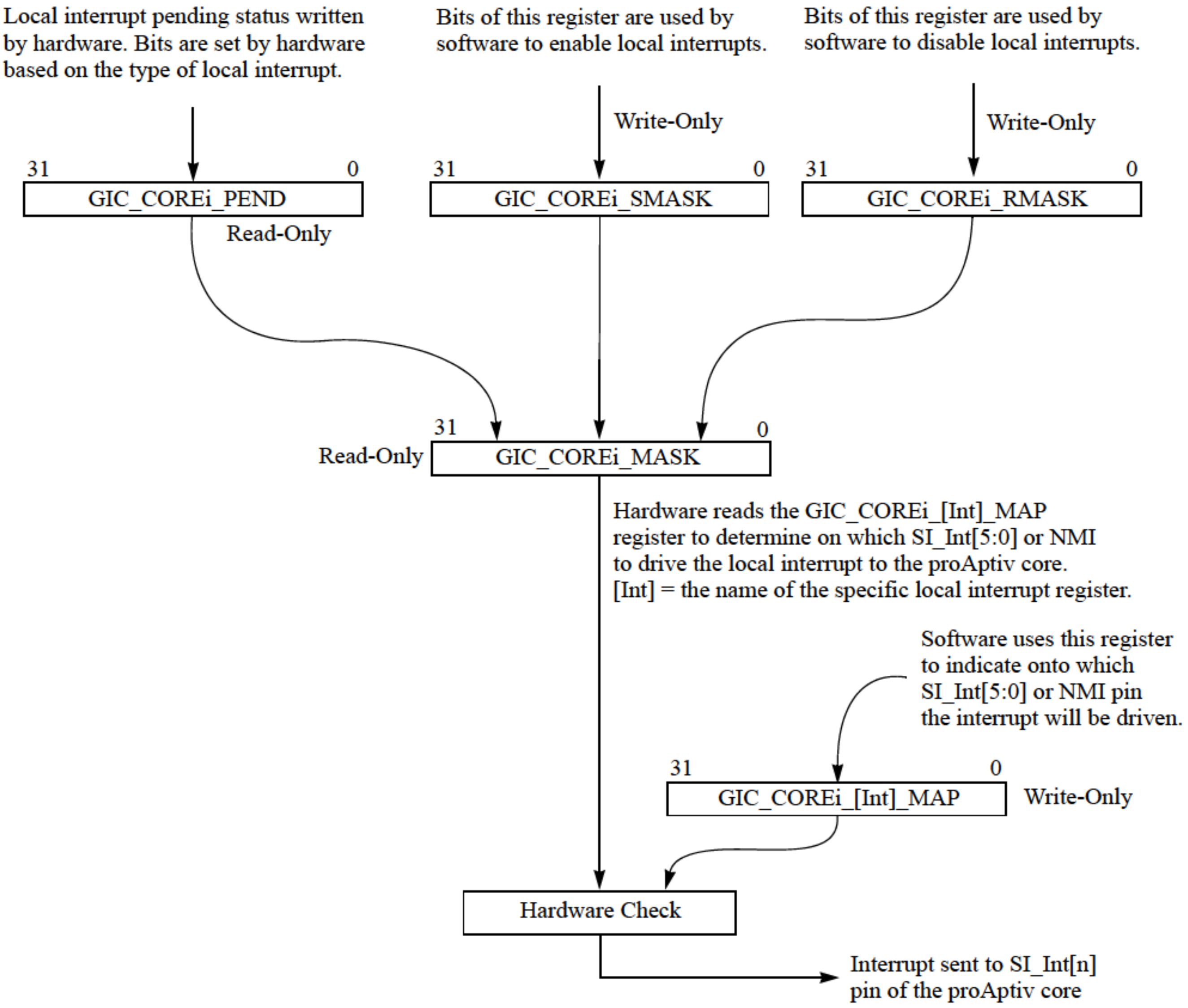
Interrupt	Register Name	Offset	Bits Used	Function
WatchDog	GIC_COREi_PEND	0x0004	0	Set by hardware on a local WatchDog timer interrupt.
	GIC_COREi_MASK	0x0008	0	Set by hardware based on the state of bit 0 of the SMASK and RMASK registers. Used to determine whether the interrupt will be processed or ignored.
	GIC_COREi_RMASK	0x000C	0	Used by software to disable WatchDog timer interrupts.
	GIC_COREi_SMASK	0x0010	0	Used by software to enable WatchDog timer interrupts.
	GIC_COREi_WD_MAP	0x0040	31, 5:0	Used by software to map the WatchDog timer interrupt to one of the SI_Int[5:0] pins of the proAptiv Multiprocessing System core.
Count and Compare	GIC_COREi_PEND	0x0004	1	Set by hardware on a local Count/Compare interrupt.
	GIC_COREi_MASK	0x0008	1	Set by hardware based on the state of bit 1 of the SMASK and RMASK registers. Used to determine whether the interrupt will be processed or ignored.
	GIC_COREi_RMASK	0x000C	1	Used by software to disable Count/Compare interrupts.
	GIC_COREi_SMASK	0x0010	1	Used by software to enable Count/Compare interrupts.
	GIC_COREi_ COMPARE_MAP	0x044	31, 5:0	Used by software to map the Count/Compare interrupt to one of the SI_Int[5:0] pins of the proAptiv Multiprocessing System core.
Timer	GIC_COREi_PEND	0x0004	2	Set by hardware on a local timer interrupt.
	GIC_COREi_MASK	0x0008	2	Set by hardware based on the state of bit 2 of the SMASK and RMASK registers. Used to determine whether the interrupt will be processed or ignored.
	GIC_COREi_RMASK	0x000C	2	Used by software to disable timer interrupts.
	GIC_COREi_SMASK	0x0010	2	Used by software to enable timer interrupts.
	GIC_COREi_ TIMER_MAP	0x048	31, 5:0	Used by software to map the timer interrupt to one of the SI_Int[5:0] pins of the proAptiv Multiprocessing System core.
Performance Counter	GIC_COREi_PEND	0x0004	3	Set by hardware on a performance counter interrupt.
	GIC_COREi_MASK	0x0008	3	Set by hardware based on the state of bit 3 of the SMASK and RMASK registers. Used to determine whether the interrupt will be processed or ignored.
	GIC_COREi_RMASK	0x000C	3	Used by software to disable performance counter interrupts.
	GIC_COREi_SMASK	0x0010	3	Used by software to enable performance counter interrupts.
	GIC_COREi_ PERFCTR_MAP	0x0050	31, 5:0	Used by software to map the performance counter interrupt to one of the SI_Int[5:0] pins of the proAptiv Multiprocessing System core.

Table 9.11 Local Interrupt Masking and Mapping Register Usage Per Interrupt Type *(continued)*

Interrupt	Register Name	Offset	Bits Used	Function
Software Interrupt 0	GIC_COREi_PEND	0x0004	4	Set by hardware on a software interrupt 0 occurrence.
	GIC_COREi_MASK	0x0008	4	Set by hardware based on the state of bit 4 of the SMASK and RMASK registers. Used to determine whether the interrupt will be processed or ignored.
	GIC_COREi_RMASK	0x000C	4	Used by software to disable software interrupt 0 interrupts.
	GIC_COREi_SMASK	0x0010	4	Used by software to enable software interrupt 0 interrupts.
	GIC_COREi_ SWInt0_MAP	0x0054	31, 5:0	Used by software to map software interrupt 0 to one of the SI_Int[5:0] pins of the proAptiv Multiprocessing System core.
Software Interrupt 1	GIC_COREi_PEND	0x0004	5	Set by hardware on a software interrupt 1 occurrence.
	GIC_COREi_MASK	0x0008	5	Set by hardware based on the state of bit 5 of the SMASK and RMASK registers. Used to determine whether the interrupt will be processed or ignored.
	GIC_COREi_RMASK	0x000C	5	Used by software to disable software interrupt 1 interrupts.
	GIC_COREi_SMASK	0x0010	5	Used by software to enable software interrupt 1 interrupts.
	GIC_COREi_ SWInt1_MAP	0x0058	31, 5:0	Used by software to map software interrupt 1 to one of the SI_Int[5:0] pins of the proAptiv Multiprocessing System core.
Fast Debug Channel	GIC_COREi_PEND	0x0004	6	Set by hardware on a Fast Debug Channel (FDC) interrupt.
	GIC_COREi_MASK	0x0008	6	Set by hardware based on the state of bit 6 of the SMASK and RMASK registers. Used to determine whether the interrupt will be processed or ignored.
	GIC_COREi_RMASK	0x000C	6	Used by software to disable FDC interrupts.
	GIC_COREi_SMASK	0x0010	6	Used by software to enable FDC interrupts.
	GIC_COREi_FDC_MAP	0x004C	31, 5:0	Used by software to map the FDC interrupt to one of the SI_Int[5:0] pins of the proAptiv Multiprocessing System core.

The general overview of the local interrupt peding, masking, and mapping process is shown in Figure 9.7.

Figure 9.7 Local Interrupt Masking and Mapping in the GIC


Local interrupt pending status written by hardware. Bits are set by hardware based on the type of local interrupt.
Bits of this register are used by software to enable local interrupts.
Bits of this register are used by software to disable local interrupts.
Write-Only
Write-Only
31
0
GIC_COREi_PEND
Read-Only
31
0
GIC_COREi_SMASK
31
0
GIC_COREi_RMASK
31
0
Read-Only
GIC_COREi_MASK
Hardware reads the GIC_COREi_[Int]_MAP register to determine on which SI_Int[5:0] or NMI to drive the local interrupt to the proAptiv core. [Int] = the name of the specific local interrupt register.
Software uses this register to indicate onto which SI_Int[5:0] or NMI pin the interrupt will be driven.
31
0
GIC_COREi_[Int]_MAP
Write-Only
Hardware Check
Interrupt sent to SI_Int[n] pin of the proAptiv core

Each of the registers listed in Figure 9.7 above can be found in the following sections:

- Section 9.6.2.2 “Local Interrupt Pending Register”
- Section 9.6.2.5 “Local Interrupt Set Mask Register”
- Section 9.6.2.4 “Local Interrupt Reset Mask Register”
- Section 9.6.2.3 “Local Interrupt Mask Register”
- Section 9.6.2.6 “Local Map to Pin Registers”

9.4.7.3 Watchdog Timer Interrupts

For more information, refer to Section 9.4.6.2, "GIC Watchdog Timer".

9.4.7.4 Count and Compare Interrupts

A count and compare interrupt occurs when the contents of the of *GIC_COREi_CompareLo* and *GIC_COREi_CompareHi* registers match the contents of *GIC_SH_CounterLo* and *GIC_SH_CounterHi*, the Count/Compare interrupt is triggered. Refer to Section "Counter Based Interrupt Example" for more information.

When a count and compare interrupt is generated, hardware sets bit 1of the *Local Interrupt Pending* register (*GIC_COREi_PEND*) at offset address 0x0004. Hardware then reads the state of bit 1 in the *Local Interrupt Mask* register (*GIC_COREi_MASK*) at offset address 0x0008 to determine whether the count and compare interrupt has been masked. The *GIC_COREi_MASK* register is a read-only register.

Software can affect the state of this register using the write-only *Local Interrupt Set Mask* register (*GIC_COREi_SMASK*) at offset address 0x0010 and the *Local Interrupt Reset Mask* register (*GIC_COREi_RMASK*) at offset address 0x000C. Software sets bit 1 of the *SMASK* register to enable the count and compare interrupt, or it can set bit 1 of the *RMASK* register to disable count and compare interrupts.

Once hardware has determined the masking characteristics of the interrupt, it uses the *Count/Compare Map-to-Pin* register at offset address 0x0044 to determine which *SI_Int[5:0] or NMI* pins the interrupt will be driven onto. In vectored interrupt mode, bits 5:0 of this register are used to select one of 6 core interrupts. In this mode, only encodings 0 - 5 are valid. In EIC mode, the core encodes this field to support up to 63 interrupts. For example, if software programs this field with a value of 0x20, then the WatchDog timer interrupt corresponds to interrupt level 32. This encoded value is then driven onto *SI_Int[5:0]*.

9.4.7.5 Timer Interrupts

When a timer interrupt is generated, hardware sets bit 2 of the *Local Interrupt Pending* register (*GIC_COREi_PEND*) at offset address 0x0004. Hardware then reads the state of bit 2 in the *Local Interrupt Mask* register (*GIC_COREi_MASK*) at offset address 0x0008 to determine whether the timer interrupt has been masked. The *GIC_COREi_MASK* register is a read-only register.

Software can affect the state of this register using the write-only *Local Interrupt Set Mask* register (*GIC_COREi_SMASK*) at offset address 0x0010 and the *Local Interrupt Reset Mask* register (*GIC_COREi_SMASK*) at offset address 0x000C. Software sets bit 2 of the *SMASK* register to enable the timer interrupt, or it can set bit 2 of the *RMASK* register to disable timer interrupts.

Once hardware has determine the masking characteristics of the interrupt, it uses the *Timer Map-to-Pin* register at offset address 0x0048 to determine which *SI_Int[5:0] or NMI* pins the interrupt will be driven onto. In non-EIC mode, bits 5:0 of this register are used to select one of 6 core interrupts. In non-EIC mode, only encodings 0 - 5 are valid. In EIC mode, the core encodes this field to support up to 63 interrupts.

9.4.7.6 Performance Counter Interrupts

When a timer interrupt is generated, hardware sets bit 3 of the *Local Interrupt Pending* register (*GIC_COREi_PEND*) at offset address 0x0004. Hardware then reads the state of bit 3 in the *Local Interrupt Mask* register (*GIC_COREi_MASK*) at offset address 0x0008 to determine whether the performance counter interrupt has been masked. The *GIC_COREi_MASK* register is a read-only register.

Software can affect the state of this register using the write-only *Local Interrupt Set Mask* register (*GIC_COREi_SMASK*) at offset address 0x0010 and the *Local Interrupt Reset Mask* register (*GIC_COREi_SMASK*) at offset address 0x000C. Software sets bit 3 of the *SMASK* register to enable the performance counter interrupt, or it can set bit 3 of the *RMASK* register to disable timer interrupts.

Once hardware has determine the masking characteristics of the interrupt, it uses the *Performance Counter Map-to-Pin* register at offset address 0x0050 to determine which *SI_Int[5:0] or NMI* pins the interrupt will be driven onto. In non-EIC mode, bits 5:0 of this register are used to select one of 6 core interrupts. In non-EIC mode, only encodings 0 - 5 are valid. In EIC mode, the core encodes this field to support up to 63 interrupts.

9.4.7.7 Software Interrupts

Each core provides two software interrupts; 0 and 1. Software interrupts originiate from the CPU and are only used in EIC mode. In non-EIC mode they are routed internally.

When software interrupt 0 is generated, hardware sets bit 4 of the *Local Interrupt Pending* register (*GIC_COREi_PEND*) at offset address 0x0004. Hardware then reads the state of bit 4 in the *Local Interrupt Mask* register (*GIC_COREi_MASK*) at offset address 0x0008 to determine whether the performance counter interrupt has been masked. The *GIC_COREi_MASK* register is a read-only register.

Software can affect the state of this register using the write-only *Local Interrupt Set Mask* register (*GIC_COREi_SMASK*) at offset address 0x0010 and the *Local Interrupt Reset Mask* register (*GIC_COREi_SMASK*) at offset address 0x000C. Software sets bit 4 of the *SMASK* register to enable the software interrupt 0, or it can set bit 4 of the *RMASK* register to disable software interrupt 0.

Once hardware has determine the masking characteristics of the interrupt, it uses the *Software Interrupt 0 Map-to-Pin* register at offset address 0x0054 to determine which *SI_Int[5:0] or NMI* pins the interrupt will be driven onto. In non-EIC mode, bits 5:0 of this register are used to select one of 6 core interrupts. In non-EIC mode, only encodings 0 - 5 are valid. In EIC mode, the core encodes this field to support up to 63 interrupts.

The sequence is the same for software interrupt 1, except that bit 5 of each register noted above is set instead of bit 4. In addition, software uses the *Software Interrupt 1 Map-to-Pin* register at offset address 0x0058 to determine which *SI_Int[5:0]* pin the interrupt will be driven onto.

9.4.7.8 Fast Debug Channel Interrupts

When a Fast Debug Channel (FDC) interrupt is generated, hardware sets bit 6 of the *Local Interrupt Pending* register (*GIC_COREi_PEND*) at offset address 0x0004. Hardware then reads the state of bit 6 in the *Local Interrupt Mask* register (*GIC_COREi_MASK*) at offset address 0x0008 to determine whether the fast debug channel interrupt has been masked. The *GIC_COREi_MASK* register is a read-only register.

Software can affect the state of this register using the write-only *Local Interrupt Set Mask* register (*GIC_COREi_SMASK*) at offset address 0x0010 and the *Local Interrupt Reset Mask* register (*GIC_COREi_SMASK*) at offset address 0x000C. Software sets bit 6 of the *SMASK* register to enable the fast debug channel interrupt, or it can set bit 6 of the *RMASK* register to disable fast debug channel interrupts.

Once hardware has determine the masking characteristics of the interrupt, it uses the *Fast Debug Channel Map-to-Pin* register at offset address 0x004C to determine which *SI_Int[5:0] or NMI* pins the interrupt will be driven onto. In non-EIC mode, bits 5:0 of this register are used to select one of 6 core interrupts. In non-EIC mode, only encodings 0 - 5 are valid. In EIC mode, the proAptiv Multiprocessing System core encodes this field to support up to 63 interrupts.

9.4.8 EIC Mode Setting

EIC mode is controlled through software by setting the EIC_MODE bit in the Local interrupt Control Register, GIC_VPE_CTL (link to register reference of GIC_VPE_CTL). Setting this bit enables EIC mode. This bit defaults to 0, vectored interrupt mode.

9.4.9 Enabling, Disabling, and Polling Local Interrupts

The Enabling, Disabling and Polling of local interrupts is configured through several registers in the GIC that are local to each processor.

There are 4 registers for Enabling, Disabling and Polling of local interrupts.

- Enabling an interrupt using the "GIC Local Set Mask Registers", GIC_VPE_SMASK
- Disabling an interrupt using the "GIC Local Reset Mask Registers", GIC_VPE_RMASK
- Determining the Enable/Disable state of an interrupt state using "GIC Local Interrupt Mask Register", GIC_VPE_MASK
- Polling the interrupt active state using the "GIC Local Interrupt Pending Register", GIC_VPE_PEND

9.4.9.1 Enabling External Interrupts

The "GIC Local Set Mask Register", GIC_VPE_SMASK is used to enable individual local interrupts. For synchronization purposes this is a write only register. Setting the bit enables the interrupt. The following table shows which field to set for each local interrupt. Refer to Section 9.6.2.5 "Local Interrupt Set Mask Register" for more information.

Table 9.12 Enabling External Interrupts

Field Name	Interrupt Controlled
FDC_MASK_SET	Fast Debug Channel
SWINT1_MASK_SET	Software interrupt 1
SWINT2_MASK_SET	Software interrupt 2
PERFCOUNT_MASK_SET	Local Performance Counter
TIMER_MASK_SET	CP0 Local Count/Compare Timer
COMPARE_MASK_SET	GIC Local Count/Compare Timer
WD_MASK_SET	Watchdog

9.4.9.2 Disabling External Interrupts

The "GIC Local Reset Mask Register", GIC_VPE_RMASK is used to disable individual local interrupts. For CPS synchronization purposes this is a write only register. Setting the bit disables the interrupt. The following table shows which field to set for each local interrupt. Refer to Section 9.6.2.4 "Local Interrupt Reset Mask Register" for more information.

Table 9.13 Disabling External Interrupts

Field Name	Interrupt Controlled
FDC_RESET_MASK	Fast Debug Channel
SWINT1_RESET_MASK	Software interrupt 1
SWINT2_RESET_MASK	Software interrupt 2
PERFCOUNT_RESET_MASK	Local Performance Counter
TIMER_RESET_MASK	CP0 Local Count/Compare Timer
COMPARE_RESET_MASK	GIC Local Count/Compare Timer
WD_RESET_MASK	Watchdog

9.4.9.3 Determining the Enabled or Disabled Interrupt state

The "GIC Local Mask Register", GIC_VPE_MASK is used to determine if a local interrupt is enabled. For CPS synchronization purposes this is a read only register. If a bit is set the corresponding interrupt source is enabled. If it is clear the corresponding interrupt is disabled. The following table shows which field corresponds to each local interrupt. Refer to Section 9.6.2.3 “Local Interrupt Mask Register” for more information

Table 9.14 Determining the Enabled of Disabled Interrupt State

Field Name	Interrupt Controlled
FDC_MASK	Fast Debug Channel
SWINT1_MASK	Software interrupt 1
SWINT2_MASK	Software interrupt 2
PERFCOUNT_MASK	Local Performance Counter
TIMER_MASK	CP0 Local Count/Compare Timer
COMPARE_MASK	GIC Local Count/Compare Timer
WD_MASK	Watchdog

9.4.9.4 Polling for an Active Interrupt

The "GIC Pending Register", GIC_VPE_PEND is used to determine if a external interrupt is active. This is a read only register. If a bit is set the corresponding local interrupt is active. If it is clear the corresponding interrupt is inactive. The following table shows which field corresponds to each local interrupt. Refer to Section 9.6.2.2 “Local Interrupt Pending Register” for more information

Table 9.15 Polling for an Active Interrupt

Field Name	Interrupt Controlled
FDC_PEND	Fast Debug Channel
SWINT1_PEND	Software interrupt 1
SWINT2_PEND	Software interrupt 2
PERFCOUNT_PEND	Local Performance Counter

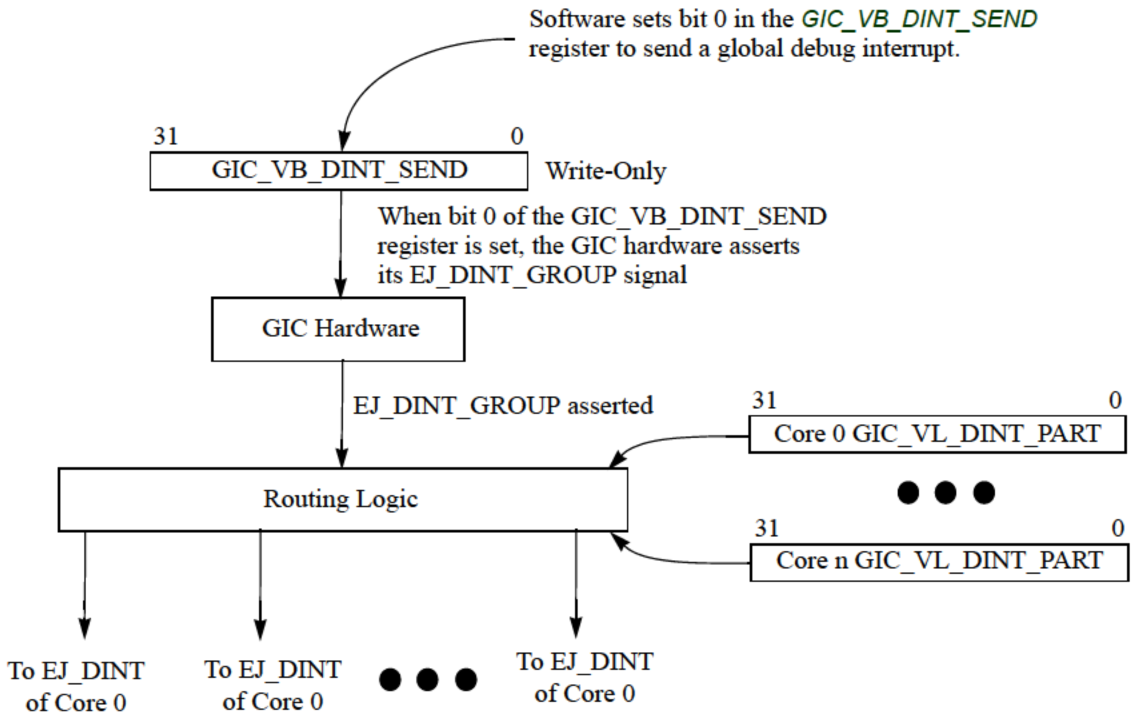
Table 9.15 Polling for an Active Interrupt

Field Name	Interrupt Controlled
TIMER_PEND	CP0 Local Count/Compare Timer
COMPARE_PEND	GIC Local Count/Compare Timer
WD_PEND	Watchdog

9.4.10 Debug Interrupt Generation

The GIC of the proAptiv Multiprocessing System allows software to globally assert a debug interrupt to all cores in the system. When the *Send_DINT* bit of the *DINT Send to Group* register (GIC_VB_DINT_SEND) in Section 9.5.3.15, "DINT Send to Group Register" is set, the *EJ_DINT_GROUP* signal of the GIC is asserted. Based on the state of this signal and the Core-Local GIC_VL_DINT_PART registers, hardware asserts the EJ_DINT signal of each core in the system. This concept is shown in Figure 9.8.

Figure 9.8 Global EJTAG Debug Interrupt Generation in the GIC



9.5 Shared Register Set

This section describes the various registers in the Shared register set.

9.5.1 GIC Register Field Types

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

Table 9.16 CP0 Register Field Types

Notation	Hardware Interpretation	Software Interpretation
R/W	A field in which all bits are readable and writable by software and, potentially, by hardware. 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.	
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 power up. If the Reset State of this field is “Undefined”, hardware updates this field only under those conditions specified in the description of the field.	A field 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 can not be read by software. Software reads of this field will return an **UNDEFINED** value.	
0	A field that hardware does not update, and for which hardware can assume a zero value.	A field to which the value written by software must be zero. Software writes of non-zero values to this field may result in **UNDEFINED** behavior of the hardware. Software reads of this field return zero as long as all previous software writes are zero. If the Reset State of this field is “Undefined,” software must write this field with zero before it is guaranteed to read as zero.

9.5.2 Shared Section Register Map

The register map of the shared section is shown in Table 9.18. These registers are accessible by any core. For the base address of this block, see Table 9.3.

All registers are 32 bits wide and should only be accessed using 32-bit uncached load/stores. Reads from unpopulated registers in the GCMP address space should return 0x0, and writes to those locations should be silently dropped without generating any exceptions.

The addresses for the registers within the Shared Section of the GIC are calculated as follows:

```
SharedSection_Register_Physical_Address=GIC_baseaddress+SharedSection_baseoffset+Re
gister_Offset
```

Table 9.17 Shared Section Register Map

Register Offset	Name	Type	Description
0x0000	GIC Config Register (`GIC_SH_CONFIG`)	R	Indicates the number of interrupts, number of cores, etc.
0x0010	GIC CounterLo (`GIC_SH_CounterLo`)	R/W	Shared Global Counter.
0x0014	GIC CounterHi (`GIC_SH_CounterHi`)	R/W	
0x0020	GIC Revision Register (`GIC_RevisionID`)	R	RevisionID of the GIC hardware.
0x0100	Global Interrupt Polarity Register0 (`GIC_SH_POL31_0`)	R/W	Polarity of the interrupt. For Level Type: 0x0 - Active Low 0x1 - Active High For Single Edge Type: 0x0 - Falling Edge used to set edge register 0x1 - Rising Edge used to set edge register At IP configuration time, the appropriate number of these registers are instantiated to support the number of External Interrupt Sources.
0x0104	Global Interrupt Polarity Register1 (`GIC_SH_POL63_32`)	R/W	
0x0108	Global Interrupt Polarity Register2 (`GIC_SH_POL95_64`)	R/W	
0x010c	Global Interrupt Polarity Register3 (`GIC_SH_POL127_96`)	R/W	
0x0110	Global Interrupt Polarity Register4 (`GIC_SH_POL159_128`)	R/W	
0x0114	Global Interrupt Polarity Register5 (`GIC_SH_POL191_160`)	R/W	
0x0118	Global Interrupt Polarity Register6 6(`GIC_SH_POL223_192`)	R/W	
0x011c	Global Interrupt Polarity Register7 (`GIC_SH_POL255_224`)	R/W	

Table 9.17 Shared Section Register Map *(continued)*

Register Offset	Name	Type	Description
0x0180	Global Interrupt Trigger Type Register0 (GIC_SH_TRIG31_0)	R/W	Edge or Level triggered 0x0 - Level 0x1 - Edge At IP configuration time, the appropriate number of these registers are instantiated to support the number of External Interrupt Sources.
0x0184	Global Interrupt Trigger Type Register1 (GIC_SH_TRIG63_32)	R/W	
0x0188	Global Interrupt Trigger Type Register2 (GIC_SH_TRIG95_64)	R/W	
0x018c	Global Interrupt Trigger Type Register3 (GIC_SH_TRIG127_96)	R/W	
0x0190	Global Interrupt Trigger Type Register4 (GIC_SH_TRIG159_128)	R/W	
0x0194	Global Interrupt Trigger Type Register5 (GIC_SH_TRIG191_160)	R/W	
0x0198	Global Interrupt Trigger Type Register6 (GIC_SH_TRIG223_192)	R/W	
0x019c	Global Interrupt Trigger Type Register7 (GIC_SH_TRIG255_224)	R/W	
0x0200	Global Interrupt Dual Edge Register (GIC_SH_DUAL31_0)	R/W	Writing a 0x1 to any bit location sets the appropriate external interrupt source to be type dual-edged. At IP configuration time, the appropriate number of these registers are instantiated to support the number of External Interrupt Sources.
0x0204	Global Interrupt Dual Edge Register (GIC_SH_DUAL63_32)	R/W	
0x0208	Global Interrupt Dual Edge Register (GIC_SH_DUAL95_64)	R/W	
0x020c	Global Interrupt Dual Edge Register (GIC_SH_DUAL127_96)	R/W	
0x0210	Global Interrupt Dual Edge Register (GIC_SH_DUAL159_128)	R/W	
0x0214	Global Interrupt Dual Edge Register (GIC_SH_DUAL191_160)	R/W	
0x0218	Global Interrupt Dual Edge Register (GIC_SH_DUAL223_192)	R/W	
0x021c	Global Interrupt Dual Edge Register (GIC_SH_DUAL255_224)	R/W	
0x0280	Global Interrupt Write Edge Register (GIC_SH_WEDGE)	W	Used for Interrupt Messages. Writes to this register atomically set or clear a specified bit in the *Edge Detect Register*.

Table 9.17 Shared Section Register Map *(continued)*

Register Offset	Name	Type	Description
0x0300	Global Interrupt Reset Mask Register (`GIC_SH_RMASK31_0`)	W	Writing a 0x1 to any bit location masks off (disables) that interrupt. At IP configuration time, the appropriate number of these registers are instantiated to support the number of External Interrupt Sources.
0x0304	Global Interrupt Reset Mask Register (`GIC_SH_RMASK63_32`)	W	
0x0308	Global Interrupt Reset Mask Register (`GIC_SH_RMASK95_64`)	W	
0x030c	Global Interrupt Reset Mask Register (`GIC_SH_RMASK127_96`)	W	
0x0310	Global Interrupt Reset Mask Register (`GIC_SH_RMASK159_128`)	W	
0x0314	Global Interrupt Reset Mask Register (`GIC_SH_RMASK191_160`)	W	
0x0318	Global Interrupt Reset Mask Register (`GIC_SH_RMASK223_192`)	W	
0x031c	Global Interrupt Reset Mask Register (`GIC_SH_RMASK255_224`)	W	
0x0380	Global Interrupt Set Mask Register (`GIC_SH_SMASK31_00`)	W	Writing a 0x1 to any bit location sets the mask (enables) for that interrupt. At IP configuration time, the appropriate number of these registers are instantiated to support the number of External Interrupt Sources.
0x0384	Global Interrupt Set Mask Register (`GIC_SH_SMASK63_32`)	W	
0x0388	Global Interrupt Set Mask Register (`GIC_SH_SMASK95_64`)	W	
0x038c	Global Interrupt Set Mask Register (`GIC_SH_SMASK127_96`)	W	
0x0390	Global Interrupt Set Mask Register (`GIC_SH_SMASK159_128`)	W	
0x0394	Global Interrupt Set Mask Register (`GIC_SH_SMASK191_160`)	W	
0x0398	Global Interrupt Set Mask Register (`GIC_SH_SMASK223_192`)	W	
0x039c	Global Interrupt Set Mask Register (`GIC_SH_SMASK255_224`)	W	

Table 9.17 Shared Section Register Map *(continued)*

Register Offset	Name	Type	Description
0x0400	Global Interrupt Mask Register (`GIC_SH_MASK31_00`)	R	Shows the enabled global interrupts. If bit N is set, global interrupt N is enabled. At IP configuration time, the appropriate number of these registers are instantiated to support the number of External Interrupt Sources.
0x0404	Global Interrupt Mask Register (`GIC_SH_MASK63_32`)	R	
0x0408	Global Interrupt Mask Register (`GIC_SH_MASK95_64`)	R	
0x040c	Global Interrupt Mask Register (`GIC_SH_MASK127_96`)	R	
0x0410	Global Interrupt Mask Register (`GIC_SH_MASK159_128`)	R	
0x0414	Global Interrupt Mask Register (`GIC_SH_MASK191_160`)	R	
0x0418	Global Interrupt Mask Register (`GIC_SH_MASK223_192`)	R	
0x041c	Global Interrupt Mask Register (`GIC_SH_MASK255_224`)	R	
0x0480	Global Interrupt Pending Register (`GIC_SH_PEND31_00`)	R	Shows the pending global interrupts before masking. If bit N is set, the global interrupt N is pending. At IP configuration time, the appropriate number of these registers are instantiated to support the number of External Interrupt Sources.
0x0484	Global Interrupt Pending Register (`GIC_SH_PEND63_32`)	R	
0x0488	Global Interrupt Pending Register (`GIC_SH_PEND95_64`)	R	
0x048c	Global Interrupt Pending Register (`GIC_SH_PEND127_96`)	R	
0x0490	Global Interrupt Pending Register (`GIC_SH_PEND159_128`)	R	
0x0494	Global Interrupt Pending Register (`GIC_SH_PEND191_160`)	R	
0x0498	Global Interrupt Pending Register (`GIC_SH_PEND223_192`)	R	
0x049c	Global Interrupt Pending Register (`GIC_SH_PEND255_224`)	R	
0x0500	Global Interrupt Map Src0 to Pin Register (`GIC_SH_MAP0_PIN`)	R/W	Maps this interrupt source to a particular pin - within *Int[5:0]* or *NMI*. At IP configuration time, the appropriate number of these registers are instantiated to support the number of External Interrupt Sources.
0x0504	Global Interrupt Map Src1 to Pin Register (`GIC_SH_MAP1_PIN`)	R/W	
0x0508	Global Interrupt Map Src2 to Pin Register (`GIC_SH_MAP2_PIN`)	R/W	
...	...	R/W	
0x08fc	Global Interrupt Map Src255 to Pin Register (`GIC_SH_MAP255_PIN`)	R/W	

Table 9.17 Shared Section Register Map *(continued)*

Register Offset	Name	Type	Description
0x2000	Global Interrupt Map Src0 to Core Register (`GIC_SH_MAP0_CORE31_0`)	R/W	Assigns this interrupt source to a particular core. At IP configuration time, the appropriate number of these registers are instantiated to support the number of External Interrupt Sources and the number of cores.
0x2020	Global Interrupt Map Src1 to Core Register (`GIC_SH_MAP1_CORE31_0`)	R/W	
0x2040	Global Interrupt Map Src2 to Core Register (`GIC_SH_MAP2_CORE31_0`)	R/W	
.....	....	R/W	
0x3fe0	Global Interrupt Map Src255 to Core Register (`GIC_SH_MAP255_CORE31_0`)	R/W	
0x6000	DINT Send to Group Register (GIC_VB_DINT_SEND)	R/W	Sends the DebugInterrupt to the specified core.
All other offsets	Reserved for future extensions		Reserved for future extensions.

9.5.3 Shared Section Register Descriptions

The physical address for the Shared Section registers is calculated as follows:

```
GIC_BaseAddress + SharedSection_BaseAddress + RegisterOffset
```

9.5.3.1 Global Config Register

Figure 9.9 Global Config Register Format

31 29	28	27 24	23 16	15 9	8 0
R	COUNT STOP	COUNTBITS	NUMINTERRUPTS	0	PVPES

Table 9.18 GIC Config Register (GIC_SH_CONFIG — Offset 0x0000)

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
R	31:29	Reserved. Read as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0
COUNTSTOP	28	Setting this bit will stop *GIC_CounterHi* and *GIC_CounterLo*. Used to freeze the shared counters when cores go into power-down or debug modes.	R/W	0
COUNTBITS	27:24	Number of Implemented Bits in *GIC_CounterHi*. Total Number of Counter Bits = 32 + COUNTBITS*4, E.g.: 0x0 = 32bits, *GIC_CounterHi* not implemented 0x1 = 36bits, *GIC_CounterHi* width = 4 bits 0x2 = 40bits, *GIC_CounterHi* width = 8 bits ... 0x7 = 60bits, *GIC_CounterHi* width = 28 bits 0x8 = 64bits, *GIC_CounterHi* width = 32 bits 9-15 Reserved	R	0x8

Table 9.18 GIC Config Register (GIC_SH_CONFIG — Offset 0x0000) *(continued)*

<table>
<tr><th colspan="2">Register Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/
Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td><i>NUMINTERRUPTS</i></td><td>23:16</td><td>Number of External Interrupt Sources
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0x0</td><td>8</td></tr>
<tr><td>0x1</td><td>16</td></tr>
<tr><td>0x2</td><td>24</td></tr>
<tr><td>0x3</td><td>32</td></tr>
<tr><td>0x4</td><td>40</td></tr>
<tr><td>...</td><td></td></tr>
<tr><td>0x3E</td><td>248</td></tr>
<tr><td>0x3F</td><td>256</td></tr>
<tr><td>All others</td><td>Reserved</td></tr>
</table>
Value is fixed by customer at IP configuration time.</td><td>R</td><td>IP Configuration Value</td></tr>
<tr><td>R</td><td>15:9</td><td>Reserved. Read as 0x0. Writes ignored. Must be written with a value of 0x0.</td><td>R</td><td>0</td></tr>
<tr><td><i>PVPES</i></td><td>8:0</td><td>Total number of VPEs in the system.Note that in the proAptiv core, there is one VPE per core.
0: 1 VPE</td><td>R</td><td>IP Configuration Value</td></tr>
</table>

9.5.3.2 GIC CounterLo

Figure 9.10 GIC CounterLo Register Format

31 … 0
GIC_SH_CounterLo

Table 9.19 GIC CounterLo (GIC_SH_CounterLo — Offset 0x0010)

Register Fields		Description	Read/ Write	Reset State
Name	**Bits**			
GIC_SH_CounterLo	31:0	Lower Half of an up-counter. When the counter reaches its maximum value, the counter rolls over to a value of 0x0. The counter is running at an implementation-specific frequency which is fixed, that is, not changing dynamically due to power management. It is recommended that this frequency be as close as possible to the highest clock frequency of the CPU subsystem. This counter is disabled by writing the *COUNTSTOP* bit in the *GIC_SH_CONFIG* register. This counter should only be written when *GIC_SH_CONFIG*$_{COUNTSTOP}$ = 1; otherwise, the registers results after the write are unpredictable.	R/W	0

9.5.3.3 GIC CounterHi

Figure 9.11 GIC CounterHi Register Format

31 … 0
GIC_SH_CounterHi

Table 9.20 GIC CounterHi (GIC_SH_CounterHi — Offset 0x0014)

Register Fields		Description	Read/ Write	Reset State
Name	**Bits**			
GIC_SH_CounterHi	31:0	Upper Half of an up-counter. When the counter reaches its maximum value, the counter rolls over to a value of 0x0. The counter is running at an implementation-specific frequency which is fixed, that is, not changing dynamically due to power management. It is recommended that this frequency be as close as possible to the highest clock frequency of the CPU subsystem. This counter is disabled by writing the *COUNTSTOP* bit in the *GIC_SH_CONFIG* register. This counter should only be written when *GIC_SH_CONFIG*$_{COUNTSTOP}$ = 1; otherwise, the register results after the write are unpredictable. Unimplemented bits ignore writes and return 0 when read.	R/W	0

9.5.3.4 GIC Revision Register

Figure 9.12 GIC Revision Register Format

31 16	15 8	7 0
0	MAJOR_REV	MINOR_REV

Table 9.21 GIC Revision Register (GIC_RevisionID — Offset 0x0020)

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
0	31:16	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0x0
MAJOR_REV	15:8	This field reflects the major revision of the GIC block. A major revision might reflect the changes from one product generation to another.	R	Preset
MINOR_REV	7:0	This field reflects the minor revision of the GIC block. A minor revision might reflect the changes from one release to another.	R	Preset

9.5.3.5 Global Interrupt Polarity Registers

There are eight Global Interrupt Polarity registers to cover all 256 possible system interrupts. These registers work in conjunction with the eight *Global Interrupt Trigger Type (GIC_SH_TRIGn)* and *Global Interrupt Dual Edge (GIC_SH_DUALn)* registers to select the polarity, active high/low trigger, and single/dual edge for each of the 256 interrupts. Refer to Section 9.4.2, "Configuring Interrupt Sources" for more informatioin.

They are located at the following eight offsets.

Table 9.22 Global Interrupt Polarity Register Mapping

Offset	Acronym	Register Name
0x0100	GIC_SH_POL31_0	Polarity selection for interrupt pins 31:0
0x0104	GIC_SH_POL63_32	Polarity selection for interrupt pins 63:32
0x0108	GIC_SH_POL95_64	Polarity selection for interrupt pins 95:64
0x010C	GIC_SH_POL127_96	Polarity selection for interrupt pins 127:96
0x0110	GIC_SH_POL159_128	Polarity selection for interrupt pins 159:128
0x0114	GIC_SH_POL191_160	Polarity selection for interrupt pins 191:160
0x0118	GIC_SH_POL223_192	Polarity selection for interrupt pins 223:191
0x011C	GIC_SH_POL255_224	Polarity selection for interrupt pins 255:192

In the register below, the x_y nomenclature indicates the bit range covered by each register shown above. For example, GIC_SH_POL63_32 indicates that this register handles the polarity for interrupts 63:32.

Figure 9.13 GIC Interrupt Polarity Register Format

31	0
GIC_SH_POLx_y	

Table 9.23 Global Interrupt Polarity Registers (GIC_SH_POLx_y — See Table 9.23 for Mapping)

<table>
<tr><th colspan="2">Register Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/
Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td><i>GIC_SH_POLx_y</i></td><td>31:0</td><td>Each bit in this register represents an interrupt source.
The state of the bit indicates the polarity of the interrupt.
If the interrupt type (as denoted by <i>Global Interrupt Trigger Type</i> and <i>Global Interrupt Dual Edge</i> registers) is Level:.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Active Low</td></tr>
<tr><td>1</td><td>Active High</td></tr>
</table>
If the interrupt type is Single-edge:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Falling Edge denotes interrupt source has toggled.</td></tr>
<tr><td>1</td><td>Rising Edge denotes interrupt source has toggled.</td></tr>
</table>
If the interrupt type is Dual-edge, this register is not used</td><td>R/W</td><td>0</td></tr>
</table>

9.5.3.6 Global Interrupt Trigger Type Registers

There are eight Global Interrupt Trigger Type registers to cover all 256 possible system interrupts. These registers work in conjunction with the eight *Global Interrupt Polarity (GIC_SH_POLn)* and *Global Interrupt Dual Edge (GIC_SH_DUALn)* registers to select the polarity, active high/low trigger, and single/dual edge for each of the 256 interrupts. Refer to Section 9.4.2, "Configuring Interrupt Sources" for more informatioin.

They are located at the following eight offsets.

Table 9.24 Global Interrupt Trigger Type Register Mapping

Offset	Acronym	Register Name
0x0180	GIC_SH_TRIG31_0	Interrupt trigger selection for interrupt pins 31:0
0x0184	GIC_SH_TRIG63_32	Interrupt trigger selection for interrupt pins 63:32
0x0188	GIC_SH_TRIG95_64	Interrupt trigger selection for interrupt pins 95:64
0x018C	GIC_SH_TRIG127_96	Interrupt trigger selection for interrupt pins 127:96
0x0190	GIC_SH_TRIG159_128	Interrupt trigger selection for interrupt pins 159:128
0x0194	GIC_SH_TRIG191_160	Interrupt trigger selection for interrupt pins 191:160
0x0198	GIC_SH_TRIG223_192	Interrupt trigger selection for interrupt pins 223:191
0x019C	GIC_SH_TRIG255_224	Interrupt trigger selection for interrupt pins 255:192

In the register below, the x_y nomenclature indicates the bit range covered by each register shown above. For example, GIC_SH_TRIG63_32 indicates that this register handles the trigger level for interrupts 63:32.

Figure 9.14 GIC Interrupt Trigger Type Register Format

31
0
GIC_SH_TRIGx_y

Table 9.25 Global Interrupt Trigger Type Registers (GIC_SH_TRIGx_y — See Table 9.25 for Mapping)

<table>
<tr><th colspan="2">Register Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/
Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td><i>GIC_SH_TRIGx_y</i></td><td>31:0</td><td>Each bit in this register represents an interrupt source. The state of the bit indicates the nature of the interrupt signaling.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Level</td></tr>
<tr><td>1</td><td>Edge
Single edge or dual-edge signaling denoted by <i>Global Interrupt Dual Edge Register</i>.</td></tr>
</table></td><td>R/W</td><td>0</td></tr>
</table>

9.5.3.7 Global Interrupt Dual Edge Registers

There are eight Global Interrupt Dual Edge registers to cover all 256 possible system interrupts. These registers work in conjunction with the eight *Global Interrupt Polarity (GIC_SH_POLn)* and *Global Interrupt Trigger Type (GIC_SH_TRIGn)* registers to select the polarity, active high/low trigger, and single/dual edge for each of the 256 interrupts. Refer to Section 9.4.2, "Configuring Interrupt Sources" for more informatioin.

They are located at the following eight offsets.

Table 9.26 Global Interrupt Dual Edge Register Mapping

Offset	Acronym	Register Name
0x0200	GIC_SH_DUAL31_0	Interrupt single/dual edge selection for interrupt pins 31:0
0x0204	GIC_SH_DUAL63_32	Interrupt single/dual edge selection for interrupt pins 63:32
0x0208	GIC_SH_DUAL95_64	Interrupt single/dual edge selection for interrupt pins 95:64
0x020C	GIC_SH_DUAL127_96	Interrupt single/dual edge selection for interrupt pins 127:96
0x0210	GIC_SH_DUAL159_128	Interrupt single/dual edge selection for interrupt pins 159:128
0x0214	GIC_SH_DUAL191_160	Interrupt single/dual edge selection for interrupt pins 191:160
0x0218	GIC_SH_DUAL223_192	Interrupt single/dual edge selection for interrupt pins 223:191
0x021C	GIC_SH_DUAL255_224	Interrupt single/dual edge selection for interrupt pins 255:192

In the register below, the x_y nomenclature indicates the bit range covered by each register shown above. For example, GIC_SH_DUAL63_32 indicates that this register handles the edge triggering for interrupts 63:32.

Figure 9.15 GIC Interrupt Dual Edge Register Format

31 … 0
GIC_SH_DUALx_y

Table 9.27 Global Dual Edge Registers (GIC_SH_DUALx_y — See Table 9.27 for Mapping)

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
GIC_SH_DUALx_y	31:0	Each bit in this register represents an interrupt source. This register is only meaningful is the equivalent bit in the *Global Interrupt Trigger Type* register is set to 0x1 = Edge signaling. Indicates single or dual-edged signaling. **Encoding** | **Meaning** 0 | Single edge 1 | Dual Edge	R/W	0

9.5.3.8 Global Interrupt Write Edge Register

This register is used to support interrupt messages. A write to this register will atomically set or clear one bit in the *Edge Detect Register*. Setting a bit in this register will be treated equivalently to having the edge detection logic see an active edge. This bypasses the edge detection logic and thus it does not matter whether the corresponding interrupt is configured to be rising, falling, or dual edge sensitive. However, the behavior is undefined unless the equivalent bit in the *Global Interrupt Trigger Type* register is set to 0x1 indicating edge signaling.

Figure 9.16 GIC Interrupt Write Edge Register Format

31	30 … 0
RW	INTERRUPT

Table 9.28 Global Interrupt Write Edge Registers (GIC_SH_WEDGE Offset 0x0280)

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
RW	31	Controls whether this write is setting or clearing a bit in the *Edge Detect Register*. If this bit is set, the selected bit in the register is set. If this bit is cleared, the selected bit in the register is cleared.	W	Undefined
Interrupt	30:0	This field is the encoded value of the interrupt that is being cleared or set. For example, a value of 0xB means interrupt 11 (decimal).	W	Undefined

9.5.3.9 Global Interrupt Reset Mask Registers

There are eight Global Interrupt Reset Mask registers to cover all 256 possible system interrupts. These registers work in conjunction with the eight *Global Interrupt Set Mask (GIC_SH_SMASKn)* registers to enable and disable individual interrupts. Refer to Section 9.4.2, "Configuring Interrupt Sources" for more informatioin.

These registers are located at the following eight offsets.

Table 9.29 Global Interrupt Reset Mask Register Mapping

Offset	Acronym	Register Name
0x0300	GIC_SH_RMASK31_0	Interrupt reset mask for interrupt pins 31:0
0x0304	GIC_SH_RMASK63_32	Interrupt reset mask for interrupt pins 63:32
0x0308	GIC_SH_RMASK95_64	Interrupt reset mask for interrupt pins 95:64
0x030C	GIC_SH_RMASK127_96	Interrupt reset mask for interrupt pins 127:96
0x0310	GIC_SH_RMASK159_128	Interrupt reset mask for interrupt pins 159:128
0x0314	GIC_SH_RMASK191_160	Interrupt reset mask for interrupt pins 191:160
0x0318	GIC_SH_RMASK223_192	Interrupt reset mask for interrupt pins 223:191
0x031C	GIC_SH_RMASK255_224	Interrupt reset mask for interrupt pins 255:192

In the register below, the x_y nomenclature indicates the bit range covered by each register shown above. For example, GIC_SH_RMASK63_32 indicates that this register handles the reset mask for interrupts 63:32.

Figure 9.17 GIC Interrupt Reset Mask Register Format

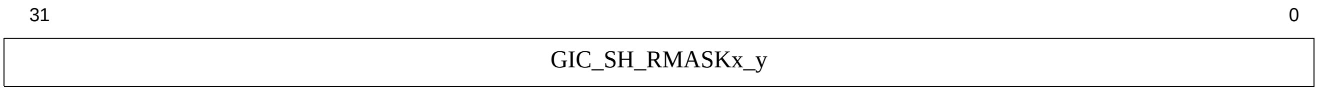


Table 9.30 Global Interrupt Reset Mask Registers (GIC_SH_RMASKx_y — See Table 9.30 for Mapping)

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
GIC_SH_RMASKx_y	31:0	Each bit in this register represents an interrupt source. Writing this register with a 0x1 in any bit position(s) will cause only the corresponding bit/interrupt(s) in the *Global Interrupt Mask Register* to be reset (value->0). This is used by software to temporarily disable interrupts.	W	Undefined

9.5.3.10 Global Interrupt Set Mask Registers

There are eight Global Interrupt Set Mask registers to cover all 256 possible system interrupts. These registers work in conjunction with the eight *Global Interrupt Reset Mask (GIC_SH_RMASKn)* registers to enable and disable individual interrupts. Refer to Section 9.4.2, "Configuring Interrupt Sources" for more informatioin.

These registers are located at the following eight offsets.

Table 9.31 Global Interrupt Set Mask Register Mapping

Offset	Acronym	Register Name
0x0380	GIC_SH_SMASK31_0	Interrupt set mask for interrupt pins 31:0
0x0384	GIC_SH_SMASK63_32	Interrupt set mask for interrupt pins 63:32
0x0388	GIC_SH_SMASK95_64	Interrupt set mask for interrupt pins 95:64
0x038C	GIC_SH_SMASK127_96	Interrupt set mask for interrupt pins 127:96

Table 9.31 Global Interrupt Set Mask Register Mapping *(continued)*

Offset	Acronym	Register Name
0x0390	GIC_SH_SMASK159_128	Interrupt set mask for interrupt pins 159:128
0x0394	GIC_SH_SMASK191_160	Interrupt set mask for interrupt pins 191:160
0x0398	GIC_SH_SMASK223_192	Interrupt set mask for interrupt pins 223:191
0x039C	GIC_SH_SMASK255_224	Interrupt set mask for interrupt pins 255:192

In the register below, the x_y nomenclature indicates the bit range covered by each register shown above. For example, GIC_SH_SMASK63_32 indicates that this register handles the set mask for interrupts 63:32.

Figure 9.18 GIC Interrupt Set Mask Register Format

31
0
GIC_SH_SMASKx_y

Table 9.32 Global Set Mask Registers (GIC_SH_SMASKx_y — See Table 9.32 for Mapping)

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
GIC_SH_SMASKx_y	31:0	Each bit in this register represents an interrupt source. Writing this register with a 0x1 in any bit position(s) will cause only the corresponding bit/interrupt(s) in the *Global Interrupt Mask Register* to be set(value->0x1). This is used by software to enable interrupts.	W	Undefined

9.5.3.11 Global Interrupt Mask Registers

There are eight Global Interrupt Reset Mask registers to cover all 256 possible system interrupts. These read-only registers are used to indicate when an external interrupt occurs. An individual interrupt bit is set when an interrupt occurs and the corresponding Global Interrupt Set Mask bit is set, thereby enabling the interrupt. Refer to Section 9.5.3.10, "Global Interrupt Set Mask Registers" for more information.

These registers work in conjunction with the eight *Global Interrupt Set Mask (GIC_SH_SMASKn)* and *Global Interrupt Reset Mask (GIC_SH_RMASKn)* registers to manage and process interrupts. Refer to Section 9.4.2, "Configuring Interrupt Sources" for more informatioin.

These registers are located at the following eight offsets.

Table 9.33 Global Interrupt Mask Register Mapping

Offset	Acronym	Register Name
0x0400	GIC_SH_MASK31_0	Interrupt status for interrupt pins 31:0
0x0404	GIC_SH_MASK63_32	Interrupt status for interrupt pins 63:32
0x0408	GIC_SH_MASK95_64	Interrupt status for interrupt pins 95:64
0x040C	GIC_SH_MASK127_96	Interrupt status for interrupt pins 127:96
0x0410	GIC_SH_MASK159_128	Interrupt status for interrupt pins 159:128
0x0414	GIC_SH_MASK191_160	Interrupt status for interrupt pins 191:160

Table 9.33 Global Interrupt Mask Register Mapping *(continued)*

Offset	Acronym	Register Name
0x0418	GIC_SH_MASK223_192	Interrupt status for interrupt pins 223:191
0x041C	GIC_SH_MASK255_224	Interrupt status for interrupt pins 255:192

In the register below, the x_y nomenclature indicates the bit range covered by each register shown above. For example, GIC_SH_MASK63_32 indicates that this register handles the masking for interrupts 63:32.

Figure 9.19 GIC Interrupt Mask Register Format

31	0
GIC_SH_MASKx_y	

Table 9.34 Global Interrupt Mask Registers (GIC_SH_MASKx_y — See Table 9.34 for Mapping)

Register Fields				
Name	Bits	Description	Read/ Write	Reset State
GIC_SH_MASKx_y	31:0	Each bit in this register represents an interrupt source. Reports which of the external interrupt sources are enabled. Used by software to determine which interrupt sources are currently enabled.	R	0x00000000

9.5.3.12 Global Interrupt Pending Registers

There are eight Global Interrupt Pending registers to cover the pending status of all 256 possible system interrupts. These read-only registers are set by hardware when an external interrupt is pending.

These registers work in conjunction with the eight *Global Interrupt Set Mask (GIC_SH_SMASKn)*, *Global Interrupt Reset Mask (GIC_SH_RMASKn)*, and *Global Interrupt Mask (GIC_SH_MASKn)* registers to manage and process interrupts. Refer to Section 9.4.2, "Configuring Interrupt Sources" for more informatioin.

These registers are located at the following eight offsets.

Table 9.35 Global Interrupt Pending Register Mapping

Offset	Acronym	Register Name
0x0480	GIC_SH_PEND31_0	Interrupt pending status for interrupt pins 31:0
0x0484	GIC_SH_PEND63_32	Interrupt pending status for interrupt pins 63:32
0x0488	GIC_SH_PEND95_64	Interrupt pending status for interrupt pins 95:64
0x048C	GIC_SH_PEND127_96	Interrupt pending status for interrupt pins 127:96
0x0490	GIC_SH_PEND159_128	Interrupt pending status for interrupt pins 159:128
0x0494	GIC_SH_PEND191_160	Interrupt pending status for interrupt pins 191:160
0x0498	GIC_SH_PEND223_192	Interrupt pending status for interrupt pins 223:191
0x049C	GIC_SH_PEND255_224	Interrupt pending status for interrupt pins 255:192

In the register below, the x_y nomenclature indicates the bit range covered by each register shown above. For example, GIC_SH_PEND63_32 indicates that this register handles the interrupt pending status for interrupts 63:32.

Figure 9.20 GIC Interrupt Pending Register Format

31 0
GIC_SH_PENDx_y

Table 9.36 Global Interrupt Pending Registers (GIC_SH_PENDx_y — See Table 9.36 for Mapping)

Register Fields				
Name	**Bits**	**Description**	**Read/ Write**	**Reset State**
GIC_SH_PENDx_y	31:0	There are eight Interrupt Pending register that are used to indicate the pending status of all 256 possible interrupts in the system Each bit indicates which of the external interrupt sources are asserted/pending before masking. Used by software to find the external source that caused the CPU interrupt.	R	Undefined

9.5.3.13 Global Interrupt Map to Pin Registers

There are up to 256 Global Interrupt Map-to-Pin registers in the GIC to cover the mapping of all 256 possible system interrupts. This corresponds to one register per external interrupt signal. The number of registers instantiated at build time depends on the number of external system interrupts. These are write-only registers. Software is not expected to change these registers frequently. Software is expected to keep a shadow copy of these registers in memory so that Read-Modify-Write hazards are avoided.

Each interrupt pin can be mapped to one of three signal types: *SI_Int[5:0]* or *SI_NMI*. Bits 31:30 of this register are used to indicate to which signal type the interrupt will be mapped. Only one of these bits can be set at any given time. Bits 5:0 indicate the actual mapping for each external interrupt pin. For example, if bit 31 of this register is set, the external interrupt is routed to the *SI_Int[5:0]* pins of the appropriate core .

For the register offset addresses corresponding to each register, refer Table 9.7, "Mapping of External Interrupts"

Figure 9.21 GIC Interrupt Map to Pin Register Format

31	30	29 6	5 0
MAP_TO_PIN	MAP_TO_NMI	R	MAP

Table 9.37 Global Interrupt Map to Pin Registers (GIC_SH_MAPx_y)

Register Fields				
Name	**Bits**	**Description**	**Read/ Write**	**Reset State**
MAP_TO_PIN	31	If this bit is set, this interrupt source is mapped to a core interrupt pin (specified by the *MAP* field below). Only one of the *MAP_TO_PIN* or *MAP_TO_NMI* bits can be set at any one time.	RW	0x1
MAP_TO_NMI	30	If this bit is set, this interrupt source is mapped to NMI. Only one of the *MAP_TO_PIN* or *MAP_TO_NMI,* or *MAP_TO_YQ* bits can be set at any one time.	RW	0
Reserved	29:6	Read as 0x0. Writes ignored. Must be written with a value of 0x0.	-	0

Table 9.37 Global Interrupt Map to Pin Registers (GIC_SH_MAPx_y) *(continued)*

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
MAP	5:0	When the *MAP_TO_PIN* bit is set, this field contains the encoded value of the core interrupts signals *Int[62:0]*. In EIC mode, this represents one less than the EIC interrupt level (e.g. a value of 0x20 represents interrupt level 21). For non-EIC mode, the value represents the CPU interrupt to be asserted (e.g. a value of 0x03 represents interrupt 3), and only values of 0 to 5 are legal.	RW	0

9.5.3.14 Global Interrupt Map to Core Registers

There are up to 512 Global Interrupt Map-to-Core registers in the GIC to cover the mapping of all 256 possible system interrupts. This corresponds to two registers per external interrupt signal. However, the high-order register is not used in the proAptiv Multiprocessing System core as described in Section 9.5.3.14, "Global Interrupt Map to Core Registers".

The number of registers instantiated at build time depends on the number of external system interrupts. These are write-only registers. Software is not expected to change these registers frequently. Software is expected to keep a shadow copy of these registers in memory so that Read-Modify-Write hazards are avoided.

For the register offset addresses corresponding to each register, refer Table 9.7, "Mapping of External Interrupts"

Figure 9.22 GIC Interrupt Map to Core Register Format

31	0
GIC_SH_MAPi_COREn	

Table 9.38 Global Interrupt Map to Core Registers (GIC_SH_MAP_COREn — See Table 9.7 for Mapping)

Register Fields				
Name	Bits	Description	Read/ Write	Reset State
GIC_SH_MAPi_COREn	31:0	Setting any bit in this register causes the interrupt source to be routed to the corresponding VPE. For all GIC_SH_MAPi_CORE registers, only one bit may be set at a time. That is, an interrupt source will be routed to one and only one core.	W	0

9.5.3.15 DINT Send to Group Register

This register allows software to assert the EJ_DINT_GROUP signal directly. Refer to Section 9.4.10 "Debug Interrupt Generation" for more information.

Figure 9.23 DINT Send to Group Register Format

31 … 1	0
R	SEND_DINT

Table 9.39 DINT Send to Group Register (GIC_VB_DINT_SEND Offset 0x6000)

Register Fields		Description	Read/ Write	Reset State
Name	**Bits**			
R	[31:1]	Read as Zero. Writes ignored.	-	0x0
SEND_DINT	[0]	If this register field is written with a value of 0x1, the *EJ_DINT_GROUP* signal is asserted in a one-shot manner.	W	0x0

See Chapter 16, "Multi-CPU Debug" on page 611 for more information about how this register is used.

9.6 GIC Core-Local and Core-Other Register Set

9.6.1 Core-Local and Core-Other Register Maps

The Core-Local and Core-Other interrupt register maps are described in Table 9.41 below. For the base addresses of these blocks, see Table 9.3. Each core in the proAptiv sub-system contains a set of these registers.

The physical address for the registers within the Core-Local section are calculated as follows:

```
Core-Local_Register_Physical_Address = GIC_BaseAddress + Core-Local_BaseOffset +
Register Offset
```

Similarly, for the Core-Other section:

```
Core-Other_Register_Physical_Address = GIC_BaseAddress + Core-Other_BaseOffset +
Register Offset
```

All registers are 32 bits wide and should only be accessed using 32-bit uncached load/stores. Reads from unpopulated registers in the GCMP address space will return 0x0, and writes to those locations will be silently dropped without generating any exceptions.

Table 9.40 Core-Local and Core-Other Register Maps

Register Offset	Name	Type	Description
0x0000	Local Interrupt Control Register (GIC_COREi_CTL)	R/W	Enable EIC Mode.
0x0004	Local Interrupt Pending Register (GIC_COREi_PEND)	R	Status of the local interrupts before masking.
0x0008	Local Mask Register (GIC_COREi_MASK)	R	Mask bits, if set will enable the corresponding interrupts in the interrupt vector.
0x000c	Local Reset Mask Register (GIC_COREi_RMASK)	W	Setting a bit in this register causes the corresponding bits in the *GIC_COREi_MASK* register to be cleared atomically with respect to other bits.
0x0010	Local Set Mask Register (GIC_COREi_SMASK)	W	Setting a bit in this register causes the corresponding bits in the *GIC_COREi_MASK* register to be set atomically with respect to other bits.
0x0040	Local WatchDog Map-to-Pin Register (GIC_COREi_WD_MAP)	R/W	This register is used to route the local WatchDog interrupt to the desired core pin.
0x0044	Local GIC Counter/Compare Map-to-Pin Register (GIC_COREi_COMPARE_MAP)	R/W	This register is used to route the local GIC Compare/Count Interrupt to the desired core pin. This is an optional register instantiated at IP configuration time.
0x0048	Local CPU Timer Map-to-Pin Register (GIC_COREi_TIMER_MAP)	R/W	This register is used to route the local CPU Timer interrupt to the desired core pin.

Table 9.40 Core-Local and Core-Other Register Maps *(continued)*

Register Offset	Name	Type	Description
0x004c	Local CPU Fast Debug Channel Map-to-Pin Register (GIC_COREi_FDC_MAP)	R/W	This register is used to route the local CPU Fast Debug Channel interrupt to the desired core pin. This is an optional register instantiated at IP configuration time.
0x0050	Local Perf Counter Map-to-Pin Register (GIC_COREi_PERFCTR_MAP)	R/W	This register is used to route the local Performance Counter interrupt to the desired core pin. This is an optional register instantiated at IP configuration time.
0x0054	Local SWInt0 Map-to-Pin Register (GIC_COREi_SWInt0_MAP)	R/W	This register is used to route the local SWInt0 interrupt to the desired core pin. This is an optional register instantiated at IP configuration time.
0x0058	Local SWInt1 Map-to-Pin Register (GIC_COREi_SWInt1_MAP)	R/W	This register is used to route the local SWInt1 interrupt to the desired core pin. This is an optional register instantiated at IP configuration time.
0x0080	Core-Other Addressing Register (GIC_COREi_OTHER_ADDR)	R/W	Sets the *VPENum* of the register that will be accessed through the Core-Other address space.
0x0088	Core-Local Identification Register (GIC_COREi_IDENT)	R	Indicates the Core number of the local Core.
0x0090	Programmable/Watchdog Timer0 Config Register (GIC_COREi_WD_CONFIG0)	R/W	Local Programmable or Watchdog Timer0 related registers. See register description for more details.
0x0094	Programmable/Watchdog Timer0 Count Register (GIC_COREi_WD_COUNT0)	R	
0x0098	Programmable/Watchdog Timer0 Initial Count Register (GIC_COREi_WD_INITIAL0)	R/W	
0x00A0	CompareLo Register (GIC_COREi_CompareLo)	R/W	Compare Register. See register description for more details.
0x00A4	CompareHi Register (GIC_COREi_CompareHi)	R	
0x0100	EIC Shadow Set for Interrupt Src0 (GIC_COREi_EICSS0)	R/W	EIC Shadow Set for Interrupt Source0.
0x0104	EIC Shadow Set for Interrupt Src1 (GIC_COREi_EICSS1)	R/W	EIC Shadow Set for Interrupt Source1.
0x0108 - 0x01F8	EIC Shadow Set for Interrupt Src2 through Interrupt Src62 (GIC_VPEi_EICSS2 - GIC_COREi_EICSS62)	R/W	EIC Shadow Set for Interrupt Source2 through Source62.
0x01FC	EIC Shadow Set for Interrupt Src63 (GIC_COREi_EICSS63)	R/W	EIC Shadow Set for Interrupt Source63.

Table 9.40 Core-Local and Core-Other Register Maps *(continued)*

Register Offset	Name	Type	Description
0x3000	Core-Local DINT Group Participate Register (`GIC_VL_DINT_PART` `GIC_VO_DINT_PART`)	R/W	Controls whether this Core pays attention to the *DebugInt_GroupRequest* register.
0x3080	Core-Local DebugBreak Group Register (`GIC_VL_BRK_GROUP` `GIC_VO_BRK_GROUP`)	R/W	Allows multiple Core to simultaneously enter Debug Mode.
All Other Offsets	RESERVED		Reserved for Future Extensions.

9.6.2 Core-Local and Core-Other Section Register Description

The following subsections describes the registers of the Core-Local and Core-Other sections.

9.6.2.1 Local Interrupt Control Register

Figure 9.24 Local Interrupt Control Register Format

31 5	4	3	2	1	0
R	FDC_ ROUTABLE	SWINT_ ROUTABLE	PERFCOUNT_ ROUTABLE	TIMER_ ROUTABLE	EIC_MODE

Table 9.41 Local Interrupt Control Register (GIC_COREi_CTL — Offset 0x0000)

Register Fields				
Name	Bits	Description	Read/ Write	Reset State
RESERVED	31:5	Read as 0x0. Writes ignored. Must be written with a value of 0x0.		0
FDC_ROUTABLE	4	If this bit is set, the CPU Fast Debug Channel Interrupt is routable within the GIC. If this bit is clear, the CPU Fast Debug Channel Interrupt is hardwired to one of the *SI_Int* pins as described by the CPU's COP0 *IntCtlIPFDCI* register field.	R	IP Configuration Value
SWINT_ROUTABLE	3	If this bit is set, the CPU SW Interrupts are routable within the GIC. If this bit is clear, then the CPU SW Interrupts are routed back to the CPU directly.	R	IP Configuration Value
PERFCOUNT_ROUTABLE	2	If this bit is set, the CPU Performance Counter Interrupt is routable within the GIC. If this bit is clear, the CPU Performance Counter Interrupt is hardwired to one of *SI_Int* pins as described by the CPU's COP0 $IntCtlI_{PPCI}$ register field.	R	IP Configuration Value
TIMER_ROUTABLE	1	If this bit is set, the CPU Timer Interrupt is route-able within the GIC. If this bit is clear, the CPU Timer Interrupt is hardwired to one of the *SI_Int* pins, as described by the CPU's COP0 $IntCtlI_{PTI}$ register field.	R	IP Configuration Value
EIC_MODE	0	Writing a 1 to this bit will set this local interrupt controller to EIC (External Interrupt Controller) mode.	R/W	0

9.6.2.2 Local Interrupt Pending Register

Figure 9.25 Local Interrupt Pending Register Format

31 7	6	5	4	3	2	1	0
R	FDCPEND	SWINT1_ PEND	SWINT0_ PEND	PERFCOUNT_ PEND	TIMER_ PEND	COMPARE _PEND	WQ_PEND

Table 9.42 Local Interrupt Pending Register (GIC_COREi_PEND — Offset 0x0004)

Register Fields		Description	Read/ Write	Reset State
Name	**Bits**			
R	31:7	Read as 0x0. Writes ignored. Must be written with a value of 0x0.		0
FDC_PEND	6	Indicates the status of the local Fast Debug Channel interrupt prior to masking	R	Undefined
SWINT1_PEND	5	Indicates the status of the local SW interrupt1 prior to masking.	R	Undefined
SWINT0_PEND	4	Indicates the status of the local SW interrupt0 prior to masking.	R	Undefined
PERFCOUNT_PEND	3	Indicates the status of the local Performance Counter interrupt prior to masking.	R	Undefined
TIMER_PEND	2	Indicates the status of the local CPU Timer interrupt prior to masking.	R	Undefined
COMPARE_PEND	1	Indicates the status of the local GIC Count/Compare interrupt prior to masking.	R	Undefined
WD_PEND	0	Indicates the status of the local WatchDog interrupt prior to masking.	R	Undefined

9.6.2.3 Local Interrupt Mask Register

This is a read-only register. Refer to Section 9.4.2, "Configuring Interrupt Sources" for more information.

Figure 9.26 Local Interrupt Mask Register Format

31 7	6	5	4	3	2	1	0
R	FDC- MASK	SWINT1_ MASK	SWINT0_ MASK	PERFCOUNT_ MASK	TIMER_ MASK	COMPARE_ MASK	WQ_MASK

Table 9.43 Local Interrupt Mask Register (GIC_COREi_MASK — Offset 0x0008)

Register Fields		Description	Read/ Write	Reset State
Name	**Bits**			
RESERVED	31:7	Read as 0x0	R	0x0
FDC_MASK	6	If this bit is set, the local Fast Debug Channel interrupt is enabled	R	0x1
SWINT1_MASK	5	If this bit is set, the local SWInt1 Interrupt is enabled.	R	0x1
SWINT0_MASK	4	If this bit is set, the local SWInt0 Interrupt is enabled.	R	0x1

Table 9.43 Local Interrupt Mask Register (GIC_COREi_MASK — Offset 0x0008)

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
PERFCNT_MASK	3	If this bit is set, the local Performance Counter Interrupt is enabled.	R	0x1
TIMER_MASK	2	If this bit is set, the local CPU Timer Interrupt is enabled.	R	0x1
COMPARE_MASK	1	If this bit is set, the local GIC Count/Compare Interrupt is enabled.	R	0x1
WD_MASK	0	If this bit is set, the local WatchDog Interrupt is enabled.	R	0x1

9.6.2.4 Local Interrupt Reset Mask Register

Figure 9.27 Local Interrupt Reset Mask Register Format

31 7	6	5	4	3	2	1	0
R	FDC RMASK	SWINT1_ RMASK	SWINT0_ RMASK	PERFCOUNT_ RMASK	TIMER_ RMASK	COMPARE_ RMASK	WQ_RMASK

Table 9.44 Local Interrupt Reset Mask Register (GIC_COREi_RMASK — Offset 0x000C)

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
RESERVED	31:7	Writes ignored. Must be written with a value of 0x0.		Undefined
FDC_RMASK	6	Writing a 0x1 to this bit disables the local Fast Debug Channel interrupt	W	Undefined
SWINT1_RMASK	5	Writing a 0x1 to this bit disables the local SWInt1 Interrupt.	W	Undefined
SWINT0_RMASK	4	Writing a 0x1 to this bit disables the local SWInt0 Interrupt.	W	Undefined
PERFCNT_RMASK	3	Writing a 0x1 to this bit disables the local Performance Counter Interrupt.	W	Undefined
TIMER_RMASK	2	Writing a 0x1 to this bit disables the local Timer Interrupt.	W	Undefined
COMPARE_RMASK	1	Writing a 0x1 to this bit disables the local GIC Count/ Compare Interrupt.	W	Undefined
WD_RMASK	0	Writing a 0x1 to this bit disables the local WatchDog Timer Interrupt.	W	Undefined

9.6.2.5 Local Interrupt Set Mask Register

This is a write-only register. For more information, refer to Section 9.4.2, "Configuring Interrupt Sources".

Figure 9.28 Local Interrupt Set Mask Register Format

31 7	6	5	4	3	2	1	0
R	FDC_ SMASK	SWINT1_ SMASK	SWINT0_ SMASK	PERFCOUNT_ SMASK	TIMER_ SMASK	COMPARE_ SMASK	WQ_SMASK

Table 9.45 Local Interrupt Set Mask Register (GIC_COREi_SMASK — Offset 0x0010)

Register Fields				
Name	**Bits**	**Description**	**Read/ Write**	**Reset State**
RESERVED	31:7	Writes ignored. Must be written with a value of 0x0.		Undefined
FDC_SMASK	6	Writing a 0x1 to this bit enables the local Fast Debug Channel Interrupt	W	Undefined
SWINT1_SMASK	5	Writing a 0x1 to this bit enables the local SWInt1 Interrupt.	W	Undefined
SWINT0_SMASK	4	Writing a 0x1 to this bit enables the local SWInt0 Interrupt.	W	Undefined
PERFCNT_SMASK	3	Writing a 0x1 to this bit enables the l	W	Undefined
TIMER_SMASK	2	Writing a 0x1 to this bit enables the local Timer Interrupt.	W	Undefined
COMPARE_SMASK	1	Writing a 0x1 to this bit enables the local GIC Count/Compare Interrupt.	W	Undefined
WD_SMASK	0	Writing a 0x1 to this bit enables the local WatchDog Timer Interrupt.	W	Undefined

9.6.2.6 Local Map to Pin Registers

This section includes the local map to pin registers for the interrupt types described in Table 9.47. The bit assignments for each of these registers is identical. There is one register per instantiated core. The 'i' indicates a number between 1 and 6 depending on the number of cores in the system.

Table 9.46 Local Map-to-Pin Register Mapping

Offset	Acronym	Register Name
0x0040	GIC_COREi_WD_MAP	Local Watchdog Map-to-Pin register.
0x0044	GIC_COREi_COMPARE_MAP	Local Counter/Compare Map-to-Pin register.
0x0048	GIC_COREi_TIMER_MAP	Local Timer Map-to-Pin register.
0x004C	GIC_COREi_FDC_MAP	Local Fast Debug Channel Map-to-Pin register.
0x0050	GIC_COREi_PERFCTR_MAP	Local Performance Counter Map-to-Pin register.
0x0054	GIC_COREi_SWInt0_MAP	Local Software Interrupt 0 Map-to-Pin register.
0x0058	GIC_COREi_SWInt1_MAP	Local Software Interrupt 1 Map-to-Pin register.

Figure 9.29 GIC Interrupt Map to Pin Register Format

31	30	29 6	5 0
MAP_TO_PIN	MAP_TO_NMI	R	MAP

Table 9.47 Local Map to Pin Registers (Offset 0x0040 - 0x0058 — See above for mapping)

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
MAP_TO_PIN	31	If this bit is set, this interrupt source is mapped to a VPE interrupt pin (specified by the *MAP* field below). Only one of the *MAP_TO_PIN*, *MAP_TO_NMI*, or *MAP_TO_YQ* bits can be set at any one time.	-	0x1 for Timer, PerfCount and SWIntx; 0x0 for WatchDog
MAP_TO_NMI	30	If this bit is set, this interrupt source is mapped to a VPE NMI interrupt pin. Only one of the *MAP_TO_PIN*, *MAP_TO_NMI*, or *MAP_TO_YQ* bits can be set at any one time.	R/W	0x1 for WatchDog; 0x0 for Others
R	29:6	Read as 0x0. Writes ignored. Must be written with a value of 0x0.		0
MAP	5:0	When the *MAP_TO_PIN* bit is set, this field contains the encoded value of interrupts signals *Int[62:0]*. In EIC mode, this represents one less than the EIC interrupt level (e.g. a value of 0x20 represents interrupt level 21). For non-EIC mode, the value represents the CPU interrupt to be asserted (e.g. a value of 0x03 represents interrupt 3), and only values of 0 to 5 are legal.	W	0

9.6.2.7 Core-Other Addressing Register

This register must be written with the correct value before accessing the Core-Other address section.

Figure 9.30 Core-Other Addressing Register Format

31 16	15 0
R	VPENUM

Table 9.48 Core-Other Addressing Register (GIC_COREi_OTHER_ADDRESS — Offset 0x0080)

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
R	31:16	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0
VPENUM	15:0	Number of the register set to be accessed in the Core-Other address space. Note that in the proAptiv core, there is one VPE per core, hence a VPE and a core are the same thing.	R/W	0

9.6.2.8 Core-Local Identification Register

The aliased memory scheme is normally invisible to software when accessing GIC registers within the Core-Local Control Block. What actually happens is that an offset is used to make a subset of the GIC registers appear in the Core-Local addressing Window.

This register reports the Core number that is used as the addressing offset for the Core-Local Control Block.

Figure 9.31 Core-Local Addressing Register Format

31	0
CORENUM	

Table 9.49 Core-Local Identification Register (GIC_COREi_IDENT — Offset 0x0088)

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
CORE*NUM*	31:0	This number is used as an index to the registers within the GIC when accessing the Core-local control block for this core. Note that in the proAptiv core, there is one VPE per core, hence a VPE and a core are the same thing.	R	-

9.6.3 Local Timer Register Descriptions

9.6.3.1 Watchdog Timer Config Register

For more information on the usage of this register, refer to Section 9.4.6.2, "GIC Watchdog Timer".

Figure 9.32 Watchdog Timer Config Register Format

31 8	7	6	5	4	3 1	0
R	WDRESET	WDINTR	WAIT	DEBUG	TYPE	WDSTART

Table 9.50 Watchdog Timer Config Register (GIC_COREi_WD_CONFIG — Offset 0x0090)

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
R	31:8	Read as 0x0. Writes ignored. Must be written with a value of 0x0.		0
WDRESET	7	Status bit which indicates that a Watchdog was responsible for resetting the proAptiv sub-system. A write of 0x1 to this bit of this register automatically clears this bit. This bit needs to survive a watchdog triggered reset.	R/WC	0
WDINTR	6	Status bit which indicates that a Watchdog was responsible for generating this interrupt. A write of 0x1 to this bit automatically clears the bit. Typically this interrupt is routed to the *NMI* interrupt input of the core, but could be routed to another interrupt as well.	R/WC	Undefined
WAIT	5	Stop countdown if the core is in an implementation-defined low power mode (including the mode which is entered on a WAIT instruction). 0x0 - Stop countdown if core is in low power mode. 0x1 - Low power mode has no effect on countdown.	R/W	0
DEBUG	4	Stop countdown if the core is in debug mode. 0x0 - Stop countdown if core is in Debug Mode (CP0 $DEBUG_{DM}$ bit is set). 0x1 - Debug Mode has no effect on countdown.	R/W	0

Table 9.50 Watchdog Timer Config Register (GIC_COREi_WD_CONFIG — Offset 0x0090)*(continued)*

<table>
<tr><th colspan="2">Register Fields</th><th rowspan="2">Description</th><th rowspan="2">Read/ Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td><i>TYPE</i></td><td>3:1</td><td>There are <i>three</i> ways to setup the watchdog timer:
1. It can be setup such that if it decrements to 0x0, it causes an interrupt and then stops.
2. It can be setup such that after the first countdown, the initial value is reloaded and the countdown continues. If on the second trip, the counter reaches 0x0, a reset is broadcast to all cores in the system.
3. The counter can be used as a Programmable Interval Timer (PIT).
<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>WD One Trip Mode. This asserts an interrupt, typically an NMI, and stops.</td></tr><tr><td>0x1</td><td>WD Second Countdown Mode. This asserts <i>SI_Reset</i> on all cores.</td></tr><tr><td>0x2</td><td>PIT Mode. This asserts an interrupt and reloads and keeps going.</td></tr><tr><td>0x3..0x7</td><td>Reserved</td></tr></table></td><td>R/W</td><td>0</td></tr>
<tr><td>WD_START</td><td>0</td><td>Watchdog timer start/stop. Setting this bit starts the Watchdog timer, while clearing the bit stops the timer.

0 - Stop the Watchdog timer
1 - Reload the initial count and start the Watchdog timer.</td><td>R/W</td><td>0</td></tr>
</table>

9.6.3.2 Watchdog Timer Count Register

For more information on the usage of this register, refer to Section 9.4.7.3, "Watchdog Timer Interrupts".

Figure 9.33 Watchdog Timer Count Register Format

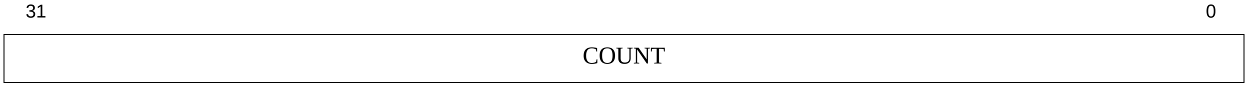


Table 9.51 Watchdog Timer Count Register (GIC_COREi_WD_COUNT — Offset 0x0094)

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
COUNT	31:0	This read-only register indicates the state of the decrementing counter. The width of the counter is 32 bits.	R	Undefined

9.6.3.3 Watchdog Timer Initial Count Register

For more information on the usage of this register, refer to Section 9.4.7.3, "Watchdog Timer Interrupts".

Figure 9.34 Watchdog Timer Initial Count Register Format

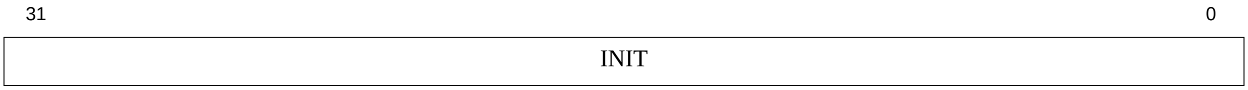


Table 9.52 Watchdog Timer Initial Count Register (GIC_COREi_WD_INITIAL — Offset 0x0098)

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
INIT	31:0	Initial value to be loaded into the Watchdog counter. Needs to be done with the counter disabled; otherwise, the results are UNPREDICTABLE.	R/W	Undefined

9.6.3.4 CompareLo Register

For more information on the usage of this register, refer to Section 9.4.7.3, "Watchdog Timer Interrupts".

Figure 9.35 CompareLo Register Format

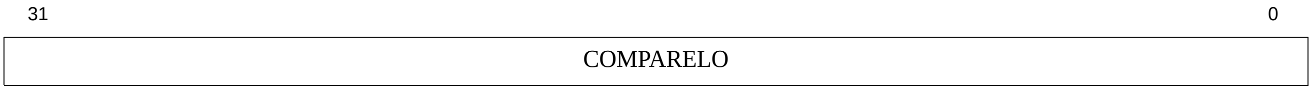


Table 9.53 CompareLo Register (GIC_COREi_CompareLo — Offset 0x00A0)

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
COMPARELO	31:0	When the contents of *GIC_COREi_CompareLo* and *GIC_COREi_CompareHi* registers match the contents of *GIC_SH_CounterLo* and *GIC_SH_CounterHi*, the *CORE*i_Compare interrupt is triggered. This registered interrupt can only be deasserted by writing either the *GIC_COREi_CompareLo* or *GIC_COREi_CompareHi* registers.	R/W	0xFFFF_FFFF

9.6.3.5 Core-Local CompareHi Register

For more information on the usage of this register, refer to Section 9.4.7.3, "Watchdog Timer Interrupts".

Figure 9.36 CompareHi Register Format

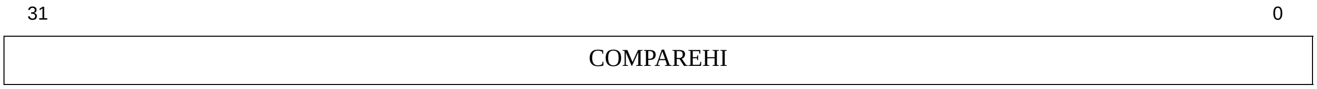


Table 9.54 Core-Local CompareHi Register (GIC_COREi_CompareHi — Offset 0x00A4)

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
COMPAREHI	31:0	See description for *GIC_COREi_CompareLo*. The width of this register matches the width of *GIC_SH_COUNTER*.	R/W	All instantiated bits = 0x1

9.6.3.6 Local EIC Shadow Set Registers

These registers are only instantiated if the GIC is configured to include EIC mode. There are 64 EIC Shadow registers located at offset addresses 0x0100 - 0x01FC. These registers are mapped as follows:.

Table 9.55 Local Map-to-Pin Register Mapping

Offset	Interrupt Source	Register Acronym	Offset	Interrupt Source	Acronym
0x0100	0	GIC_COREi_EICSRCSS0	0x0180	32	GIC_COREi_EICSRCSS32
0x0104	1	GIC_COREi_EICSRCSS1	0x0184	33	GIC_COREi_EICSRCSS33
0x0108	2	GIC_COREi_EICSRCSS2	0x0188	34	GIC_COREi_EICSRCSS34
0x010C	3	GIC_COREi_EICSRCSS3	0x018C	35	GIC_COREi_EICSRCSS35
0x0110	4	GIC_COREi_EICSRCSS4	0x0190	36	GIC_COREi_EICSRCSS36
0x0114	5	GIC_COREi_EICSRCSS5	0x0194	37	GIC_COREi_EICSRCSS37
0x0118	6	GIC_COREi_EICSRCSS6	0x0198	38	GIC_COREi_EICSRCSS38
0x011C	7	GIC_COREi_EICSRCSS7	0x019C	39	GIC_COREi_EICSRCSS39
0x0120	8	GIC_COREi_EICSRCSS8	0x01A0	40	GIC_COREi_EICSRCSS40
0x0124	9	GIC_COREi_EICSRCSS9	0x01A4	41	GIC_COREi_EICSRCSS41
0x0128	10	GIC_COREi_EICSRCSS10	0x01A8	42	GIC_COREi_EICSRCSS42
0x012C	11	GIC_COREi_EICSRCSS11	0x01AC	43	GIC_COREi_EICSRCSS43
0x0130	12	GIC_COREi_EICSRCSS12	0x01B0	44	GIC_COREi_EICSRCSS44
0x0134	13	GIC_COREi_EICSRCSS13	0x01B4	45	GIC_COREi_EICSRCSS45
0x0138	14	GIC_COREi_EICSRCSS14	0x01B8	46	GIC_COREi_EICSRCSS46
0x013C	15	GIC_COREi_EICSRCSS15	0x01BC	47	GIC_COREi_EICSRCSS47
0x0140	16	GIC_COREi_EICSRCSS16	0x01C0	48	GIC_COREi_EICSRCSS48
0x0144	17	GIC_COREi_EICSRCSS17	0x01C4	49	GIC_COREi_EICSRCSS49
0x0148	18	GIC_COREi_EICSRCSS18	0x01C8	50	GIC_COREi_EICSRCSS50
0x014C	19	GIC_COREi_EICSRCSS19	0x01CC	51	GIC_COREi_EICSRCSS51
0x0150	20	GIC_COREi_EICSRCSS20	0x01D0	52	GIC_COREi_EICSRCSS52
0x0154	21	GIC_COREi_EICSRCSS21	0x01D4	53	GIC_COREi_EICSRCSS53
0x0158	22	GIC_COREi_EICSRCSS22	0x01D8	54	GIC_COREi_EICSRCSS54
0x015C	23	GIC_COREi_EICSRCSS23	0x01DC	55	GIC_COREi_EICSRCSS55
0x0160	24	GIC_COREi_EICSRCSS24	0x01E0	56	GIC_COREi_EICSRCSS56
0x0164	25	GIC_COREi_EICSRCSS25	0x01E4	57	GIC_COREi_EICSRCSS57
0x0168	26	GIC_COREi_EICSRCSS26	0x01E8	58	GIC_COREi_EICSRCSS58
0x016C	27	GIC_COREi_EICSRCSS27	0x01EC	59	GIC_COREi_EICSRCSS59
0x0170	28	GIC_COREi_EICSRCSS28	0x01F0	60	GIC_COREi_EICSRCSS60
0x0174	29	GIC_COREi_EICSRCSS29	0x01F4	61	GIC_COREi_EICSRCSS61
0x0178	30	GIC_COREi_EICSRCSS30	0x01F8	62	GIC_COREi_EICSRCSS62
0x017C	31	GIC_COREi_EICSRCSS31	0x01FC	63	GIC_COREi_EICSRCSS63

Figure 9.37 Local EIC Shadow Set Register Format

31 … 4	3 … 0
R	EIC_SSn

Table 9.56 Local EIC Shadow Set Registers (GIC_COREi_EICSSi — Offset 0x0100 - 0x01FC)

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
R	31:4	Read as 0x0. Writes ignored. Must be written with a value of 0x0.	-	0
EIC_SSn	3:0	Encoded value that indicates the Shadow Set number to use for this particular interrupt.	R/W	Undefined

9.6.3.7 Core-Local DINT Group Participate Register

When bit 0 of this register is set, the local core monitors the state of the DINT_Send_to_Group register in the Shared register set, as well as the EJ_DINT_IN pin for debug activity. Refer to Section 9.4.10, "Debug Interrupt Generation" for more information.

Figure 9.38 Core-Local EIC DINT Group Participate Register Format

31 … 1	0
R	DINT_GP

Table 9.57 Core-Local DINT Group Participate Register (GIC_Vx_DINT_PART — Offset 0x3000)

Register Fields		Description	Read/ Write	Reset State
Name	Bits			
RESERVED	31:1	Reads as 0x0. Writes ignored. Must be written with a value of 0x0.	R	0x0
DINT_GP	0	If this bit is set, the local core pays attention to the *DINT_Send_to_Group* register as well as the external *EJ_DINT_IN* signal pin. For this case, when the *Send_DINT* bit within the *DINT _Send_to_Group* register is asserted (or the external *EJ_DINT_IN* signal is asserted), the local core will have its *EJ_DINT* or *EJ_DINT_1* signal asserted. If this bit is clear, the local core is not affected by the *DINT_Send_to_Group* register nor the external *EJ_DINT_IN* pin signal.	R/W	0x1

See Chapter 16, “Multi-CPU Debug” on page 611 for more information about how this register is used.

9.6.3.8 Core-Local DebugBreak Group Register

When the local core enters Debug Mode (denoted by the local *EJTAG_TAP.DebugM* bit being asserted), this register defines which other cores in the system will subsequently also receive a Debug Interrupt. This allows multiple cores to be synchronized to a single software debugger by entering debug mode somewhat simultaneously.

Figure 9.39 Core-Local EIC DINT Group Participate Register Format

31	0
JOIN_DB	

Table 9.58 Core-Local DebugBreak Group Register (GIC_Cx_BRK_GROUP — Offset 0x3080)

Register Fields				
Name	Bits	Description	Read/ Write	Reset State
JOIN_DB	31:0	Each bit in this register represents a core in the system. If the bit is set, the corresponding core will have its *EJ_DINT* or *EJ_DINT_1* signal asserted when the local core enters Debug Mode. If the bit is clear, the corresponding core is not affected when the core enters Debug Mode. The bit which represents the local core cannot be used to disable Debug Mode for the local core. For example, if the local core is represented by bit *i*, clearing bit *i* will NOT disable Debug Mode for the local core.	R/W	All zeros

See Chapter 17, “Multi-CPU Debug” on page 1817 for more information about how this register is used.

9.7 GIC User-Mode Visible Section

The Shared, Core-local, and Core-other sections are meant to be located in privileged system virtual address space, in which only kernel mode software can initialize and update the interrupt controller.

A separate 64KB address space is allocated so that it may be mapped to user-mode virtual address space. Within this address space are aliases for GIC registers that are read so often that it makes sense to make them available to user-mode programs without requiring a system call. The aliases for these registers are read-only. Currently, the only registers that are aliased into this space are the shared Counter registers.

The addresses for the registers within the User-Mode Visible Section of the GIC are calculated as follows:

```
SharedSection_Register_Physical_Address = GIC_baseaddress +
UMVisible_Section_baseoffset + Register_Offset
```

Table 9.59 User-Mode Visible Section Register Map

Register Offset	Name	Type	Description
0x0000	GIC CounterLo (`GIC_SH_CounterLo`)	R	Read-only alias for GIC Shared CounterLo.
0x0004	GIC CounterHi (`GIC_SH_CounterHi`)	R	Read-only alias for GIC Shared CounterHi.
Any Other Offsets	Reserved		Reserved for future extensions.

Chapter 10

Instruction and Data Scratch Pad RAM

The instruction scratchpad RAM (ISPRAM) and data scratchpad RAM (DSPRAM) options on the proAptiv Multiprocessing System™ Multiprocessing System are designed to provide low-latency access to on-chip memories. Separate SPRAM blocks exist for instruction and data references. The SPRAM ports are accessed in parallel with the caches. This saves a number of cycles that would normally be required when going through the BIU and the master OCP interface of the proAptiv Multiprocessing System core. Throughout this chapter, the term SPRAM is used to refer to the ISPRAM and DSPRAM memories.

This chapter contains details of the SPRAM interfaces and reference designs. The chapter contains the following major sections:

- Section 10.1 "Scratchpad RAM (SPRAM) Features"
- Section 10.2 "SPRAM Overview"
- Section 10.3 "SPRAM Initialization"
- Section 10.4 "SPRAM Clocking"

10.1 Scratchpad RAM (SPRAM) Features

The MIPS32® proAptiv Multiprocessing System core scratchpad has the following features:

- SPRAM is supported for instruction and data references.
- Each SPRAM block occupies one continuous region in the physical address space. The SPRAM wrappers contain the base physical address and size information.
- SPRAM is virtually indexed by the core. There is no hardware support to avoid virtual aliasing.
- Size of SPRAM may range from 4 KB to 1 MB in factors of 2.
- Data Access granularity
  - Read: 64-bit (1 doubleword).
  - Write: maximum write width is 64 bits, minimum write width is 8 bits.
- Instruction Access granularity
  - Read/Write: 64 bits of instruction plus 6 bits of precode. Smaller writes are not supported.
- SPRAM control supports single or multi-cycle access. For maximum frequency, the SPRAM access time should be less than the cache access time. For larger size SPRAMs, the integrator may choose a multi-cycle access

  - For data references, the multi-cycle accesses can be pipelined. SPRAM data needs to be returned in the requested order.

  - For instruction references, requests will be retried if the data is not available at the single cycle point.

- Multi-cycle data scratchpad RAM access is non-blocking.

- Multi-cycle instruction scratchpad RAM access is blocking (within a TC).

- The scratchpad RAM is not required to hold the last read value.

- The data scratchpad has independent tag and data ports. The tag and data arrays are always read together for the instruction scratchpad. The scratchpad RAM does not have a traditional cache tag array. Instead, it has registers holding the SPRAM configuration information.

- SPRAM access hit supersedes cache access hit.

- User may implement a DMA port to the scratchpad RAM. In the reference designs, an OCP slave port is provided.

- The proAptiv Multiprocessing System core provides integrated BIST support for single-cycle latency SPRAM.

- Optional parity protection is supported for SPRAM.

  - Instruction: 1b per 8b of instruction, 1b for 7b precode

  - Data Parity: 1b per 8b of data

10.2 SPRAM Overview

A Scratchpad RAM can be used stand-alone or combined with data or instruction cache. The existence of a scratchpad must be selected at build time.

The SPRAM array, like the cache arrays, is indexed with a virtual address and the “tag comparison” (really just decode logic for the SPRAM) is performed using a physical address. Since the SPRAM size can be larger than the 4 KB minimum page size, it is possible to have virtual aliasing in the SPRAM. Virtual aliasing occurs when a single physical address is accessed via two different virtual addresses that can simultaneously reside in memory. This is not a problem on cores using the Fixed Mapping Translation MMU. For cores with TLB-based MMUs, this can be avoided by accessing the SPRAM through unmapped (kseg0/1) addresses or using using a TLB page. This is not handled by hardware and programmers must be aware of it.

The reference designs contain 8 KB SPRAM arrays, with one cycle latency and a simple DMA port. A user can choose to implement a custom SPRAM with different size, latency, and other desired characteristics.

During normal operation, it will be impossible for a reference to hit in both SPRAM and data cache. If this error condition does occur via manipulation of the cache or SPRAM tags, the SPRAM supersedes the data cache hit. Note that this also means that a CACHE HitInvalidate operation to such a line that exists in both SPRAM and cache will not invalidate the cache entry.

The scratchpad interface consists of a core-side interface as well as an optional DMA interface. MIPS provides a reference design for the external SPRAM module called `imp_sp` and `imp_isp`. These include wrappers that instantiates a SPRAM SRAM array. The reference module can be replaced with a customized SPRAM implementation. For timing reasons, the arbitration logic for the SPRAM DMA interface is located within the `imp_cpu` hierarchy.

10.2.1 SPRAM Differences Versus a Cache

SPRAM behavior differs from cache in the following key ways:

- Software must ensure a SPRAM entry has been initialized before it is read, to avoid reading spurious data.
- The SPRAM does not refill automatically. The data SPRAM is normally initialized with stores or DMA writes to the desired address range. The instruction SPRAM can be initialized with Index Store Data CACHE ops or DMA writes.
- Store operations which hit in the data SPRAM do not produce writes to main memory, unlike write-through stores that hit in the cache and write to main memory.

NOTE: The I-Cache Fill and Fetch&Lock cacheops will refill the given line into the I-Cache even if that address hits to the ISPRAM. This is not recommended since normal fetches will hit in the ISPRAM and ignore the I-Cache contents.

10.2.2 Uncacheable References to SPRAM

SPRAM can be mapped to either cached or uncached space. The address decode and comparison for SPRAM is done regardless of the cacheability attribute.

10.2.3 Independent Tag/Data Accesses

The data SPRAM interface has independent tag and data ports. This is done to aid the efficiency of stores. A store must perform a lookup to determine if/where to write the data, then the actual data must be written. Because the lookup does not need to access the data array, these operations can occur in parallel if the data writes are buffered within the core.

Because there are no stores to the instruction SPRAM, the tag and data ports are linked. Reads will always access both the tag and data port at the same index. Writes will target either the tag or data.

10.2.4 SPRAM Tag Reads and Writes

The interface allows for SPRAM “tag” values to be read and written. The tag values are read/written by the CACHE instruction. This can optionally provide a mechanism for software to determine the SPRAM base and size configuration and change it. The reference design shows one possible use for this interface - software can probe the SPRAM to determine the base address and whether it is enabled. These values are also write-able, allowing software to dynamically configure the SPRAM parameters.

10.2.5 Multiple Cycle Data SPRAM Access

For a one-cycle latency SPRAM, the scratchpad interface will achieve cache-like access timing. However, the scratchpad interface also supports SPRAM that has a multi-cycle latency.

For a data scratchpad read, when the data from SPRAM is not ready, the processor will register the load in a load buffer and return data to the main pipeline when data is available from SPRAM. No stall of pipeline is necessary unless the result register is used by a following instruction.

10.2.6 Multiple Cycle Instruction SPRAM Access

For an instruction scratchpad read, multi-cycle latency causes more of a problem. The instruction fetch pipeline does not have the ability to stall. If the instruction data is not returned at the expected time, the request will be retried. This will add a minimum of 3 cycles of latency to all ISPRAM fetches.

When the core is operating in multi-threaded mode, this gets even trickier. Fetch requests from the different TCs can be intermixed. It is recommended that a multi-cycle ISPRAM have a buffer per TC that holds the last requested fetch to allow TCs to make forward progress.

10.2.7 Backstalling the SPRAM Data Port

The backstall mechanism is not really needed if the SPRAM can keep track of all the outstanding requests. If that is not the case, SPRAM is allowed to backstall the core if it is busy, via assertion of the *SP_ram_busy or ISP_ram_busy* signals.

This backstall mechanism may be useful if the customer implements a multi-cycle non-pipelined version of data SPRAM. The scratchpad should assert the *SP_ram_busy* when it cannot accept another request in the next cycle. The core will stall its pipeline only if it has a pending SPRAM access and it is about to enter the ER stage of the pipeline.

The above mentioned mechanism only applies to the SPRAM data port, not the tag. The tag port always requires fixed single-cycle latency

10.2.8 Access Granularity

A data SPRAM read returns either 1 word (32b) or 1 doubleword (64b) of data (plus parity/ECC). There are two read strobes controlling access to the upper and lower word of the data array. Since many core accesses are 32b or less, banking the SPRAM array and only reading the selected bank can yield power savings. Alternatively, the OR of the read strobes can be used to access a single wider array.

For word accesses, the processor core uses the lower 32b of the read data bus. If only the upper read strobe is asserted, the upper 32b word be returned on the lower 32b of the data bus. Additional alignment and shifting is handled within the core. The maximum write width is 64 bits and partial write is enabled through the byte enable signals, *SP_data_wren_ag[7:0].*

For DMA access to data SPRAM, the read will always be 64 bits wide. The maximum write width is 64 bits and partial write is enabled using the *OC_DMA_MDataByteEn[7:0]* signals.

On the instruction SPRAM, both reads and writes will always be 70 bits wide (64b of data + 6b of precode) (plus parity). The additional precode data will be generated by the core during a DMA write. Data on the OCP Slave bus will only be 64b wide.

10.2.9 Connecting I/O Devices to the Data Scratchpad Interface

In addition to, or perhaps instead of, an SRAM array, it is possible to connect I/O devices to the SPRAM interface. Connecting I/O devices to the scratchpad interface allows low latency, high throughput access to critical I/O devices in the system. To accomplish this, the integrator must ensure that the behavior of the I/O devices meets the same requirements as the SPRAM.

Connecting an I/O device to the ISPRAM is not recommended.

10.2.10 Null Connection to Unused SPRAM Interface

The presence of scratchpads must be chosen when the core is built. Even if one or both of the SPRAM interfaces are present, there does not need to be arrays connected to them. If a interface is not going to be used, then the *[I]SP_Present* input signal to the core should be driven low. All other input signals to the core for the unused SPRAM interface should also be tied low, to avoid floating inputs. All output signals from the core related to the unused SPRAM interface can be left unconnected.

10.3 SPRAM Initialization

Since the scratchpad is really a RAM-based structure, it must be initialized with valid data before it can be used. Following are few ways to initialize the data SPRAM.

- DMA: The RAM array can be initialized from the system using DMA writes.
- Stores: For data SPRAM, the array can be initialized with normal store instructions that hit in the SPRAM region.
- CACHE Index Store Data instruction: Indexed cache operations can be forced to go to the SPRAM by setting the SPR bit in the Coprocessor0 *ErrCtl* register. When this bit is set, it is possible to use the Index Store Data flavor of the CACHE instruction to move data from the *DataLo/DataHi* Cop0 registers into the SPRAM. This mechanism does not require any backing memory and can even be used to load the SPRAM from an EJTAG probe for early system bringup. For the data SPRAM, using stores to initialize the array is usually a much more efficient mechanism.

10.3.1 ISPRAM Boot

Uncached requests can still be serviced by the SPRAMs. On MIPS CPUs, the boot vector is located at an uncacheable address. Since the SPRAM has cache-like timing even when responding to uncached accesses, it can run much faster. This can make booting directly from the ISPRAM an interesting possibility.

Note: When fetching from uncached addresses, even if they hit in the SPRAM, the core will only use 32b at a time instead of 64b. This will reduce the performance versus SPRAM hits in cached space, but will still be much better than normal uncached accesses.

In order to boot from the ISPRAM, the instructions must be loaded into the array before the core can start executing them. The reference design does include some support for this via the *ispram_boot* internal signal. By default, this signal is statically driven to 0, but commented out RTL shows how to connect it to one of the sideband external signals to allow it to be dynamically controlled. There is example RTL for either directly connecting it or synchronizing the input signal - the latter is recommended.

In order to load the ISPRAM via DMA and boot directly from it:

- Set *ispram_boot* = 1 while *SI_Reset* = 1
  - This sets the base address to the physical address of the boot vector (either 0x1fc0_0000 or *SI_ExceptionBase* if *SI_UseExceptionBase* = 1) and sets the enable bit.
- While in reset, the DMA port will be inactive (core deasserts Accept signals)
- After *SI_Reset*->0, hold *ispram_boot* = 1 until the ISPRAM has been loaded via DMA.

  - Note: the DMA port is held inactive while the core is in reset, thus the DMA can only happen after reset has been deasserted
  - This causes *ISP_dma_stallreq_xx* = 1 which gives the DMA priority over core requests .
  - And also sets *ISP_datavld_nxt_if* = 0, which indicates that the data is not available yet and the core will retry any accesses that hit in the ISPRAM.
- Once the ISPRAM has been loaded, *ispram_boot* should be deasserted, allowing the core accesses to hit out of the ISPRAM.

10.3.2 ISPRAM Precode Bits

Six precode bits are included with every 64b of instruction data in the ISPRAM array. These bits contain information about branches and jumps. Having this information allows the fetch unit to quickly react to the potential change of flow and start fetching along the predicted path. These precode bits are not used when executing MIPS16e™ code.

When the ISPRAM array is loaded using Index Store Data CACHE instructions or by using the core's DMA interface, the precode bits are generated automatically and sent out as part of the write data. So, for many systems, nothing special will need to be done with the precode bits. In some custom ISPRAM blocks, however, it may not be possible to utilize the precode blocks within the core. Two examples are if the ISPRAM block contains a ROM array or if the core DMA interface is not used.

Table 10.1 Precode Bits

Name	Bit	Description
L/M	6	This bit has two different meanings depending on the state of the B and J bits. 1. If B is set, then this bit is set if the branch is a branch likely instruction. 2. If J is set, then this bit is set if the Jump instruction is a JALX instruction. If either of the B or J bits are set, then the IFU fetches a delay slot instruction. When both B and J bits are set, this indicates that both instructions decoded to look like branches, Jumps, or ERET instructions. This can happen if one instruction isn't really an instruction, but is instead data.
X	5	0: branch/jump is in bits[31:0] 1: branch/jump is in bits [63:32]
B	4	Branch instruction
J	3	Jump instruction
S	2	Indicates a subroutine call. Return address will be pushed onto return prediction stack
G	1	Indicates Jump Register is not predicted
U/R	0	On branches, indicates an unconditional branch On jumps, indicates a return

Table 10.2 MIPS32 Control Transfer Instructions

Instruction	Precoding	Notes
`B (BEQ rs==rt)`	BU	Does not use branch predictor.
`BAL (BGEZAL r0)`	BSU	Does not use branch predictor. push PC+8 onto RPS
`BC1[TF]`	B	
`BC1[TF]L`	B	
`BC2[TF]`	B	
`BC2[TF]L`	B	
`BEQ (rs != rt)`	B	
`BEQL`	B	
`B[GL][ET]Z`	B	
`BGEZAL (rs != r0)`	B	
`BLTZAL`	B	
`B[GE,LT]ZALL`	B	
`B[GL][ET]ZL`	B	
`BNE`	B	
`BNEL`	B	
`BPOSGE32`	B	Instruction from MIPS DSP ASE
`DERET`	G	decode in IS
`ERET`	G	decode in IS
`J`	J	
`JAL`	JS	push PC+8 onto RPS
`JALR[.HB] (rd = $31)`	JSG	push PC+8 onto RPS possible MIPS16e mode change
`JALR[.HB] (rd != $31)`	JG	possible MIPS16e mode change
`JALX`	JS	push PC+8 onto RPS switch to MIPS16e mode
`JR (rs = $31)`	JR	possible speculative MIPS16e mode change
`JR (rs != $31)`	JGSR	possible MIPS16e mode change
`JR.HB (rs = $31)`	JGR	possible MIPS16e mode change pop RPS but don’t use
`JR.HB (rs != $31)`	JG	possible MIPS16e mode change
`ILLEGAL`	BJG	If both instructions decode as a branch/jump, set this to let fetch unit know there is a strange situation that needs resolving. This can happen when data is packed with instructions, when precoding MIPS16e instruction data, or when there is an illegal code sequence.

10.4 SPRAM Clocking

The SPRAM block receives two clocks from the core, *[I]SP_gclk* and *[I]SP_gfclk*. *[I]SP_gclk* is a global gated clock, while *[I]SP_gfclk* is free-running. When top level clock gating is implemented, *[I]SP_gclk* is not active in the low power sleep mode entered using the WAIT instruction. The processor enters sleep mode only if there are no pending SPRAM transactions. When the processor core is in sleep mode, a DMA request will re-enable *[I]SP_gclk* in order to process the request.

For the best power management, most of logic in the SPRAM block should reside on the gated *[I]SP_gclk*. Only the minimal logic needed to detect a DMA request and wake up the core needs to reside on the free-running *[I]SP_gfclk*.

10.4.1 Scratchpad Reference Design

The reference scratchpad design (called `imp_[i]sp`) supports a basic scratchpad implementation. It is configurable within certain constraints:

- SPRAM size can range from 4 KByte to 1 MByte. The supported sizes are 4KB, 8KB, 16KB, 32KB, 64KB, 128 KB, 256 KB, 512 KB, or 1 MB
- SPRAM tag has a base address and a size register used for hit detection. Both base and size register are accessible through the CACHE instruction.
- The address range must be naturally aligned (i.e. a 64KB SPRAM's base address must be on a 64KB boundary).
- The array always returns data in a single cycle.
- The scratchpad contains an OCP DMA slave port.

Following are some considerations of the reference SPRAM design which are not covered in the previous sections.

10.4.2 Tag Registers in the Reference SPRAM

Using the CACHE instruction, it is possible to read or write the "tag" value associated with the SPRAM. To provide a common software interface, it is recommended that all SPRAM implementations provide some standard configuration information via this mechanism.

In the reference SPRAM wrapper, the "tag" of SPRAM consists of a base address register and a size register. If the SPR bit in the *ErrCtl* register is set, an Index Load Tag CACHE instruction reads the SPRAM tag and place the contents in the *TagLo* register, while an Index Store Tag CACHE instruction writes the SPRAM tag with the data from *TagLo* register. Bit3 of the index is used to select between base address and size register; when bit3 =1, the size register is selected, otherwise the base address register is selected. The format of the base and size registers are shown in Table 10.3 and Table 10.4, respectively.

Table 10.3 Format of the Base Address Register in the Reference SPRAM Wrapper

Field	Description
sp_base_xx[31:12]	Base address of the SPRAM region
sp_base_xx[11]	SPRAM valid

Table 10.4 Format of the Size Register in the Reference SPRAM Wrapper

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>sp_size_xx[31:12]</td><td>Size of the SPRAM:
<table>
<tr><th>SPRAM size</th><th>Value</th></tr>
<tr><td>4KB</td><td>20’h00001</td></tr>
<tr><td>8KB</td><td>20’h00002</td></tr>
<tr><td>16KB</td><td>20’h00004</td></tr>
<tr><td>32KB</td><td>20’h00008</td></tr>
<tr><td>64KB</td><td>20’h00010</td></tr>
<tr><td>128KB</td><td>20’h00020</td></tr>
<tr><td>256KB</td><td>20’h00040</td></tr>
<tr><td>512KB</td><td>20’h00080</td></tr>
<tr><td>1M</td><td>20’h00100</td></tr>
</table>
</td></tr>
</table>

Software can then query or modify these registers to determine the base and size information. An Index Load Tag will read bits [31:12] from the base address or size register and write them into bits [31:12] of the *TagLo* register. Bit [11] of the base address register serves as a valid bit for SPRAM and will be written into the valid field (bit [7]) of the *TagLo* register. Similarly, an Index Store Tag will write bits [31:12] of the *TagLo* register into bits [31:12] of the base address or size register, and the *TagLo* valid field into the valid bit of base address register.

The reset value of base address information is incorporated into the SPRAM via `‘define` within the reference SPRAM module. The reset value of the size tag is based on the actual size of the array.

10.4.3 Enabling SPRAM Access

After power up, the reference SPRAM is always set to invalid through the reset of bit [11] of the base address register. So a CACHE Index Store Tag instruction is needed to enable the SPRAM. The core comparison logic uses bit [11] of *[I]SP_tag_rdata_xx[31:11]* as the valid bit of SPRAM and an access can hit on SPRAM only when this bit is set.

When a custom SPRAM is implemented, this bit should be set accordingly for the design.

10.4.4 SPRAM BIST Support

The core includes an integrated SPRAM BIST controller which can provide BIST support for single-cycle latency SPRAM. The integrated SPRAM BIST controller is capable of supporting two algorithms, March C+ or IFA-13 (IFA-13 includes support for retention testing).

When integrated memory BIST is running, the SPRAM array is tested in parallel with other sub arrays of the instruction and data caches and trace memory.

A custom RAM BIST module is also possible. For a multi-cycle SPRAM implementation, custom BIST is required since the integrated controller only accommodates single-cycle access.

10.4.5 SPRAM Parity Support

Parity protection is optionally enabled for SPRAM. A parity error on a SPRAM read will either cause a CacheErr exception (for a load or fetch) or an error response on the OCP bus (DMA access). The CacheErr parity error detection logic resides in the core. For the reference design, if parity is enabled, it must be supported by the instruction cache, data cache and both SPRAM arrays.

From the reference SPRAM module, the outputs *SP_parity_present, SP_ecc_present, and ISP_parity_present* indicate whether each SPRAM array is parity protected. If a custom SPRAM module is built, users might choose not to check parity for SPRAM even though parity checking for instruction and data caches is enabled; in this case, the output should be de-asserted and no parity checking will be done for that SPRAM.

Chapter 11

Hardware and Software Initialization

A proAptiv Multiprocessing System contains only a minimal amount of hardware initialization and relies on software to fully initialize the device.

This chapter contains the following sections:

- Section 11.1 “Hardware-Initialized Processor State”
- Section 11.2 “Software-Initialized Processor State”

11.1 Hardware-Initialized Processor State

The proAptiv Multiprocessing System is not fully initialized by hardware reset. Only a minimal subset of the processor state is cleared. This is enough to bring the CPU 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.

11.1.1 Coprocessor 0 State

Much of the hardware initialization occurs in Coprocessor 0:

- *Random* - cleared to maximum value on Reset
- *Wired* - cleared to 0 on Reset
- $Status_{BEV}$ - set to 1 on Reset
- $Status_{TS}$ - cleared to 0 on Reset
- $Status_{NMI}$ - cleared to 0 on Reset
- $Status_{ERL}$ - set to 1 on Reset
- $Status_{RP}$ - cleared to 0 on Reset
- $CDMMBase_{EN}$ - cleared to 0 on Reset
- $WatchLo_{I,R,W}$ - cleared to 0 on Reset
- *Config* fields related to static inputs - set to input value by Reset
- $Config_{K0}$ - set to 010 (uncached) on Reset

- *$Config_{KU}$* - set to 010 (uncached) on Reset
- *$Config_{K23}$* - set to 010 (uncached) on Reset
- *$Debug_{DM}$* - cleared to 0 on Reset (unless EJTAGBOOT option is used to boot into Debug Mode, as described in Chapter 14, “EJTAG Debug Support”.
- *$Debug_{LSNM}$* - cleared to 0 on Reset
- *$Debug_{IBusEP}$* - cleared to 0 on Reset
- *$Debug_{DBusEP}$* - cleared to 0 on Reset
- *$Debug_{IEXI}$* - cleared to 0 on Reset
- *$Debug_{SSt}$* - cleared to 0 on Reset

11.1.2 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.

11.1.3 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.

11.1.4 Static Configuration Inputs

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

11.1.5 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.

11.2 Software-Initialized Processor State

Software is required to initialize parts of the device, as described below.

11.2.1 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.

11.2.2 TLB

Because of the hidden bit indicating initialization, the CPU does not initialize the TLB upon Reset. This is an implementation-specific feature of the proAptiv Multiprocessing System CPU and cannot be relied upon if writing generic code for MIPS32/64 processors.

11.2.3 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.

11.2.4 Coprocessor 0 State

Miscellaneous COP0 states need to be initialized before exiting 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), and *SW0* and *SW1* (Software Interrupts) should be cleared.

- *Config*: *K0* should be set to the desired Cache Coherency Algorithm (CCA) prior to accessing kseg0.

- *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.

11.3 Boot and CMP Bringup

After the system is reset and released, all cores configured in hardware to power up will execute their boot sequence. Typically, CPU0 powers up, while all other CPUs are configured to remain powered down. Alternatively, all CPUs can be hardware configured to remain powered down to be awakened through a hardware signal connected to SOC-specific logic.

After system reset, all caches are in an unknown state and must be initialized. It is advisable for core0 to initialize the L2 cache prior to powering up the other cores, but this is not required if other synchronization methods are utilized. For L1 caches, this is expected to be done using IndexStTag ops running on the same CPU. Prior to the data cache being initialized, processing an intervention would cause unpredictable results, potentially corrupting main memory with random data. Thus, the system starts with all of the cores outside the coherence domain until explicitly enabled by software.

```
Core0:
Initialize cop0 state
Initialize L2 Cache
Initialize GCR state
Startup other cores if needed
CoreN:
Initialize L1 Caches
Enable Coherence
Switch to coherent CCA
```

Chapter 12

Floating-Point Unit

This chapter describes the optional MIPS32® Floating-Point Unit (FPU) and contains the following sections:

12.1 Features Overview

The FPU is provided via Coprocessor 1 (CP1). 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.
- Separate and independent arithmetic and data channels
- 1:1 frequency ratio to proAptiv core.
- 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 subtract from zero instructions.

- Supports dual-issue coprocessor 1 interface.

- Hardware support for denormalized numbers. Denormalized numbers are supported in hardware for the add, subtract, compare, and convert functions, but rely on a software handler to operate on the denormalized multiply, divide, and square root functions.

- 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.

12.2 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.

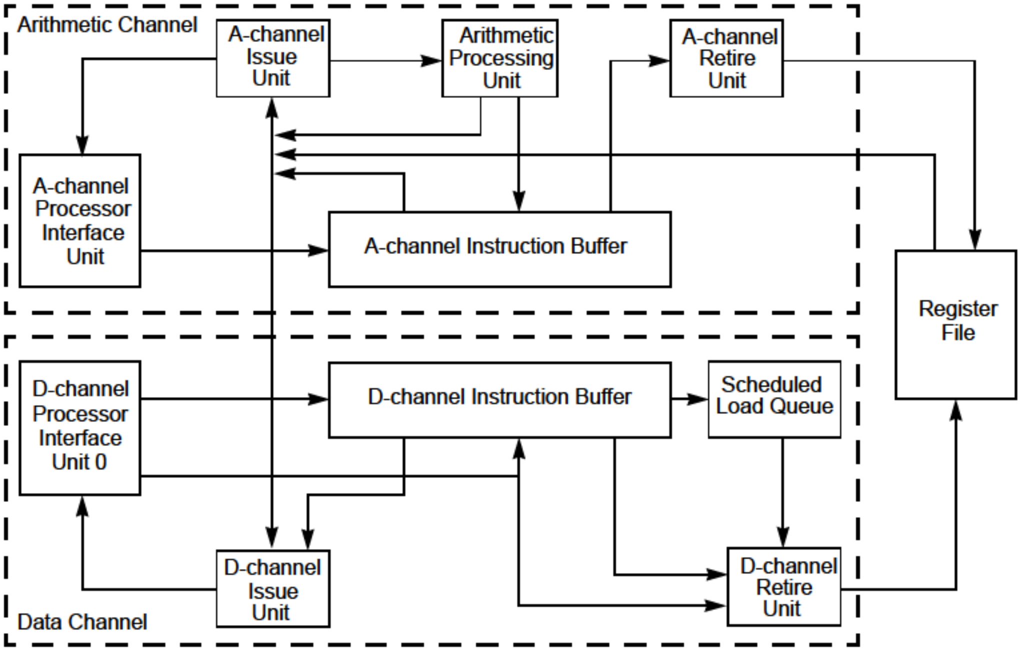
12.3 Enabling the Floating-Point Coprocessor

Coprocessor 1 is enabled by setting the CU1 bit in the CP0 *Status* register. When this bit is cleared, Coprocessor 1 is disabled, and any attempt to execute a floating-point instruction causes a *Coprocessor Unusable* exception.

12.4 Architectural Overview

As shown in Figure 12.1, the FPU is divided into two channels, arithmetic and data. The data channel (D-channel) handles transferring data to and from the FPU. The arithmetic channel (A-channel) handles all arithmetic operations.

Figure 12.1 FPU Block Diagram


Arithmetic Channel
A-channel Issue Unit
Arithmetic Processing Unit
A-channel Retire Unit
A-channel Processor Interface Unit
A-channel Instruction Buffer
Register File
D-channel Processor Interface Unit 0
D-channel Instruction Buffer
Scheduled Load Queue
D-channel Issue Unit
D-channel Retire Unit
Data Channel

Below are the names of the major blocks within the FPU.

- Arithmetic and Data channel Processor Interface Units
- Arithmetic and Data channel Issue Units
- Arithmetic and Data channel Instruction Buffers
- Arithmetic Processing Unit
- Arithmetic and Data channel Retire Units
- Scheduled Load Queue Unit

Each of these blocks is described in the following subsections.

12.4.1 Instruction Flow Through the FPU

The FPU is logically divided into three phases:

- Phase 1 — Allocate
- Phase 2 — Issue
- Phase 3 — Retire

All instructions in the FPU go through these three phases.

12.4.1.1 Allocate

When an instruction is received by the Processor Interface Unit (PIU), it is allocated an entry in the buffer. The buffer entries serve as holding locations for instruction data as the instruction passes through each phase of the FPU. There is one buffer entry per instruction. Subsequent downstream blocks update the associated buffer entry as the instruction proceeds down the pipeline until it is retired in the retire unit and the buffer entry is de-allocated.

12.4.1.2 Issue

An instruction is issued when it has all of the data required to execute the instruction. An instruction is retired when the instruction has completed and it is safe to write the result to the register file. Instructions that have been allocated and not yet retired are considered active.

Instructions within each channel are issued in order. Issues between the two channels are not necessarily in order. One of the channels can often issue ahead of the other channel.

12.4.1.3 Retire

Instructions within each channel are retired in order. Retires between the two channels may be slightly out of order. An instruction is allowed to retire if it is the oldest outstanding instruction, or if there are no RAW or WAR hazards with older instructions in the other channel. The retire block is responsible for determining when an instruction can be retired and the corresponding buffer entry de-allocated. The retire block is also responsible for cancelling an instruction in the arithmetic unit if that instruction has been killed or nulled.

Hardware moves ‘To-Coprocessor’ instructions to the Scheduled Load Queue shown in Figure 12.1 if the instruction was complete except for receiving the data from the integer core. These instructions are retired and set aside with just enough information to complete the register file write when the To-Coprocessor data comes back.

12.4.2 Processor Interface Units

There is one Processor Interface Unit (PIU) per channel that controls all CP1 interactions on that channel. There is one PIU for arithmetic operations and one PIU for load/store operations.

The arithmetic and data channel Processor Interface Units performs the following functions.

- Controls all CP1 interface transactions.
- Receives instructions from the CP1 interface and performs initial decode
- Allocating queue entries
- Communicates ready/busy status back to the integer core when there are not enough empty shelves
- Receives To-Coprocessor data from the CPU
- Extracts exception information from the shelves and passes it onto the CP1 interface
- Communicates condition code checks and moves From-Coprocessor data to the CPU

Instructions to the FPU are initiated by the CP1 interface in order and enter the PIU. Upon entering the PIU, each instruction is decoded and allocated a buffer entry. Logic within the PIU keeps track of which buffer entries have been allocated and which ones are free, and also tracks the relative ‘age’ of each instruction in the buffer array (whether other instructions were before or after in program order).

Two instructions can be allocated each cycle. As instructions enter the queue and are subsequently retired, the Processor Interface Unit monitors the level of the buffer to determine if a stall must be issued to the CPU when the buffer level passes a predetermined threshold.

12.4.3 Issue Unit

There is one issue unit for arithmetic operations and one issue unit for load/store operations.

The arithmetic channel Issue Unit is responsible for determining when an arithmetic instruction can be issued based upon retrieving all the source data required for the instruction. This block is also responsible for doing any remaining decode required to issue the instruction and for stalling if the arithmetic block requests a stall. The arithmetic Issue Unit also maintains an issue pointer that points to the buffer entry of the next instruction to issue.

The data channel Issue Unit is responsible for determining when a load/store instruction can be issued based upon retrieving all the source data required for the instruction. This block is also responsible for doing any remaining decode required to issue the instruction and for stalling if the load/store block requests a stall. The data channel Issue Unit also maintains an issue pointer that points to the buffer entry of the next load/store instruction to issue.

12.4.4 Instruction Buffers

There are two instruction buffers, one for the arithmetic channel and one for the data channel. These buffers are defined as holding locations for instruction data for the lifespan of the instruction. There is one buffer entry per instruction. The corresponding Processor Interface Unit allocates buffer entries to incoming instructions. Subsequent downstream blocks update data in the buffer as the instruction proceeds down the pipeline until the instruction is finally retired in the retire unit and the queue entry is de-allocated.

The issue circuitry may retrieve (bypass) data from the queues (both sets), the register file, intercepting the data coming out of the arithmetic block, or intercepting the To-Coprocessor data coming from the integer core.

Note that buffer entries for a given channel are de-allocated in order. At no time does the FPU ever de-allocate entries in the buffer such that there are holes (non-contiguous entries). If an instruction is nullified or killed, that instruction will still remain active until it is retired in order and de-allocated. In this way, all pointers can always look ahead to the next buffer entry.

12.4.5 Arithmetic Processing Unit

The arithmetic processing unit resides in the arithmetic channel as shown in Figure 12.1. In this block the arithmetic instructions are executed and completed. There is independent issue control for the arithmetic channel and for the to-from (data) channel. An issue pointer within each channel's issue unit keeps track of the buffer entry with the next instruction to issue. When an instruction is issued in that channel the issue pointer is incremented.

12.4.6 Retirement Unit

There are independent retire units for the arithmetic channel and the data channel. Retirement in each channel occurs in order. In each retire unit there is a pointer that points to the next instruction in that channel to retire. Retire can happen out of order with respect to the other channel. An instruction is allowed to retire if it is the oldest outstanding instruction, or if there are no older instructions in the other channel that have not yet issued or that write to the same destination register.

In addition to checking for older conflicting instructions in the other channel, the instruction must be complete and ready to retire. When an instruction is eligible for retirement, the following events occur in each channel:

For the Arithmetic channel:

- If the instruction has completed successfully and was not nullified for any reason, the data is written to the register file.
- If the instruction was not canceled, then the FCSR Cause, Flag, and CC (condition code) bits are updated.
- If the instruction was canceled, then the FCSR Cause bits are updated.
- The associated buffer entry is de-allocated by turning its Valid bit off.

For the data channel:

- If the instruction has completed successfully and was not nullified for any reason, the data is written to the register file.
- If the instruction has successfully completed but the associated data has not yet been received, the instruction is moved to the *Scheduled Load Queue* to await the data.
- The associated buffer entry is de-allocated by turning its Valid bit off.

12.4.7 Scheduled Load Queue

The Scheduled Load Queue resides in the data channel and stores instructions waiting for To-Coprocessor data from the CP1 interface. This queue allows the To-Data instructions to retire even if they have not yet received their data. For example, if a load had a data cache miss.

Each Scheduled Load Queue entry contains enough information so that the data can be written to the register file once it arrives. The scheduled load queue entries are allocated from the retire unit. Upon allocation, the next empty entry in the queue is loaded with the contents of the instruction being allocated by the retire unit.

12.4.8 Arithmetic Channel Data Flow

The arithmetic pipe performs all arithmetic computations (e.g., add, multiply, conversions, divide, mov and compare). There are 4 classes of arithmetic instructions. Everything within a given class has the same rules for when an instruction can be issued and when an instruction stalls.

- Multiply and Multiply-add instructions
- Add and Convert instructions
- Divide and Square root instructions
- Quick path instructions (move, C.cond, etc.)

The inputs to the arithmetic pipeline are the opcode (heavily decoded) and the three 64 bit values; fr, fs, and ft. Part of the decoded opcode contains information such as data format and rounding mode. Additionally, each instruction will be issued with a buffer entry. The buffer entry indicates where all completed instruction data will be saved.

On the output side of the Arithmetic Unit there are independent stalls for each class of instruction indicating whether or not the arithmetic pipe can take an instruction of that class in a given clock cycle. Note that the Quick Path instructions never stall.

As shown above, there are four groups of arithmetic instructions. Each group has its own dedicated pipeline. However, because different instructions can have different latencies, each pipeline contains a different number of stages:

- 4-stage Multiply pipeline
- 4-stage Add pipeline
- 3-stage Divide and Square Root pipeline merged with a 3 stage buffer pipeline.
- 2-stage Quick Path / Early Trap module.

These pipelines handle all floating point instructions as described below:

12.4.8.1 Multiply

Multiply instructions are handled in the 4-stage Multiply pipeline. Floating point *Multiply* (MUL) instructions can be single or double precision.

12.4.8.2 Multiply-Add

Multiply-Add instructions require use of both the Multiply pipeline and Add pipeline to complete. The Multiply-Add instruction requires more clock cycles to complete than the standard MUL instruction. There are eight varieties of Multiply-Add instructions divided into four types; *Multiply-add*, *multiply-subtract*, *multiply-add-negate*, *multiply-subtract-negate*.

12.4.8.3 Add/Subtract/Data Conversion Instructions

The Add, Subtract, and Data Conversion instructions are handled in the 4-stage Add pipeline. Add instructions have a 4 clock execution time, assuming no delays.

12.4.8.4 Divide/Square Root Instructions

The Divide and Square Root instructions are handled in the 3-stage Divide and Square Root pipeline. This is the class of instructions that will iterate in order to derive the result. The control for these instructions can cycle through the pipeline multiple times. Results are fed back to previous pipeline stages for successive computations. A maximum of two divide and square root instructions may be executed concurrently. The Divide/Sqrt operation uses the multiple pipeline. The Divide and Square Root instructions include: *divide*, *square root*, *reciprical*, and *reciprical square root.*

12.4.8.5 Early Trap block / Quick Path

The Early Trap block is responsible for detecting early traps and unimplemented instructions. All instructions are issued to the Quick Path pipeline as well as the appropriate math pipelines listed above. For example, a Multiply instruction would go to both the Quick Path pipeline and the Multiply pipeline simultaneously.

The Quick Path pipeline analyzes the three inputs (fr,fs,ft) and classifies each one as Signaling Nan, infinity, power of 2, denormal, zero, or other (where ‘other’ is a normal uninteresting number). Based upon the input classification and the opcodes, the Quick Path pipeline will signal all early traps. Early traps are cases where the FPU does not need to do the full calculation to determine whether an instruction is exceptional. The no-exception/exception status is sent over the CP1 interface to the graduation logic in the core. The Quick Path pipeline stores the trap information to an entry in Instruction Buffer 1.

Compare and *Move* instructions are also executed in the quick path block. All other instructions are executed in the Quick Path pipeline if a computation is not required (e.g., there is an NaN input).

12.5 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.

12.5.1 Floating-Point Formats

The FPU provides the following two floating-point formats:

- A 32-bit single-precision floating point (type S)
- A 64-bit double-precision floating point (type D)

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

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

The single and double floating-point data types are composed of three fields—sign, exponent, fraction—whose sizes are listed in Table 12.1.

Table 12.1 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 12.1 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 12.2 and 12.3 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 12.2 Single-Precision Floating-Point Format (S)

31	30 23	22 0
S	Exponent	Fraction
1	8	23

Figure 12.3 Double-Precision Floating-Point Format (D)

63	62 52	51 0
S	Exponent	Fraction
1	11	52

Values are encoded in the specified format using the unbiased exponent, fraction, and sign values listed in Table 12.2. The high-order bit of the Fraction field, identified as b_1, is also important for NaNs.

Table 12.2 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 (*FCSR*$_{NAN2008}$ = 0)	`0x7fffffff`	`0x7fffffff ffffffff`
			0	QNaN	Quiet NaN (*FCSR*$_{NAN2008}$ = 0)	`0x7fbfffff`	`0x7ff7ffff ffffffff`
E_max + 1	≠ 0		0	SNaN	Signaling NaN (*FCSR*$_{NAN2008}$ = 1)	`0x7fbfffff`	`0x7ff7ffff ffffffff`
			1	QNaN	Quiet NaN (*FCSR*$_{NAN2008}$ = 1)	`0x7fffffff`	`0x7fffffff ffffffff`
E_max +1	0	1		$-\infty$	Minus infinity	`0xff800000`	`0xfff00000 00000000`
		0		$+\infty$	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`

Table 12.2 Value of Single or Double Floating-Point Data Type Encoding *(continued)*

Unbiased E	f	s	b_1	Value V	Type of Value	Typical Single Bit Pattern[1]	Typical Double Bit Pattern[1]
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.

12.5.1.1 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.

12.5.1.2 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 the table above) for plus infinity (+∞), minus infinity (–∞), quiet non-numbers (QNaN), and signaling non-numbers (SNaN).

12.5.1.3 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 12.8.2 “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 12.8.2.1 “Invalid Operation Exception”.

12.5.1.4 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, ABS.fmt, NEG.fmt`**) non-arithmetic; they do not signal IEEE 754 exceptions.

12.5.1.5 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 12.3 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 12.3 Value Supplied When a New Quiet NaN is Created

Format	New QNaN value ($FCSR_{NAN2008}$ = 0)	New QNaN value ($FCSR_{NAN2008}$ = 1)
Single floating point	`0x7FBF_FFFF`	`0x7FFF_FFFF`
Double floating point	`0x7FF7_FFFF_FFFF_FFFF`	`0x7FFF_FFFF_FFFF_FFFF`
Word fixed point	`0x7FFF_FFFF`	`0x7FFF_FFFF`
Longword fixed point	`0x7FFF_FFFF_FFFF_FFFF`	`0x7FFF_FFFF_FFFF_FFFF`

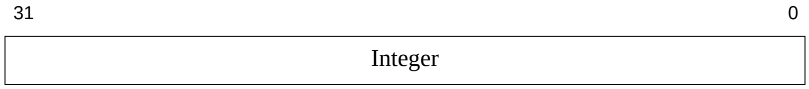
12.5.2 Fixed-Point Formats

The FPU provides two fixed-point data types:

- A 32-bit Word fixed point (type W), shown in Figure 12.4
- A 64-bit Longword fixed point (type L), shown in Figure 12.5

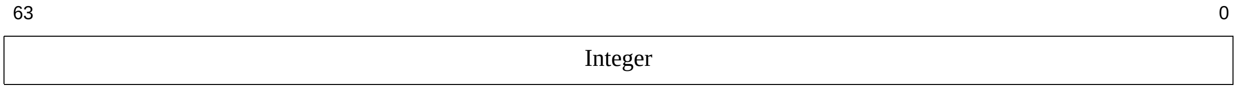
The fixed-point values are held in 2’s complement format, which is used for signed integers in the CPU. Unsigned fixed-point data types are not provided by the architecture; application software can synthesize computations for unsigned integers from the existing instructions and data types.

Figure 12.4 Word Fixed-Point Format (W)



[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.

Figure 12.5 Longword Fixed-Point Format (L)



12.6 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 thirty-two 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 thirty-two 32-bit registers with D-format values stored in even-odd pairs of registers; thus the register file can also be viewed as having sixteen 64-bit registers.

12.6.1 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 12.6 and 12.7 show the FPR organization and the way that operand data is stored in them.

Figure 12.6 Single Floating-Point or Word Fixed-Point Operand in an FPR

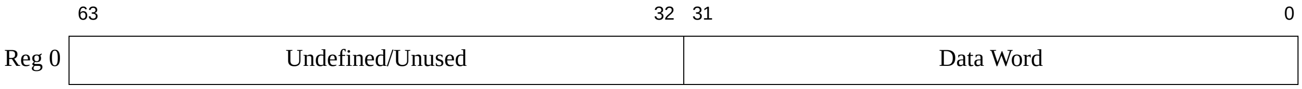
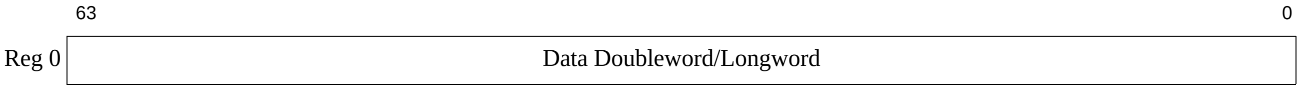


Figure 12.7 Double Floating-Point or Longword Fixed-Point Operand in an FPR



12.6.2 Formats of Values Used in Floating Point 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.

12.6.3 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 12.8 and Figure 12.9, 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 12.8 FPU Word Load and Move-to Operations

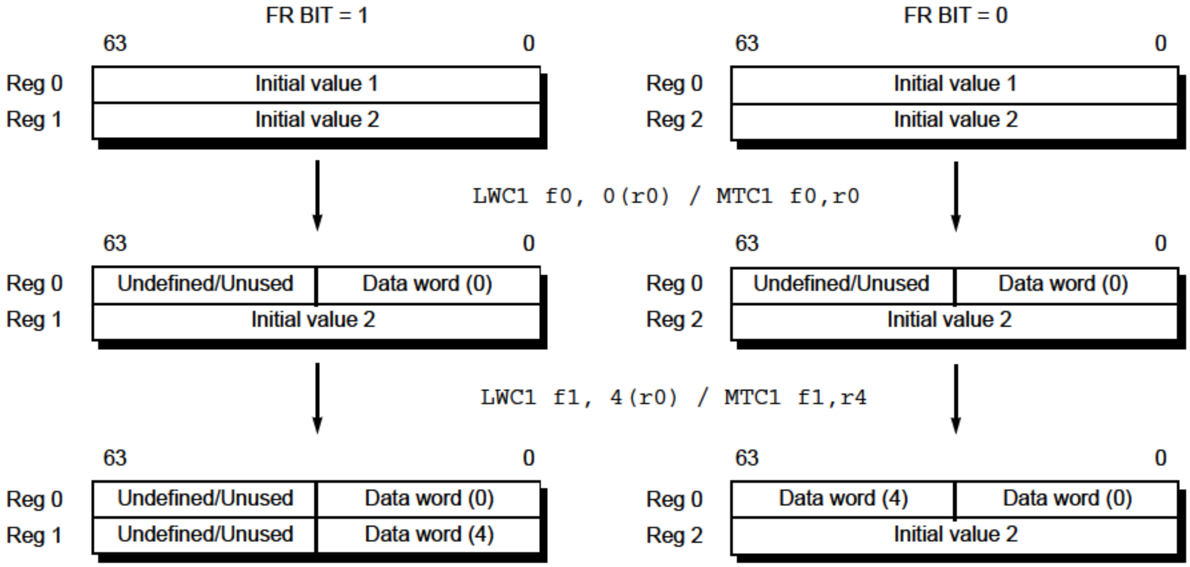
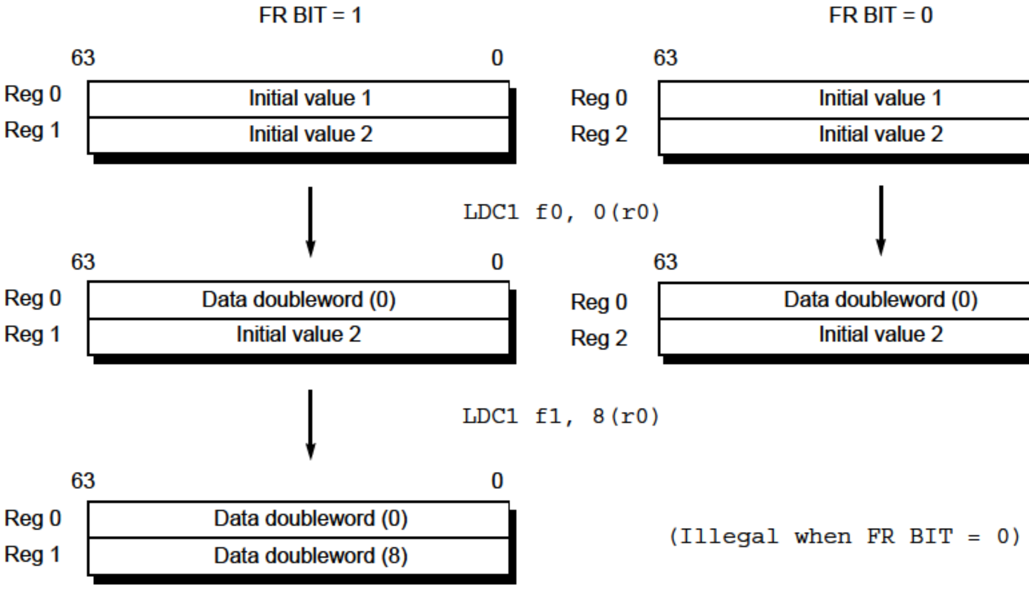


Figure 12.9 FPU Doubleword Load and Move-to Operations



12.7 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 12.4 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 12.4 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 12.5 defines the notation used for the read/write properties of the register bit fields.

Table 12.5 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.

12.7.1 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 12.10 shows the format of the *FIR*; Table 12.6 describes the *FIR* bit fields.

Figure 12.10 FIR Format

31 25	24	23	22	21	20	19	18	17	16	15 8	7 0
0	FC	Has2008	F64	L	W	3D	PS	D	S	ProcessorID	Revision

Table 12.6 FIR Bit Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:25	Reserved.	R	0

Table 12.6 FIR Bit Field Descriptions*(continued)*

Fields		Description	Read / Write	Reset State
Name	Bits			
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
Has2008	23	Indicates that one or more IEEE-754-2008 features are implemented. This bit is always set in proAptiv to indicate that the MAC2008, ABS2008, NAN2008 bits within the FCSR register exist. For more information, refer to Section 12.7.5 "Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)".	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
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

Table 12.6 FIR Bit Field Descriptions*(continued)*

Fields		Description	Read / Write	Reset State
Name	**Bits**			
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.	R	
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.	R	Hardwired

12.7.2 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 12.11 shows the format of the *FCCR*; Table 12.7 describes the *FCCR* bit fields.

Figure 12.11 FCCR Format

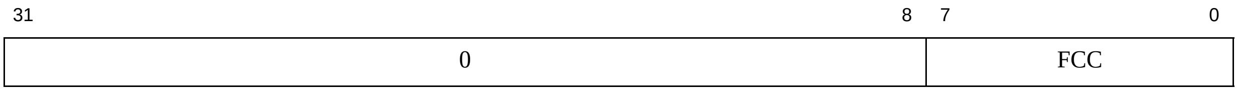


Table 12.7 FCCR Bit Field Descriptions

Fields		Description	Read / Write	Reset State
Name	**Bits**			
FCC	7:0	Floating-point condition code. Refer to the description of this field in Section 12.7.5 “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

12.7.3 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 12.12 shows the format of the *FEXR*; Table 12.8 describes the *FEXR* bit fields.

Figure 12.12 FEXR Format

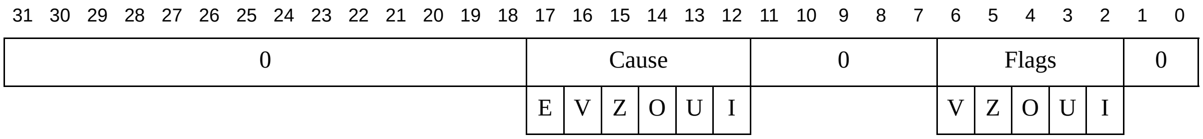


Table 12.8 FEXR Bit Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:18	These bits must be written as zeros; they return zeros on reads.	0	0
Cause	17:12	Cause bits. Refer to the description of this field in Section 12.7.5, "Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)".	R/W	Undefined
0	11:7	These bits must be written as zeros; they return zeros on reads.	0	0
Flags	6:2	Flag bits. Refer to the description of this field in Section 12.7.5, "Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)".	R/W	Undefined
0	1:0	These bits must be written as zeros; they return zeros on reads.	0	0

12.7.4 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 12.13 shows the format of the *FENR*; Table 12.9 describes the *FENR* bit fields.

Figure 12.13 FENR Format

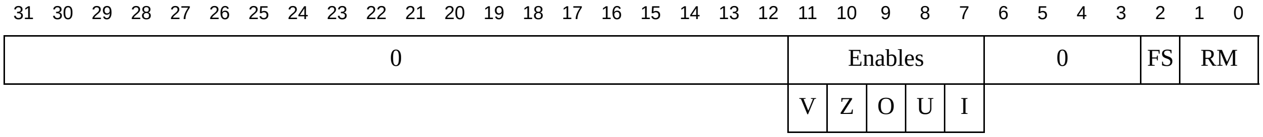


Table 12.9 FENR Bit Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:12	These bits must be written as zeros; they return zeros on reads.	0	0
Enables	11:7	Enable bits. Refer to the description of this field in Section 12.7.5, "Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)".	R/W	Undefined
0	6:3	These bits must be written as zeros; they return zeros on reads.	0	0
FS	2	Flush to Zero bit. Refer to the description of this field in Section 12.7.5, "Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)".	R/W	Undefined
RM	1:0	Rounding mode. Refer to the description of this field in Section 12.7.5, "Floating-Point Control and Status Register (FCSR, CP1 Control Register 31)".	R/W	Undefined

12.7.5 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 12.14 shows the format of the *FCSR*; Table 12.10 describes the *FCSR* bit fields.

Figure 12.14 FCSR Format

31						25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FCC							FS	FCC	0		MAC 2008	ABS 2008	NAN 2008	Cause						Enables					Flags					RM	
7	6	5	4	3	2	1		0	0	0	0	1	1	E	V	Z	O	U	I	V	Z	O	U	I	V	Z	O	U	I		

Table 12.10 FCSR Bit Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit			
FCC	31:25, 23	Floating-point condition codes. These bits record the result of floating-point compares and are tested for floating-point conditional branches and conditional moves. The FCC bit to use is specified in the compare, branch, or conditional move instruction. For backward compatibility with previous MIPS ISAs, the FCC bits are separated into two non-contiguous fields.	R/W	Undefined
FS	24	Flush to Zero (FS). The FS bit controls the handling of denormalized operands and is encoded as follows: 0: IEEE-compliant mode. Low performance on denormal operands and tiny results. 1: Regular embedded applications. High performance on denormal operands and tiny results. Refer to Section 12.7.6 “Operation of the FS Bit” for more details on this bit.	R/W	Undefined

Table 12.10 FCSR Bit Field Descriptions*(continued)*

Fields		Description	Read / Write	Reset State
Name	Bit			
0	22:21	These bits must be written as zeros; they return zeros on reads.	0	0
MAC2008	20	Fused multiply-add mode compliant with IEEE Standard 754-2008.The fused multiply-add operation multiplies and adds as if with unbounded range and precision, rounding only once to the destination format. The fused multiply-add is not supported in the proAptiv core. proAptiv implements the unfused multiply-add, which rounds the intermediary multiplication result to the destination format. This field applies to the MADD fmt, NMADD fmt, MSUB fmt and NMSUB fmt instructions. 0: Unfused multiply-add 1: IEEE 754-2008 fused multiply-add	RO	0
ABS2008	19	ABS fmt & NEG fmt instructions compliant with IEEE Standard 754-2008. The IEEE 754-2008 standard requires that the ABS and NEG functions accept QNAN inputs without trapping. This bit is always set in the proAptiv core to indicate support for the IEEE 754-2008 standard. 0: ABS & NEG trap for QNAN input 1: ABS & NEG accept QNAN input without trapping. IEEE 754-2008 behavior.	RO	1
NAN2008	18	Quiet and signaling NaN encodings recommended by the IEEE Standard 754-2008, i.e. a quiet NaN is encoded with the first bit of the fraction being 1 and a signaling NaN is encoded with the first bit of the fraction field being 0. In the proAptiv core, this bit is always set to indicate support for the IEEE Standard 754-2008 encoding. 0: MIPS NaN encoding 1: IEEE 754-2008 NaN encoding	RO	1
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 12.11 for the meaning of each cause bit.	R/W	Undefined

Table 12.10 FCSR Bit Field Descriptions*(continued)*

Fields		Description	Read / Write	Reset State
Name	Bit			
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 12.11 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 12.11 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 12.12 for the encoding of this field.	R/W	Undefined

Table 12.11 Cause, Enables, and Flags Definitions

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

Table 12.12 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.

12.7.6 Operation of the FS Bit

The FS bit in the CP1 *FCSR* register controls 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 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 significantly improved performance in the presence of denormal values.

Use of the FS bit affects handling of denormalized floating-point numbers and tiny results for the instructions listed below:

Table 12.13 Handling Denormalized Floating-point Numbers

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

[1] For **ABS, C.cond**, and **NEG**, denormal input operands or tiny results does 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.

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 positive normalized number (MinPosNorm) depending on the rounding mode settings. Table 12.14 lists the flushing behavior for tiny results..

Table 12.14 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	+MinPosNorm

Table 12.14 Zero Flushing for Tiny Results

Rounding Mode	Negative Tiny Result	Positive Tiny Result
RM (RM = 3)	-MinPosNorm	+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 12.15.

Table 12.15 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.

12.7.7 FCSR Cause Bit Update Flow

12.7.7.1 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.

12.7.7.2 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.

12.7.7.3 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.

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 12.7.7.2 "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.

12.7.7.4 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.

12.7.7.5 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.

12.8 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.

12.8.1 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 12.16). 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*.

12.8.2 Exception Conditions

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

- Section 12.8.2.1 “Invalid Operation Exception”
- Section 12.8.2.2 “Division By Zero Exception”
- Section 12.8.2.3 “Underflow Exception”
- Section 12.8.2.4 “Overflow Exception”
- Section 12.8.2.5 “Inexact Exception”
- Section 12.8.2.6 “Unimplemented Operation Exception”

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 12.16 summarizes the default results.

Table 12.16 Result for Exceptions Not Trapped

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

Table 12.16 Result for Exceptions Not Trapped*(continued)*

Bit	Description	Default Action
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.

12.8.2.1 Invalid Operation Exception

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

The following operations are invalid:

- One or both operands are a signaling NaN (except for the non-arithmetic MOV.fmt, MOVT fmt, MOVF fmt, MOVN fmt, and MOVZ.fmt instructions).
- Addition or subtraction: magnitude subtraction of infinities, such as (+∞) + (–∞) or (–∞) - (–∞).
- Multiplication: 0 × ∞, 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.

12.8.2.2 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.

12.8.2.3 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 archi-

tecture specifies that loss of accuracy be detected as inexact result, when the delivered result differs from what would have been computed if both the exponent range and precision were unbounded.

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

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

12.8.2.4 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.

12.8.2.5 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.

12.8.2.6 Unimplemented Operation Exception

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

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 handled by the FS bit.

12.9 Latency and Repeat Rates

Table 12.17 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 12.17 FPU Latency and Repeat Rates

Instruction[1]	Latency (core cycles)	Repeat Interval (core cycles)
ADD.[S,D], SUB.[S,D], MUL.[S,D]	4	1
MADD.[S,D], MSUB.[S,D], NMADD.[S.D], NMSUB.[S,D]	7	1
CVT.S.D, CVT.D.S, CVT.[S,D].[W,L], 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]	4	1
ABS.[S,D], NEG.[S,D], C.cond.[S,D], CABS.cond.[S,D]	2	1
MOV.[S,D], MOVF.[S,D], MOVN.[S,D], MOVT.[S,D], MOVZ.[S,D]	2	1
RECIP.S	11	4
RECIP.D	17	5
RSQRT.S	14	4
RSQRT.D	23	10
DIV.S[2]	12-16 (avg. 12.25)	9
DIV.D[2]	18-22 (avg. 18.25)	15
SQRT.S[2]	14-18 (avg. 14.25)	11
SQRT.D[2]	23 - 27 (avg. 23.25)	20
MTC1, DMTC1, LWC1, LDC1,LDXC1, LUXC1, LWXC1	4	1
MFC1, DMFC1, SWC1, SDC1, SDXC1, SUXC1, SWXC1	1	1

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

[2] Round to the nearest mode and no special operands

12.10 FPU Performance Counters

The proAptiv architecture contains a wide variety of performance counters as shown in Table 12.18. The event type is encoded into the *Event* field (bits 11:5) of the CP0 *Performance Counter Control 0 - 3* registers (CP0 register 25, Select 0, 2, 4, and 6). Refer to the CP0 chapter for more information on these registers.

Table 12.18 FPU Performance Counter Events and Codes

Event Number	Counter 0/2	Counter 1/3
112	Counts the number of cycles that the arithmetic channel is full and signalling busy to the integer core.	Counts the number of cycles the to-from channel is full and signalling busy to the integer core
113	Counts the total number of arithmetic instructions issued.	Counts the total number of to-from instructions issued.
114	Counts the total number of arithmetic add/multiply class instructions (add, sub, cvt, ceil, floor, round, trunc, mul).	Counts the total number of arithmetic multiply-add instructions (madd, msub, nmadd, nmsub).
115	Counts the total number of arithmetic iteration class instructions (div, recip, sqrt, rsqrt).	Counts the total number of arithmetic compare class instructions (C.cond)
116	Counts the total number of arithmetic miscellaneous class instructions (abs, neg, move, bra).	Counts the total number of data stall retires due to an scheduled load queue preempt write.
117	Counts the total number arithmetic channel conflict stalls. This retire stall occurs if there is an older instruction in the other channel with the same destination register, or if there is an older, unissued, instruction in the other channel.	Counts the total number of to-from channel conflict stalls. Same as arithmetic condition.
118	Counts the total number of arithmetic channel kill received stalls. This retire stall occurs if the instruction has not received a kill strobe.	Counts the total number of data channel kill received stalls. Same as arithmetic condition.
119	Counts the total number of arithmetic channel result valid stalls. This retire stall occurs if the result is not yet available from the APU.	Counts the total number of data channel retire stalls due to no room in the Scheduled Load Queue.
120	Counts the total number of arithmetic channel instruction issue stalls. This retire stall occurs if the instruction has not yet been issued.	Counts the total number of data channel instruction issue stalls. Same as arithmetic condition.
121	Counts the total number of arithmetic channel retire stall cycles. This is the sum of all of the retire stall conditions on counters 0/2 as described in events 117 - 120.	Counts the total number of data channel retire stall cycles. This is the sum of all of the retire stall conditions on counters 1/3 as described in events 116 - 120.
122	Counts the total number of arithmetic channel indeterminate dependency or format mismatch stalls. A stall occurs when: a. The youngest instruction is unknown due to taking an additional clock cycle to determine which of many instructions is the youngest. b. The youngest instruction is unknown because it has not yet received a null strobe. c. The youngest dependent instruction has a format mismatch that precludes the bypassing of data.	Counts the total number of data channel indeterminate dependency or format mismatch stalls. Stall conditions for the data channel are the same as for arithmetic channel.

Table 12.18 FPU Performance Counter Events and Codes *(continued)*

Event Number	Counter 0/2	Counter 1/3
123	Counts the total number of arithmetic channel APU stalls. This type of stall occurs when the APU is unable to take this class of instruction.	Reserved.
124	Counts the total number of arithmetic channel arithmetic data stalls. This type of stall occurs when the arithmetic unit is waiting for data from an arithmetic instruction.	Counts the total number of data channel arithmetic data stalls. Same as arithmetic condition.
125	Counts the total number of arithmetic channel to-data stalls. This type of stall occurs when the arithmetic unit is waiting for data from a to-from instruction.	Counts the total number of data channel to-data stalls. Same as arithmetic condition.
126	Counts the number of arithmetic channel stalls due to execution of a CTC1 or CFC1 instruction. Either the instruction is a CTC1/CFC1 and is stalling while waiting for all older instructions to retire, or an instruction is stalled while waiting for an older CTC1/CFC1 instruction to retire.	Counts the number of data channel stalls due to execution of a CTC1 or CFC1 instruction. Either the instruction is a CTC1/CFC1 and is stalling while waiting for all older instructions to retire, or an instruction is stalled while waiting for an older CTC1/CFC1 instruction to retire.
127	Counts the number of all arithmetic channel issue stall cycles. This is the sum of all of the above arithmetic channel issue stall conditions on counters 0/2 as described in events 122 - 126.	Counts the number of all data channel issue stall cycles. This is the sum of all of the above data channel issue stall conditions on counters 1/3 as described in events 122, 124 - 126.

12.11 Instruction Overview

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

- Section 12.11.1 “Data Transfer Instructions”
- Section 12.11.2 “Arithmetic Instructions”
- Section 12.11.3 “Conversion Instructions”
- Section 12.11.4 “Formatted Operand-Value Move Instructions”
- Section 12.11.5 “Conditional Branch Instructions”
- Section 12.11.6 “Miscellaneous Instructions”

12.11.1 Data Transfer Instructions

The FPU has two separate register sets: floating point coprocessor general registers (FPRs) and floating point 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 12.19 lists the supported transfer operations.

Table 12.19 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

12.11.1.1 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.

12.11.1.2 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 12.20 through 12.22 list the FPU data transfer instructions.

Table 12.20 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 12.21 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 12.22 FPU Move To and From Instructions

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

12.11.2 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).

Instructions involving multiply, divide, or square root usually take the Umimplemented Operation exception for any denormal operand or result. The FS, FO, and FN bits in the CP1 *FCSR* register can override this behavior as described in Section 12.7.6, "Operation of the FS Bit".

Table 12.23 lists the FPU IEEE compliant arithmetic operations.

Table 12.23 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 12.24 lists the FPU-approximate arithmetic operations.

Table 12.24 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 12.25. 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, or subtract instructions.

Table 12.25 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

12.11.3 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 12.7.6, "Operation of the FS Bit".

Table 12.26 and Table 12.27 list the FPU conversion instructions according to their rounding mode.

Table 12.26 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 12.27 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

12.11.4 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 12.28 through Table 12.30 list the formatted operand-value move instructions.

Table 12.28 FPU Formatted Operand Move Instructions

Mnemonic	Instruction
MOV fmt	Floating-Point Move

Table 12.29 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 12.30 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

12.11.5 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.

Table 12.31 lists the conditional branch (branch and branch likely) FPU instructions; Table 12.32 lists the deprecated conditional branch likely instructions.

Table 12.31 FPU Conditional Branch Instructions

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

Table 12.32 Deprecated FPU Conditional Branch Likely Instructions

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

12.11.6 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 12.33 lists these conditional move instructions.

Table 12.33 CPU Conditional Move on FPU True/False Instructions

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

12.12 Alphabetical Listing of Floating Point Instructions

Table 12.34 shows an alphabetical listing of the floating point unit instruction set, along with the associated instruction group, the page number location of the actual instruction. The actual instruction can be viewed by clicking on either the instruction of the page number reference in the table below. For the definition of each instruction, refer to Table 12.20 through Table 12.33 above.

Table 12.34 Alphabetical Listing of FPU Instructions

Instruction Name	Instruction Group
ABS.fmt	Move
ADD.fmt	Arithmetic
BC1F	Conditional Branch
BC1FL	Conditional Branch
BC1T	Conditional Branch
BC1TL	Conditional Branch
C.cond.fmt	Arithmetic
CEIL.L fmt	Conversion
CEIL.W fmt	Conversion
CFC1	Move
CTC1	Move
CVT.D fmt	Conversion
CVT.L fmt	Conversion
CVT.S fmt	Conversion
CVT.W.fmt	Conversion
DIV fmt	Arithmetic
FLOOR.L.fmt	Conversion
FLOOR.W.fmt	Conversion
LDC1	Load/Store
LDXC1	Load/Store
LUXC1	Load/Store
LWC1	Load/Store
LWXC1	Load/Store
MADD.fmt	Multiply-Accumulate
MFC1	Move
MFHC1	Move
MOV fmt	Move
MOVF.fmt	Move
MOVN.fmt	Move
MOVT fmt	Move
MOVZ fmt	Move
MSUB.fmt	Multiply-Accumulate

Table 12.34 Alphabetical Listing of FPU Instructions*(continued)*

Instruction Name	Instruction Group
MTC1	Move
MUL fmt	Arithmetic
NEG.fmt	Move
NMADD fmt	Multiply-Accumulate
NMSUB fmt	Multiply-Accumulate
RECIP fmt	Arithmetic
ROUND.L fmt	Conversion
ROUND.W fmt	Conversion
RSQRT fmt	Arithmetic
SDC1	Load/Store
SDXC1	Load/Store
SQRT fmt	Arithmetic
SUB.fmt	Arithmetic
SUXC1	Load/Store
SWC1	Load/Store
SWXC1	Load/Store
TRUNC.L fmt	Conversion
TRUNC.W fmt	Conversion

Chapter 13

MIPS DSP-R2 Application Specific Extension

The proAptiv core includes support for the MIPS DSP 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. The DSP instruction set is a collection of special-case instructions, in many cases aimed at the known ‘hot-spots’ of important algorithms that are common in DSP applications. The DSP Revision 2 (DSP-R2) is a superset of DSP Revision 1 and includes all of the instructions in revision 1.

This chapter contains the following sections:

13.1 MIPS32® DSP-R2 ASE Features

The DPS2 ASE contains features that support multiple DSP-R2 functions as described below:

- Q31 and Q15 (signed 16-bit) fractions
- Saturating arithmetic
- Multiplying fractions
- Rounding
- Multiply-accumulate sequences

- Single Instruction Multiple Data (SIMD) operations

13.1.1 Q31 and Q15 Signed 16-bit Fractions

DSP-R2 applications use fixed-point fractional data types. Such a fraction is a signed integer that represents an integer divided by some power of two. A 32-bit fractional format where the implicit divisor is 2^{16} (65536) would be referred to as a Q15.16 format; that's because there are 16 bits devoted to fractional precision and 15 bits to the whole number range (the highest bit does duty as a sign bit and isn't counted).

Using this notation, Q31.0 is a conventional signed integer, and Q0.31 is a fraction representing numbers between -1 and 1. The Q0.31 notation is the most popular 32-bit format for DSP-R2 applications since it won't overflow when multiplied. Q0.31 is often abbreviated to Q31.

13.1.2 Saturating Arithmetic

The DSP-R2 instruction set in the proAptiv core provides instruction that perform both saturated and non-saturated arithmetic. When a calculation overflows, the saturating (**`_SA`**) instructions make the result the most positive or most negative representable value.

13.1.3 Multiplying Fractions

Multiplying two Q31 fractions by re-using a full-precision integer multiplier results in a 64-bit result that consists of a Q62 result with (in the very highest bit) a second copy of the sign bit. A left-shift-by-1 must then be performed on this value to produce a Q63 format. Similarly, Q15 multiplies that generate a Q31 value must also perform a shift-left. This is the function of the **`MULQ`** instructions.

13.1.4 Rounding

Some fractional operations implicitly discard the least-significant bits. To get a better approximation, increment the truncated result by one when the discarded bits represent more than a half of the value of a 1 in the new LS position. That is how the term '*rounding*' is defined in this chapter.

13.1.5 Multiply-Accumulate Sequences

For enhanced performance performing fractional and saturating operations, the proAptiv core contains four accumulators for multiply-accumulate sequences (with fixed-point types, sometimes saturating). For backward compatibility with previous generation MIPS processors that have only one accumulator, the new *ac0* accumulator functions as the previous generation *HI*/*LO*.

13.1.6 SIMD Operations

Many DSP-R2 calculations are a good match for Single Instruction Multiple Data (SIMD) or *vector* operations, where the same arithmetic operation is applied in parallel to several sets of operands.

In the MIPS DSP-R2 ASE, some operations are SIMD type - two 16-bit operations or four 8-bit operations are carried out in parallel on operands packed into a single 32-bit general-purpose register. Instructions operating on vectors can be recognized because the name includes **`.PH`** (paired-half, usually signed, often fractional) or **`.QB`** (quad-byte, always unsigned, only occasionally fractional).

13.2 Common Applications

Different target applications generally need different data size and precision. The DSP-R2 ASE can be used by the following applications.

- *32-bit data*: audio (non-hand-held) decoding/encoding - a wide range of “hi-fi” standards for consumer audio or television sound.
- *16-bit data*: digital voice for telephony. International telephony code/decode standards include G.723.1 (8Ksample/s, 5-6Kbit/s data rate, 37ms delay), G.729 (8Kbit/s, 15ms delay) and G.726 (16-40Kbit/s, computationally simpler and higher quality, good for carrying analogue modem tones). Application-specific filters are used for echo cancellation, noise cancellation, and channel equalization. Also used for soft modems and general ‘DSP’ work such as filters, correlation, and convolution.
- *8-bit data*: processing of printer images, JPEG (still) images and video data.

13.3 Software Detection of the DSP ASE Revision 2

The presence of the MIPS DSP-R2 ASE in the proAptiv core is indicated by two static bits in the *Config3* register: the ‘DSP Present’ bit (*Config3*$_{DSPP}$*)* indicates the presence of the DSP-R2 ASE, and the ‘DSP Rev2 Present’ bit (*Config3*$_{DSP2P}$*)* indicates the presence of the MIPS DSP ASE Rev2. Because all members of the proAptiv family support both ASEs, these bits are always set to 1.

The CP0 Status register (*Status*$_{MX}$) must be set to enable access to the extra instructions defined by the DSP-R2 moduile, as well as to the **`MTLO/HI, MFLO/HI`** instructions that access accumulators ac1, ac2, and ac3. Executing a DSP-R2 ASE instruction or the **`MTLO/HI, MFLO/HI`** instructions with this bit set to zero causes a DSP-R2 State Disabled Exception (exception code 26 in the CP0 *Cause* register). This exception can be used by system software to do lazy context-switching.

13.4 DSP-R2 Registers

The DSP-R2 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-R2 ASE and to include these additional registers in the context save and restore operations.

13.4.1 DSP-R2 Accumulator Registers

Whereas a standard MIPS32 architecture CPU has just one 64-bit multiply unit accumulator (accessible as *hi*/*lo*), the DSP-R2 ASE in the proAptiv core provides four 64-bit accumulators. Instructions accessing the extra accumulators specify a 2-bit field as 0 - 3 (0 selects the original accumulator).

The DSP-R2 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-R2 instructions that target the accumulators use two instruction bits to specify the destination accumulator, with the zero value referring to the original accumulator.

13.4.2 DSP-R2 ASE Control Register

This is a part of the user-mode programming model for the DSP-R2 ASE, and is a 32-bit value read and written with the **`RDDSP`**/**`WRDSP`** instructions. It holds state information for some DSP-R2 sequences.

Figure 13.1 DSP-R2 Control Register

31 28	27 24	23 16	15	14	13	12 7	6	5 0
0	CCOND	OUFLAG	0	EFI	C	SCOUNT	0	POS

Table 13.1 Field Descriptions for DSP-R2 Control Register

Name	Bit(s)	Description	Read/ Write	Reset State
0	31:28	Reserved. Write as zero. Undefined on read.	R/W	Undefined
CCOND	27:24	Condition bits set by compare instructions (there have to be four to report on compares between vector types). "Compare" operations on scalars or vectors of length two only touch the lower-numbered bits.	R/W	0
OUFLAG	23:16	Overflow/underflow flags. One of these bits may be set when a result overflows (whether or not the result is saturated depends on the instruction - the flag is set in either case). Underflow indicates a value that is negative but with excessive absolute value. Any overflowed/underflowed result produced by any DSP-R2 ASE instruction sets an *ouflag* bit, *except* for **`ADDSC/ADDWC`** and **`SHILO/SHILOV`** instructions. See Table 13.2 for a full list of which bits are set by what instructions.	R/W	
0	15	Reserved. Write as zero. Undefined on read.	R/W	
EFI	14	Extract field of instruction. This bit is set by any of the accumulator-to-register bit field extract instructions (**`EXTP`**, **`EXTPV`**, **`EXTPDP`**, or **`EXTPDP`**) only if the instruction finds there are insufficient bits to extract. In other words, if $\mathit{DSPControl}_{POS}$, which is supposed to mark the highest-numbered bit of the field being extracting, is less than the size value specified by the instruction. Note that 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	
C	13	Carry bit for 32-bit add/carry instructions **`ADDSC`** and **`ADDWC`**. 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	
SCOUNT	12:7	Size count. This field specifies the size of the bit field to be inserted, while $\mathit{DSPControl}_{POS}$ specifies the insert position. This field is used by "variable" bit field insert and extract instructions such as **`INSV`** (the normal MIPS32 **`INS/EXT`** instructions have the field size and position hard-coded in the instruction).	R/W	
0	6	Reserved. Write as zero. Undefined on read.	R/W	

Table 13.1 Field Descriptions for DSP-R2 Control Register

Name	Bit(s)	Description	Read/ Write	Reset State
POS	5:0	Insertion position. This field is used by the variable insert instruction **INSV** and specifies the position of the bit field to be inserted, while *DSPControl*$_{SCOUNT}$ specifies the size of the bit field to be inserted. This field to specify the insert position. In most insertions (following the lead of the standard MIPS32 insert/ extract instructions) this field is set to the lowest bit number. However, in the DSP-R2 ASE extract-from-accumulator instructions (**EXTP**, **EXTPV**, **EXTPDP** and **EXTPDPV**), this field identifies the *highest*-numbered bit in the field. The **EXTPDP** and **EXTPDPV** instructions post-decrement this field (by the bit-field length *size*) to help software which is unpacking a series of bit fields from a dense data structure. The **MTHLIP** instruction increments the value in this field by 32 after copying the value of *lo* to *hi*.	R/W	

The bits of the overflow flag *OUFLAG* field in the *DSPControl* register are set by a number of instructions, as described in Table 13.2. These bits are sticky and can be reset only by an explicit write to these bits in the register (using the **WRDSP** instruction). Refer to the following section for more information on the DSP-R2 instructions.

Table 13.2 Instructions that set the 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**.

13.5 DSP-R2 Instruction Types

The DSP-R2 instruction set in the proAptiv core is a collection of special-case instructions aimed at addressing known ‘hot-spots’ of important DSP algorithms.

In this section, the DSP-R2 instructions have been divided into the following subsections, which represent their likely usage and application and type of the result derived. Most of the multiplication instructions have multiple uses and are divided based on the most obvious use.

- Arithmetic - 64-bit
- Arithmetic - saturating and/or SIMD Types
- Bit-shifts - saturating and/or SIMD types
- Comparison and "conditional-move" operations on SIMD types - includes **PICK** instructions.
- Conversions to and from SIMD types
- Multiplication - SIMD types with result in GP register
- Multiply Q15s from paired-half and accumulate
- Load with register+register address
- DSPControl register access
- Accumulator access instructions
- Dot products and building blocks for complex multiplication - includes full-word (Q31) multiply-accumulate
- Other DSP ASE instructions - everything else...

A complete alphabetical list of DSP-R2 instructions is shown in Section 13.11 “DSP-R2 ASE Instruction Groups”, followed by a description of each individual instruction.

13.5.1 Hints in instruction names

An instruction’s name may have some suffixes which are often informative:

Q: generally means it treats operands as fractions (which isn’t important for adds and subtracts, but is important for multiplications and convert operations);

_**S**: usually means the full-precision result is saturated to the size of the destination; _**SA** is used for instructions which saturate intermediate results before accumulating; and **R**: denotes rounding (see above);

.W, .PH, .QB: suggest the operation is dealing with 32-bit word, paired-halfword, or quad-byte values respectively. Where there are two of these (as in **MAQ_S.W.PHL**) the first one suggests the type of the result, and the second the type of the operand(s).

V: (in a shift instruction) suggests that the shift amount is defined in a register, rather than being encoded in a field of the instruction.

13.5.2 Arithmetic — 64-bit

ADDSC/**ADDWC** generate and use a carry bit, for efficient 64-bit add.

13.5.3 Arithmetic — Saturating and/or SIMD Types

- *32-bit signed saturating arithmetic*: **`ADDQ_S.W`**, **`SUBQ_S.W`** and **`ABSQ_S.W`**.
- *Paired-half and quad-byte SIMD arithmetic*: perform the same operation simultaneously on both 16-bit halves or all four 8-bit bytes of a 32-bit register. The “**`Q`**” in the instruction mnemonic for the PH operations here is cosmetic: Q15 and signed 16-bit integer add/subtract operations are bit-identical - Q15 only behaves very differently when converted or multiplied.

  The paired half operations are: **`ADDQ.PH`**/**`ADDQ_S.PH`**, **`SUBQ.PH`**/**`SUBQ_S.PH`** and **`ABSQ_S.PH`**.

  The quad-byte operations (all unsigned) are: **`ADDU.QB`**/**`ADDU_S.QB`**, **`SUBU.QB`**/**`SUBU_S.QB`**.
- *Sum of quad-byte vector*: **`RADDU.W.QB`** does an unsigned sum of the four bytes found in a register, zero extends the result and delivers it as a 32-bit value.

13.5.4 Bit-shifts — Saturating and/or SIMD Types

All shifts can either have a shift amount encoded in the instruction, or - indicated by a trailing “**`V`**” in the instruction name - provided as a register operand. **`PH`** and 32-bit shifts have optional forms which saturate the result.

- *32-bit signed shifts*: include a saturating version of shift left, **`SHLL_S.W`**; and an auto-rounded shift right (just the “arithmetic”, sign-propagating form): **`SHRA_R.W`**. Recall from above that rounding can be imagined as pre-adding a half to the least significant surviving bit.
- *Paired-half and quad-byte SIMD shifts*: **`SHLL.PH`**/**`SHLLV.PH`**/**`SHLL_S.PH`**/**`SHLLV_S`** are as above. For **`PH`** only there’s a shift-right-arithmetic instruction (“arithmetic” means it propagates the sign bit downward) **`SHRA.PH`**, which has a variant which rounds the result **`SHRA_R.PH`**.

  The quad-byte shifts are unsigned and don’t round or saturate: **`SHLL.QB`**/**`SHLLV.QB`**, **`SHRL.QB`**/**`SHRLV.QB`**.

13.5.5 Comparison and “Conditional-Move” Operations on SIMD Types

The “**`cmp`**” operations simultaneously compare and set flags for two or four values packed in a vector (with equality, less-than and less-than-or-equal tests). For PH that’s **`CMP.EQ.PH`**, **`CMP.LT.PH`** and **`CMP.LE.PH`**. The result is left in the two LS bits of *DSPControl[ccond]*.

For quad-byte values **`CMPU.EQ.QB`**, **`CMPU.LT.QB`** and **`CMPU.LE.QB`** simultaneously compare and set flags for four bytes in *DSPControl[ccond]* - the flag relating to the bytes found in the low-order bits of the source register is in the lowest-numbered bit (and so on). There’s an alternative set of instructions **`CMPGU.EQ.QB`**, **`CMPGU.LT.QB`** and **`CMPGU.LE.QB`** which leave the 4-bit result in a specified general-purpose register.

`PICK.PH` uses the two LS bits of *DSPControl[ccond]* (usually the outcome of a paired-half compare instruction, see above) to determine whether corresponding halves of the result should come from the first or second source register. Among other things, this can implement a paired-half conditional move. You can reverse the order of your conditional inputs to do a move dependent on the complementary condition, too.

`PICK.QB` does the same for QB types, this time using four bits of *DSPControl[ccond]*.

13.5.6 Conversions to and from SIMD Types

Conversion operations from larger to smaller fractional types have names which start "**`PRECRQ...`**" for "precision reduction, fractional". Conversion operations from smaller to larger have names which start "**`PRECE...`**" for "precision expansion".

- *Form vector from high/low parts of two other paired-half values*: **`PACKRL.PH`** makes a paired-half vector from two half vectors, swapping the position of each sub-vector. It can be used to acquire a properly formed sub-vector from a non-aligned data stream.
- *One Q15 from a paired-half to a Q31 value*: **`PRECEQ.W.PHL`**/**`PRECEQ.W.PHR`** select respectively the "left" (high bit numbered) or "right" (low bit numbered) Q15 value from a paired-half register, and load it into the result register as a Q31 (that is, it's put in the high 16 bits and the low 15 bits are zeroed).
- *Two bytes from a quad-byte to paired-half*: **`PRECEQU.PH.QBL`**/**`PRECEQU.PH.QBR`** picks two bytes from either the "left" (high bit numbered) or "right" (low bit numbered) halves of a quad-byte value, and unpacks to a pair of Q15 fractions.

  `PRECEQU.PH.QBLA` does the same, except that it picks two "alternate" bytes from bits 31-24 and 15-8, while **`PRECEQU.PH.QBRA`** picks bytes from bits 23-16 and 7-0.

  Similar instructions without the **`q`** - **`PRECEU.PH.QBL`**, **`PRECEU.PH.QBR`**, **`PRECEU.PH.QBLA`**" and **`PRECEU.PH.QBRA`** - work on the same register fields, but treat the quantities as integers, so the 16-bit results get their low bits set.
- *2*×Q31 to a paired-half: both operands and result are assumed to be signed fractions, so **`PRECRQ.PH.W`** just takes the high halves of the two source operands and packs them into a paired-half; **`PRECRQ_RS.PH.W`** rounds and saturates the results to Q15.
- *2*×paired-half to quad-byte: you need two source registers to provide four paired-half values, of course. This is a fractional operation, so it's the low bits of the 16-bit fractions which are discarded.

  `PRECRQ.QB.PH` treats the paired-half operands as unsigned fractions, retaining just the 8 high bits of each 16-bit component.

  `PRECRQU_S.QB.PH` treats the paired-half operands as Q15 signed fractions and both rounds and saturates the result (in particular, a negative Q15 fraction produces a zero byte, since zero is the lowest representable quantity).
- *Replicate immediate or register value to paired-half*: in **`REPL.PH`** the value to be replicated is a 10-bit signed immediate value (that's in the range `-512` ≤ x ≤ 511) which is sign-extended to 16 bits, whereas in **`REPLV.PH`** the value - assumed to be already a Q15 value - is in a register.
- *Replicate single value to quad-byte*: there's both a register-to-register form **`REPLV.QB`** and an immediate form **`REPL.QB`**.

13.5.7 Multiplication - SIMD Types with Result in GP Register

When a multiply's destination is a general-purpose register, the operation is still done in the multiply unit, and you should expect it to overwrite the *hi*/*lo* registers (otherwise known as *ac0*.)

- *8-bit*×16-bit 2-way SIMD multiplication: **`MULEU_S.PH.QBL`**/**`MULEU_S.PH.QBR`** picks the "left" (high bit numbered) or "right" (low bit numbered) pair of byte values from one source register and a pair of 16-bit values from the other. Two unsigned integer multiplications are done at once, the results unsigned-saturated and delivered to the two 16-bit halves of the destination.

The asymmetric use of the source operands is not a bit like a Q15 operation. But 8×16 multiplies are heavily used in imaging and video processing (JPEG image encode/decode, for example).

- *Paired-half SIMD multiplication*: **MULQ_RS.PH** multiplies two Q15s at once and delivers it to a paired-half value i n a general-purpose register, with rounding and saturation.
- *Multiply half-PH operands to a Q31 result*: **MULEQ_S.W.PHL**/**MULEQ_S.W.PHR** pick the “left”/”right” Q15 value respectively from each operand, multiply and store a Q31 value.

  “Precision-doubling” multiplications like this *can* overflow, but only in the extreme case where you multiply -1×-1, and can’t represent 1 exactly.

13.5.8 Multiply Q15s from Paired-Half and Accumulate

MAQ_S.W.PHL/**MAQ_S.W.PHR** picks either the left/high or right/low Q15 value from each operand, multiplies them to Q31 and accumulates to a Q32.31 result. The multiply is saturated only when it’s -1×-1.

MAQ_SA.W.PHL/**MAQ_SA.W.PHR** differ in that the final result is saturated to a Q31 value held in the low half of the accumulator (required by some ITU voice encoding standards).

13.5.9 Load with Register + Register Address

Previously available only for floating point data[1]: **LWX** for 32-bit loads, **LHX** for 16-bit loads (sign-extended) and **LBUX** for 8-bit loads, zero-extended.

13.5.10 DSP-R2 Control Register Access

WRDSP RS,MASK sets *DSPControl* fields, but only those fields which are enabled by a 1 bit in the 6-bit mask.

RDDSP reads *DSPControl* into a GPR; but again it takes a mask field. Bit fields in the GPR corresponding to *DSPControl* fields which are not enabled will be set all-zero.

The mask bits tie up with fields like this:

Table 13.3 Mask bits for instructions accessing the DSPControl register

Mask Bit	*DSPControl field*
0	pos
1	scount
2	c
3	ouflag
4	ccond
5	EFI

13.5.11 Accumulator Access Instructions

- *Historical instructions which now access new accumulators*: the familiar **MFHI**/**MFLO**/**MTHI**/**MTLO** instructions now take an optional extra accumulator-number parameter.

[1] Well, an integer instruction is also included in the MIPS SmartMIPS™ ASE.

- *Shift and move to general register*: **EXTR.W**/**EXTR_R.W**/**EXTR_RS.W** gets a 32-bit field from an accumulator (starting at bit 0 up to 31) and puts the value in a general purpose register. At your option you can specify rounding and signed 32-bit saturation.

  EXTRV.W/**EXTRV_R.W**/**EXTRV_RS.W** do the same but specify the field's starting bit number with a register.

- *Extract bit field from accumulator*: **EXTP**/**EXTPV** takes a bit field (up to 32 bits) from an accumulator and moves it to a GPR. The length of the field can be an immediate value or from a register. The position of the field is determined by *DSPControl[pos]*, which holds the bit number of the most significant bit.

  EXTPDP/**EXTPDPV** do the same, but also auto-decrement *DSPControl[pos]* to the bit-number just below the field you extracted.

- *Accumulator rearrangement*: **SHILO**/**SHILOV** has a *signed* shift value between -32 and +31, where positive numbers shift right, and negative ones shift left. The "v" version, as usual, takes the shift value from a register. The right shift is a "logical" type so the result is zero extended.

- *Fill accumulator pushing low half to high*: **MTHLIP** moves the low half of the accumulator to the high half, then writes the GPR value in the low half. Generally used to bring 32 more bits from a bit stream into the accumulator for parsing by the various **EXT**... instructions.

13.5.12 Dot Products and Building Blocks for Complex Multiplication

In 2-dimensional vector math (or in any doubled-up step of a multiply-accumulate sequence which has been optimized for 2-way SIMD) you're often interested in the *dot product* of two vectors:

```
v[0]*w[0] + v[1]*w[1]
```

In many cases you take the dot product of a series of vectors and add it up, too.

Some algorithms use complex numbers, represented by 2D vectors. Complex numbers use `i` to stand for "the square root of -1", and a vector `[a,b]` is interpreted as `a+ib` (mathematicians leave out the multiply sign and use single-letter variables, habits which would not be appreciated in C programming!) Complex multiplication just follows the rules of multiplying out sums, remembering that `i*i=-1`, so:

```
(a + ib)*(c + id) = (a*c - b*d) + i(a*d + b*c)
```

Or in vector format:

```
[a, b] * [c, d] = [a*c - b*d, a*d + b*c]
```

The first element of the result (the "real component") is like a dot product but with a subtraction, and the second (the "imaginary component") is like a dot product but with the vectors crossed.

- *Q15 dot product from paired-half, and accumulate*: **DPAQ_S.W.PH** does a SIMD multiply of the Q15 halves of the operands, then adds the results and saturates to form a Q31 fraction, which is accumulated into a Q32.31 fraction in the accumulator.

  DPSQ_S.W.PH does the same but subtracts the dot product from the accumulator.

  For the imaginary component of a complex multiply, first swap the Q15 numbers in one of the register operands with a **ROT** (bit-rotate) instruction.

For the real component of a complex Q15 multiply, you have the difference-of-products instruction **MULSAQ_S.W.PH**, which parallel-multiplies both Q15 halves of the PH operands, then computes the difference of the two results and leaves it in an accumulator in Q32.31 format (beware: this does not accumulate the result).

- *16-bit integer dot-product from paired-half, and accumulate*: **DPAU.H.QBL**/**DPAU.H.QBR** picks two QB values from each source register, parallel-multiplies the corresponding pairs to integer 16-bit values, adds them together and then adds the whole lot into an accumulator. **DPSU.H.QBL**/**DPSU.H.QBR** do the same sum-of-products, but the result is then subtracted from the accumulator. In both cases, note this is integer (not fractional) arithmetic.
- *Q31 saturated multiply-accumulate*: is the nearest thing you can get to a dot-product for Q31 values. **DPAQ_SA.L.W** does a Q31 multiplication and saturates to produce a Q63 result, which is added to the accumulator and saturated again. **DPSQ_SA.L.W** does the same, except that the multiply result is subtracted from the accumulator (again, useful for the real component of a complex number).

13.5.13 Other DSP-R2 ASE Instructions

- *Branch on DSPControl field*: **BPOSGE32** branches if *DSPControl[pos]*≥32.

  Typically the test is for “is it time to load another 32 bits of data from the bit stream yet?”.
- *Circular buffer index update*: **MODSUB** takes an operand which packs both a maximum index value and an index step, and uses it to decrement a “buffer index” by the step value, but arranging to step from zero to the provided maximum.
- *Bit field insert with variable size/position*: **INSV** is a bit-insert instruction. It acts like the MIPS32 standard instruction **INS** except that the position and size of the inserted field are specified not as immediates inside the instruction, but are obtained from *DSPControl[pos]* (which should be set to the lowest numbered bit of the field you want) and *DSPControl[scount]* respectively.
- *Bit-order reversal*: **BITREV** reverses the bits in the low 16 bits of the register. The high half of the destination is zero.

  The bit-reverse operation is a computationally crucial step in buffer management for FFT algorithms, and a 16-bit operation supports up to a 32K-point FFT, which is much more than enough. A full 32-bit reversal would be expensive and slow.

13.6 DSP-R2 Code Optimization for Performance

Some optimization methods are seldom used for general-purpose software, but are often seen in DSP-R2 code. A typical example is a technique called zipping, which reduces the number of data loads in algorithms like FIR filters. Consider the calculation of the first two output values of an 8-tap FIR filter. The illustration below shows how the coefficients get multiplied by the data samples:

```
 Input data samples:       d0  d1  d2  d3  d4  d5  d6  d7  d8  ...
Coefficients for y0:       c0  c1  c2  c3  c4  c5  c6  c7
Coefficients for y1:           c0  c1  c2  c3  c4  c5  c6  c7

 First output (y0):     c0d0 + c1d1 + c2d2 + c3d3 + c4d4 + c5d5 + c6d6 + c7d7
Second output (y1):     c0d1 + c1d2 + c2d3 + c3d4 + c4d5 + c5d6 + c6d7 + c7d8
```

A very naive implementation will load each coefficient and data sample from memory every time they are needed. A more optimized implementation will load the coefficients just once and keep them in registers. It will load data samples d0-d7 first, to compute the first output, and then data samples d1-d8 to compute the second output. With zipping, an even more optimized implementation will load d0-d8 once and use the loaded values for both output calculations. The relatively large number of general-purpose registers in the MIPS architecture is useful for applying this technique. An even larger number of samples can be kept in registers if the SIMD features of the DSP-R2 ASE are used. In this case, another slightly rearranged set of coefficients may be needed, as illustrated below for the case of 16-bit coefficients and samples packed into 32-bit words:

```
    Input data samples:      d0:d1  d2:d3  d4:d5  d6:d7  d8:d9
   Coefficients for y0:      c0:c1  c2:c3  c4:c5  c6:c7
   Coefficients for y1:      00:c0  c1:c2  c3:c4  c5:c6  c7:00
```

The first set of coefficients is used to compute y0 and all even-numbered output samples. The second set is used for y1 and all odd-numbered output samples.

Because of the large number of general-purpose registers, especially considering the SIMD features offered by the DSP-R2 ASE, the proAptiv cores lend themselves well to algorithms processing more data elements at a time. For example, it is easy to meet the requirements of a radix-4 FFT implementation, which is faster than a radix-2 implementation but needs to keep a large number of values in registers during the calculation.

Many algorithms can be implemented in a variety of different ways and often some of these algorithm transformations offer performance advantages. The output results are similar in all cases, but an optimal implementation strikes the best balance between number of registers needed, number of memory operations, number and type of arithmetic operations, regularity of data access patterns, result delays, etc. Make sure the selected algorithm implementation approach is the best match to the architecture.

It should be noted that a reasonable degree of familiarity with the DSP-R2 instructions will allow the programmer to extract the best performance. The architecture specification and the core programming guide provide the necessary information.

If the data type is 16-bit or 8-bit, then attempting to rewrite the algorithm in SIMD style using operations directly supported by the DSP-R2 ASE instruction set will yield a lower cycle count due to the obtained parallelism. Once the number of instructions required to implement the algorithm is minimized, the instructions must be scheduled taking into account their result delays. When evaluating the resulting performance, obtaining a trace from one execution will illuminate the cause of stalls and also facilitate optimization.

13.6.1 Pixel Unpacking Example

As a simple illustration of efficiently using the SIMD capabilities of the processor, consider the task of unpacking YUV image data prior to processing. The YUV color space is commonly used in image and video processing. The color components (U and V) are subsampled with respect to the luminosity (Y) by a factor of two or four in the horizontal and/or vertical dimension. A commonly used format is YUV 4:2:2, which has the color components subsampled horizontally by a factor of two. Hence there is one luminosity value for each pixel, but a UV pair is shared between two adjacent pixels. Video data in YUV 4:2:2 format is commonly transmitted in the following order:

```
      Pixel data:     U0  Y0  V0  Y1  U2  Y2  V2  Y3  U4  Y4  V4  Y5  ...
```

Video processing algorithms usually perform different tasks on each of the Y, U, and V components. In order to use the SIMD capabilities of the proAptiv core, the pixel data needs to be unpacked, i.e., each of the Y, U, and V values should be separated out and grouped together. Examining the DSP-R2 ASE instruction set reveals that some instructions intended for precision reduction and expansion can be used to implement data unpacking:

```
lw               $t0, 0($a0)      # U0:Y0:V0:Y1 - two pixels in YUV 4:2:2
lw               $t1, 4($a0)      # U2:Y2:V2:Y3 - next two pixels

precequ.ph.qbra  $t2, $t0         # 00:Y0:00:Y1 - half the unpacked Ys
precrq.qb.ph     $t4, $t0, $t1    # U0:V0:U2:V2 - interleaved Us and Vs
precequ.ph.qbra  $t3, $t1         # 00:Y2:00:Y3 - the other unpacked Ys
precequ.ph.qbra  $t5, $t4         # 00:V0:00:V2 - unpacked Vs
precequ.ph.qbla  $t5, $t4         # 00:U0:00:U2 - unpacked Us
```

Unpacking the YUV data as illustrated above also has the advantage of converting each data item from 8-bit unsigned to 16-bit unsigned format. This ensures there is enough room for performing the video processing calculations with enhanced precision.

Note that the example above as well as the following examples have not been scheduled to the proAptiv core pipelines. In a real application other instructions surrounding the illustrated code fragments can be used to fill-in the delays caused by result dependencies.

13.6.2 Sum of Absolute Differences Example

Another interesting DSP-R2 ASE example shows the kernel performing the sum of absolute differences (SAD) function used in motion estimation algorithms for video compression. The function accumulates the absolute difference between the pixels from a reference 8x8-pixel block and those from a similar block inside the current video frame. Using the DSP-R2 ASE, here is how the SAD of four pixels at a time can be calculated and accumulated:

```
SUBUH_R.QB       $t0, $s0, $s1       # subtract 4 pixels with halving
ABSQ_S.QB        $t0, $t0            # find the 4 absolute values
RADDU.W.QB       $t0, $t0            # sum the absolute values
ADDU             $v0, $v0, $t0       # accumulate the result
```

The above sequence is four instructions long and will execute in four cycles when properly scheduled. Performing the same calculation using the MIPS32 Release 2 instruction set will require approximately 20 instructions. The performance advantages offered by the DSP-R2 ASE are obvious.

13.6.3 Bit Stream Unpacking Example

The last example of efficient use of DSP-R2 ASE instructions presented here is bit stream unpacking. Many audio and video compression algorithms pack various parameters of different bit widths in a continuous compressed bitstream. The decoder has to first unpack-or extract-the individual values from the bit stream before further processing them. Recognizing the importance of this task, the DSP-R2 ASE instruction set includes instructions that facilitate and accelerate bit stream unpacking. One of the accumulator registers is used as a 64-bit data buffer. The EXTP instruction variants extract a specified number of bits and optionally decrement the pos field of the DSPControl register. The pos field holds the number of remaining bits in the bit buffer. The BPOSGE32 instructions checks this number and branches if there are at least 32 bits left. And finally, the MTHLIP instruction is used to reload the bit buffer with a new 32-bit word and at the same time increment the number of available bits by 32. This process is illustrated in the code fragment below:

```
loop:
lbu       $t0, 0($a0)      # size of the data field to extract
extpdpv   $v0, ac0, $t0    # extract a value from ac0, pos -= size
addiu     $a1, $a1, 4      # increment output data pointer
addiu     $a0, $a0, 1      # increment size table pointer
```

```
bposge32  loop              # loop back if pos >= 32
sw        $v0, -4($a1)      # store the extracted value

lw        $t1, 0($a2)       # load next bitstream word
addiu     $a2, $a2, 4       # increment bitstream pointer
bne       $a2, $a3, loop    # loop until no more data available
mthlip    $t1, ac0          # update ac0 bit buffer, pos += 32
```

The illustrated code loads the size (bit width) of each field to be extracted from memory. The performance of the loop can be further improved if the sizes are static and known in advance.

13.7 Compiler Usage Model for MIPS DSP-R2

The MIPS® DSP-R2 ASE allows for performance optimization of signal processing and multimedia applications running on the proAptiv core. The DSP-R2 ASE includes a set of instructions that provide computational support for fractional data types, SIMD, saturation, and other operations commonly used in DSP applications. This section describes a compiler usage model for the MIPS DSP-R2.

Typical DSP applications include certain loops or kernels that take a large percentage of the execution time. Because of the sensitivity of DSP application performance to these kernels, programmers have tended to write these functions in assembly, hand-scheduling the code for optimal pipeline scheduling. But with the introduction of MIPS DSP-R2 extensions, compilers like GCC can be used to reduce, and perhaps eliminate, the need to write assembly code. However, to obtain the best optimizations, a particular coding style and usage must be followed, as explained in this section.

For reference, the GCC Compiler 6.03.00-rc3, based on FSF (Free Software Foundation) GCC 3.4, was used for the examples in this section. Programmers should use the newest compilers available to ensure that the DSP-R2 ASE is supported and to enable the highest performance instruction scheduling. Throughout the section, the term “GCC compiler” refers to a compiler that incorporates the DSP-R2 ASE intrinsics, such as the GCC compiler.

Using a high-level language such as C to develop applications has many advantages:

1. C programmers do not have to manually allocate registers—the compiler can allocate registers for variables.

2. C programmers do not have to manually schedule instructions—the compiler can schedule instructions based on given latency information for specific CPU pipelines.

3. C programmers do not have to consider function calling conventions when writing modular programs—the compiler saves and restores registers at entries and exits of functions per a pre-defined convention.

4. C programmers can declare SIMD variables and use generic C operators or intrinsics (built-in functions) to manage SIMD variables in order to achieve parallelism.

5. Development and debug time is shortened when using a high-level language.

6. Applications are more maintainable when written in a high-level language.

13.7.1 Enabling the DSP-R2 ASE in the Compiler

To enable the MIPS32 DSP-R2 ASE in the GCC compiler, the “-mdsp2” command-line option is required. In addition to “-mdsp2”, “-mips32r2” is recommended for better performance in accessing elements in SIMD variables, because this allows the use of the MIPS32 Release 2 instruction **INS** for more efficient code.

13.7.2 Data Types

In the GCC compiler, the Q15 data type is represented by 16-bit integer data type (short), and the Q31 data type is represented by 32-bit integer data type (int). Typedefs can be created for Q15 and Q31 as follows:

```
typedef short q15;
typedef int q31;
```

The MIPS32 DSP-R2 ASE implements four 64-bit accumulators which can be represented by a “long long” data type.

```
typedef long long a64;
```

To declare SIMD data types, typedefs with special vector_size attributes are required. For example,

```
typedef signed char v4i8 __attribute__ ((vector_size(4)));
typedef short v2q15 __attribute__ ((vector_size(4)));
```

where “v4i8” defines a SIMD data type containing four 8-bit integer data. “v2q15” defines a type containing two Q15 fractional data (which is the same as two 16-bit integer data).

13.7.3 Initialization of Q15 and Q31 Variables

To initialize Q15 variables, programmers can multiply the fractional value (e.g., 0.1234) by 32768.0. To initialize Q31 variables, programmers can multiply the fractional value by 2147483648.0.

```
Ex: /* Q15 Example  */
typedef short q15;
q15 a = 0.1234 * 32768.0;
/* ---------------------------------------------------------- */

Ex: /* Q31 Example  */typedef int q31;
q31 b = 0.2468 * 2147483648.0;
```

13.7.4 Initialization of SIMD Variables

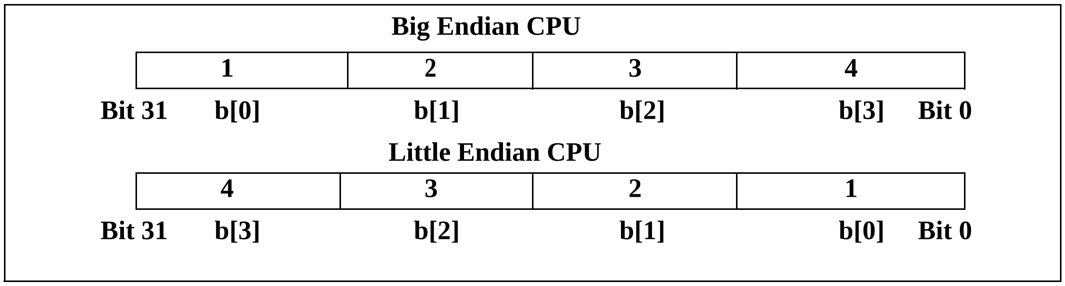
To initialize SIMD variables is similar to initializing aggregate data. The following examples show how to initialize SIMD variables.

```
Ex: /* v4i8 Example  */
v4i8 a = {1, 2, 3, 4};
v4i8 b;
b = (v4i8) {5, 6, 7, 8};
/* ---------------------------------------------------------- */

Ex: /* v2q15 Example  */
v2q15 a = {0x0fcb, 0x3a75};
v2q15 b;
b = (v2q15) {0.1234 * 32768.0, 0.4567 * 32768.0};
```

Note that CPU endianness affects the ordering of data stored in a register. In a Big Endian CPU, data is stored from the left-to-right location of a register. In a Little Endian CPU, data is stored from the right-to-left location of a register. For the example of v4i8 a = {1, 2, 3, 4}, a Big Endian CPU stores 1, 2, 3, and 4 from the left-to-right location, but a Little Endian CPU stores 1, 2, 3, and 4 from the right-to-left location as shown in Figure 13.2.

Figure 13.2 Register Values for v4i8 a = {1, 2, 3, 4} in Big Endian and Little Endian CPUs



Most arithmetic operations will simply work on the SIMD operands in the register irrespective of endianness. But the programmer must beware of such instructions in the DSP-R2 ASE that directly refer to the left or right portions of a GPR. For example, MAQ_SA.W.PHL.

13.7.5 Accessing Elements in SIMD Variables

The use of SIMD variables enables operations on multiple data in parallel. However, in certain situations, programmers need to access elements inside a SIMD variable. This can be done by using a union type that unites a SIMD type and an array of a basic type as follows.

```
typedef union
{
  v4i8 a;
  unsigned char b[4];
} v4i8_union;

typedef short q15;
typedef union
{
  v2q15 a;
  q15 b[2];
} v2q15_union;
```

As shown in Figure 13.2 for a v4i8 variable, b[0] is used in both big-endian and Little Endian CPUs to access the first element in the variable. However, b[0] is stored in the left-most position in a Big Endian CPU, but it is stored in the right-most position in a little-endian CPU. The following examples show how to extract or assign elements.

```
Ex: /* v4i8 Example  */
v4i8 i;
unsigned char j, k, l, m;
v4i8_union temp;

/* Assume we want to extract from i.  */
temp.a = i;
j = temp.b[0];
k = temp.b[1];
l = temp.b[2];
m = temp.b[3];

/* Assume we want to assign j, k, l, m to i.  */
temp.b[0] = j;
temp.b[1] = k;
temp.b[2] = l;
temp.b[3] = m;
i = temp.a;
/* ---------------------------------------------------------- */

Ex: /* v2q15 Example  */
v2q15 i;
q15 j, k;
v2q15_union temp;

/* Assume we want to extract from i.  */
temp.a = i;
j = temp.b[0];
k = temp.b[1];

/* Assume we want to assign j, k to i.  */
temp.b[0] = j;
temp.b[1] = k;
i = temp.a;
```

13.7.6 C Operators

Addition and subtraction on fractional data are similar to addition and subtraction with integer data, but multiplication requires a post-multiply shift to align the resulting values appropriately. Because fractional data uses integer data types in the GCC compiler, users must be very cautious when applying operators upon fractional data. The GCC compiler accepts all operators upon Q15 and Q31 data, but only addition and subtraction generate the expected results for Q15 and Q31. Note that operators other than “+” and “-” upon Q15 and Q31 are treated as integer arithmetic.

```
Ex: /* Q15 Example  */
typedef short q15;
q15 i, j, k, l;
i = k + l;
j = k - l;
/* ---------------------------------------------------------- */

Ex: /* Q31 Example  */
typedef int q31;
q31 i, j, k, l;
i = k + l;
j = k - l;
```

Certain C operators can be applied to SIMD variables. They are +, -, *, /, unary minus, ^, |, &, ~. The MIPS32 DSP-R2 ASE provides SIMD addition and subtraction instructions for v4i8 and v2q15, allowing the GCC compiler to generate appropriate instructions for addition and subtraction of v4i8 and v2q15 variables. For other operators, the GCC compiler synthesizes a sequence of instructions. The examples here show compiler-generated SIMD instructions when the appropriate operator is applied to SIMD variables.

```
Ex: /* v4i8 Addition  */
v4i8 test1 (v4i8 a, v4i8 b)
{
  return a + b;
}

# Generated Assembly
```

```
test1:
  j       $31
  addu.qb $2, $4, $5
# -----------------------------------------------------------

Ex: /* v4i8 Subtraction  */
v4i8 test2 (v4i8 a, v4i8 b)
{
  return a - b;
}

# Generated Assembly
test2:
  j       $31
  subu.qb $2, $4, $5
# -----------------------------------------------------------

Ex: /* v2q15 Addition  */
v2q15 test3 (v2q15 a, v2q15 b)
{
  return a + b;
}

# Generated Assembly
test3:
  j       $31
  addq.ph $2, $4, $5
# -----------------------------------------------------------

Ex: /* v2q15 Subtraction  */
v2q15 test4 (v2q15 a, v2q15 b)
{
  return a - b;
}

# Generated Assembly
test4:
  j       $31
  subq.ph $2, $4, $5
```

In situations where special integer and fractional calculations are needed and the compiler cannot generate them automatically, C intrinsics can be directly used by the programmer as described in the next section.

13.7.7 C Intrinsics for the MIPS32 DSP-R2 ASE

Intrinsics are very similar to function calls in syntax. Programmers need to pass parameters to intrinsics, and intrinsics return results to variables. The difference between intrinsics and functions is that the compiler directly maps intrinsics to instructions for better performance. Intrinsics for the MIPS32 DSP-R2 ASE use the following data types:

```
typedef signed char v4i8 __attribute__ ((vector_size(4)));
typedef short v2i16 __attribute__ ((vector_size(4)));
typedef short v2q15 __attribute__ ((vector_size(4)));
typedef int q31;
typedef int i32;
typedef long long a64;
typedef unsigned int ui32;
```

NOTE: “q31” and “i32” are actually the same as “int”, but the intrinsic that accepts “q31” processes data as Q31 fractional data, and the intrinsic that accepts “i32” processes data as 32-bit integer data.

NOTE: “a64” is the same as “long long”, but the compiler allocates “a64” variables to accumulators ($ac0, $ac1, $ac2, $ac3) ready to be operated on relevant DSP-R2 instructions.

The list of all intrinsics can be found in the MIPS32 DSP-R2 Intrinsics, and Section 13.7.9 will introduce each MIPS32 DSP-R2 intrinsic. Programmers should be familiar with the semantics of all MIPS32 DSP-R2 instructions so that the corresponding intrinsic can be used appropriately in C programs to achieve better performance.

One example of an intrinsic for the **ADDQ.PH** instruction is “v2q15 __builtin_mips_addq_ph (v2q15, v2q15).” Two v2q15 variables are required to be passed to “__builtin_mips_addq_ph” and one v2q15 variable is needed to receive the returned result from this intrinsic. The following C code demonstrates the usage of “__builtin_mips_addq_ph”.

```
Ex:
v2q15 test5 ()
{
  v2q15 a, b, c;
  a = (v2q15) {0.12 * 32768.0, 0.34 * 32768.0};
  b = (v2q15) {0.56 * 32768.0, 0.78 * 32768.0};
  c = __builtin_mips_addq_ph (a, b);
  return c;
}

# Generated Assembly
        .file   1 "test5.c"
        .section .mdebug.abi32
        .previous
        .section        .rodata.cst4,"aM",@progbits,4
        .align  2
.LC0:
        .half   3921
        .half   11141
        .align  2
.LC1:
        .half   18299
        .half   25559
        .text
        .align  2
        .align  3
        .globl  test5
        .set    nomips16
        .ent    test5
test5:
        .frame  $sp,0,$31               # vars= 0, regs= 0/0, args= 0, gp= 0
        .mask   0x00000000,0
        .fmask  0x00000000,0
        .set    noreorder
        .set    nomacro

        lui     $5,%hi(.LC0)
        lui     $4,%hi(.LC1)
        lw      $2,%lo(.LC0)($5)
        lw      $3,%lo(.LC1)($4)
        j       $31
        addq.ph $2,$2,$3
        .set    macro
        .set    reorder
        .end    test5
        .ident  "GCC: (GNU) 3.4.4 mipssde-6.03.00-20051020"
```

13.7.8 Compiler Usage and the DSPControl Register

The MIPS32 DSP-R2 ASE includes a new DSP control register that has six fields as described in Section 13.4.2 "DSP-R2 ASE Control Register". These fields are:

- CCOND (condition code bits)
- OUFLAG (overflow/underflow bits)
- EFI (extract fail indicator bit)
- C (carry bit)
- SCOUNT (size count bits)
- POS (position bits).

The compiler treats the SCOUNT and POS fields as global variables, such that instructions that modify SCOUNT or POS are never optimized away. These instructions include **`WRDSP, EXTPDP, EXTPDPV`**, and **`MTHLIP`**. A function call that jumps to a function containing **`WRDSP, EXTPDP, EXTPDPV`**, or **`MTHLIP`** is also never deleted by the compiler.

For correctness, programmers must assume that a function call clobbers all fields of the DSP control register. That is, programmers cannot depend on the values in CCOND, OUFLAG, EFI or C across a function-call boundary. They must re-initialize the values of CCOND, OUFLAG, EFI or C before using them. Note that because SCOUNT and POS fields are treated as global variables, the values of SCOUNT and POS are always valid across function-call boundaries and can be used without re-initialization.

The following example shows possibly incorrect code. The first intrinsic "__builtin_mips_addsc" sets the carry bit (C) in the DSP control register, and the second intrinsic "__builtin_mips_addwc" reads the carry bit (C) from the DSP control register. However, a function call "func" inserted between "__builtin_mips_addsc" and "__builtin_mips_addwc" may change the carry bit to affect the correct result of "__builtin_mips_addwc".

```
Ex:
int test6 (int a, int b, int c, int d)
{
  __builtin_mips_addsc (a, b);
  func();
  return __builtin_mips_addwc (c, d);
}
```

The previous example may be corrected by moving "func" before the first intrinsic or after the second intrinsic as follows.

```
Ex:
int test7 (int a, int b, int c, int d)
{
  func();
  __builtin_mips_addsc (a, b);
  return __builtin_mips_addwc (c, d);
}
/* ----------------------------------------------------------- */
```

```
int test8 (int a, int b, int c, int d)
{
  int i;
  __builtin_mips_addsc (a, b);
  i = __builtin_mips_addwc (c, d);
  func();
  return i;
}
```

13.7.9 C-Based Intrinsics for the MIPS32® DSP-R2 ASE

This section provides a basic introduction to all the intrinsics supported for the MIPS32 DSP-R2 ASE. The intrinsics are illustrated using examples, and some usage tips are provided as well. They are categorized by function and data size type as follows:

- Intrinsics to access and use the DSPControl register
- Intrinsics for signed and unsigned 8-bit integers
- Intrinsics for Q15 data
- Intrinsics for Q31 data
- Intrinsics for mixed data types of 8-bit integers and Q15/16-bit integers
- Intrinsics for mixed data types of Q15 and Q31
- Intrinsics for 64-bit accumulators
- Intrinsics for 32-bit integers
- Intrinsics for 16-bit integers
- Intrinsics for mixed data types of 16-bit integers and 32-bit integers

13.7.9.1 Intrinsics for Instructions that Access and Use the DSPControl Register

Read/Write the DSPControl Register

```
i32 __builtin_mips_rddsp (imm0_63);
void __builtin_mips_wrdsp (i32, imm0_63);
```

The immediate parameter, imm0_63 used in the two intrinsics here is a mask value used to specify which fields of the DSPControl register should be read or written respectively. The correspondence of the specific bits of the mask to specific fields in the DSPControl register is shown in Figure 13.3. As shown, bit 0 of imm0_63 is for the POS field, bit 1 of imm0_63 is for the SCOUNT field, bit 2 of imm0_63 is for the C field, bit 3 of imm0_63 is for the OUFLAG, bit 4 of imm0_63 is for the CCOND flag, and bit 5 of imm0_63 is for the EFI field. For example, to read the SCOUNT field, imm0_63 must be set to 2. To read all fields, imm0_63 must be set to 63 (1 + 2 + 4 + 8 + 16 + 32).

```
Ex:
int the_scount = (__builtin_mips_rddsp (2)) >> 7; // Read SCOUNT
int all_fields = __builtin_mips_rddsp (63); // Read all fields
```

To write the DSPControl register, programmers must pass a 32-bit integer as the first parameter to "__builtin_mips_wrdsp", as well as the imm0_63 mask value that determines which fields are to be updated. The first parameter should be a 32-bit value that mimics the format of the DSPControl register fields. Then, based on the mask value, the corresponding fields will be copied from this 32-bit value to the DSPControl register. For example, to set the SCOUNT field to 63, (63<<7) is passed as the first parameter and second parameter imm0_63 must be 2 so that an update of the SCOUNT field is done to the value 63 from the first input. To update all bits of all fields to 1, the first parameter can be 0xFFFFFFFF with a second parameter value of 63.

```
Ex:
__builtin_mips_wrdsp (63<<7, 2); // Update SCOUNT to 63
__builtin_mips_wrdsp (0xFFFFFFFF, 63); // Update all bits of fields to 1
```

Figure 13.3 Mask Value to Access the MIPS32 DSPControl Register

Bit	31 28	27 24	23 16	15	14	13	12 7	6	5 0
	0	CCOND	OUFLAG	0	EFI	C	SCOUNT	0	POS
IMM0_63 =>		16	8		32	4	2		1

Branch on Greater Than or Equal to 32 in POS

```
i32 __builtin_mips_bposge32 ();
```

This intrinsic returns 1 if the value of the POS field is greater than or equal to 32. Otherwise, the intrinsic returns 0. Programmers can use "__builtin_mips_bposge32" inside a condition test, and the compiler will optimize the code to generate the "bposge32" instruction as follows.

```
Ex:
void test9 ()
{
if (__builtin_mips_bposge32())
  result_is_true();
else
  result_is_false();
}
# Generated Assembly
test9:
        .set    noreorder
        .set    nomacro
        bposge32        .L3
        nop
        j       result_is_false
        nop
        .align  3
.L3:
        j       result_is_true
        nop
```

13.7.9.2 Using Intrinsics for Signed and Unsigned 8-bit Integers

This section introduces intrinsics that operate on signed and unsigned 8-bit integers in register SIMD fashion by using a "v4i8" data type. If the programmer wants to perform an operation such as add on a single 8-bit item, then these intrinsics can still be used by ignoring the other three un-used elements inside the "v4i8" variable. Each set of intrinsics for operations are listed below, with simple examples of their usage.

Unsigned Add/Subtract with Optional Saturation

```
v4i8 __builtin_mips_addu_qb (v4i8, v4i8);
v4i8 __builtin_mips_addu_s_qb (v4i8, v4i8);
v4i8 __builtin_mips_subu_qb (v4i8, v4i8);
v4i8 __builtin_mips_subu_s_qb (v4i8, v4i8);
#----------------------------------------------------------------------------
Ex:
v4i8 a = {1, 2, 3, 0xFF};
v4i8 b = {2, 4, 6, 8};
v4i8 r1, r2, r3, r4;
r1 = __builtin_mips_addu_qb (a, b);   // r1 will be {3, 6, 9, 7}
r2 = __builtin_mips_addu_s_qb (a, b); // r2 will be {3, 6, 9, 0xFF}
r3 = __builtin_mips_subu_qb (a, b);   // r3 will be {0xFF, 0xFE, 0xFD, 0xF7}
r4 = __builtin_mips_subu_s_qb (a, b); // r4 will be {0, 0, 0, 0xF7}
```

Unsigned Reduction Add

```
i32 __builtin_mips_raddu_w_qb (v4i8);
#----------------------------------------------------------------------------
Ex:
v4i8 a = {1, 2, 3, 4};
int sum = __builtin_mips_raddu_w_qb (a); // sum will be 1 + 2 + 3 + 4 = 10
```

Shift Left/Right Logical

```
v4i8 __builtin_mips_shll_qb (v4i8, imm0_7);
v4i8 __builtin_mips_shll_qb (v4i8, i32);
v4i8 __builtin_mips_shrl_qb (v4i8, imm0_7);
v4i8 __builtin_mips_shrl_qb (v4i8, i32);
#----------------------------------------------------------------------------
Ex:
v4i8 a = {1, 2, 3, 4};
v4i8 b = {128, 64, 32, 16};
v4i8 r1, r2, r3, r4;
int shift_amount = 2;
r1 = __builtin_mips_shll_qb (a, 1); // r1 will be {2, 4, 6, 8}
r2 = __builtin_mips_shll_qb (a, shift_amount); // r2 will be {4, 8, 12, 16}
r3 = __builtin_mips_shrl_qb (b, 3); // r3 will be {16, 8, 4, 2}
r4 = __builtin_mips_shrl_qb (b, shift_amount); // r4 will be {32, 16, 8, 4}
```

Dot Product with Accumulate/Subtract

```
a64 __builtin_mips_dpau_h_qbl (a64, v4i8, v4i8);
a64 __builtin_mips_dpau_h_qbr (a64, v4i8, v4i8);
a64 __builtin_mips_dpsu_h_qbl (a64, v4i8, v4i8);
a64 __builtin_mips_dpsu_h_qbr (a64, v4i8, v4i8);
```

NOTES:
1. The result will be a 64-bit integer.
2. The processor endianness of the data affects the format of the result.
3. Using the same “a64” variable for both the target and the first parameter could result in better performance.

```
#----------------------------------------------------------------------------
Ex: /* Assume a big-endian CPU */
v4i8 a = {1, 2, 3, 4};
v4i8 b = {4, 5, 6, 7};
a64 ac1, ac2, ac3, ac4;
```

```
ac1 = ac2 = ac3 = ac4 = 0;
ac1 = __builtin_mips_dpau_h_qbl (ac1, a, b); // ac1 will be 0 + 1*4 + 2*5 = 14
ac2 = __builtin_mips_dpau_h_qbr (ac2, a, b); // ac2 will be 0 + 3*6 + 4*7 = 46
ac3 = __builtin_mips_dpsu_h_qbl (ac3, a, b); // ac3 will be 0 - (1*4 + 2*5) = -14
ac4 = __builtin_mips_dpsu_h_qbr (ac4, a, b); // ac4 will be 0 - (3*6 + 4*7) = -46
```

Replicate a Fixed Byte Value into SIMD Elements

```
v4i8 __builtin_mips_repl_qb (imm0_255);
v4i8 __builtin_mips_repl_qb (i32);
#--------------------------------------------------------------------------
Ex:
v4i8 a, b;
int value = 100;
a = __builtin_mips_repl_qb (10); // a will be {10, 10, 10, 10}
b = __builtin_mips_repl_qb (value); // b will be {100, 100, 100, 100}
```

Compare Unsigned

```
void __builtin_mips_cmpu_eq_qb (v4i8, v4i8);
void __builtin_mips_cmpu_lt_qb (v4i8, v4i8);
void __builtin_mips_cmpu_le_qb (v4i8, v4i8);
i32 __builtin_mips_cmpgu_eq_qb (v4i8, v4i8);
i32 __builtin_mips_cmpgu_lt_qb (v4i8, v4i8);
i32 __builtin_mips_cmpgu_le_qb (v4i8, v4i8);
i32 __builtin_mips_cmpgdu_eq_qb (v4i8, v4i8); // DSPR2
i32 __builtin_mips_cmpgdu_lt_qb (v4i8, v4i8); // DSPR2
i32 __builtin_mips_cmpgdu_le_qb (v4i8, v4i8); // DSPR2
```

Note that the first three intrinsics update the condition code bits of the DSPControl register, but the middle three intrinsics write the condition code bits to a specified general purpose register. The last three intrinsics do the both.

```
#--------------------------------------------------------------------------
Ex: /* Assume a big-endian CPU */
v4i8 a = {1, 4, 10, 8};
v4i8 b = {1, 2, 100, 8};
int r1, r2, r3;
__builtin_mips_cmpu_eq_qb (a, b); // CCOND bits will be 9 (= 1001b)
__builtin_mips_cmpu_lt_qb (a, b); // CCOND bits will be 2 (= 0010b)
__builtin_mips_cmpu_le_qb (a, b); // CCOND bits will be 11 (= 1011b)
r1 = __builtin_mips_cmpgu_eq_qb (a, b); // r1 will be 9
r2 = __builtin_mips_cmpgu_lt_qb (a, b); // r2 will be 2
r3 = __builtin_mips_cmpgu_le_qb (a, b); // r3 will be 11
r1 = __builtin_mips_cmpgdu_eq_qb (a, b); // Both CCOND bits and r1 will be 9
r2 = __builtin_mips_cmpgdu_lt_qb (a, b); // Both CCOND bits and r2 will be 2
r3 = __builtin_mips_cmpgdu_le_qb (a, b); // Both CCOND bits and r3 will be 11
```

Pick Based on Condition Code Bits

```
v4i8 __builtin_mips_pick_qb (v4i8, v4i8);
```

Note that this intrinsic is usually used together with the first three compare intrinsics in Section .

```
#--------------------------------------------------------------------------
Ex:
v4i8 a = {1, 4, 10, 8};
v4i8 b = {1, 2, 100, 8};
v4i8 r;
```

```
__builtin_mips_cmpu_eq_qb (a, b); // CCOND bits will be 9 (= 1001b)
r = __builtin_mips_pick_qb (a, b); // r will be {1, 2, 100, 8}
```

Find Absolute Value

```
v4i8 __builtin_mips_absq_s_qb (v4i8); // DSPR2
#---------------------------------------------------------------------------
Ex:
v4i8 a = {-1, -128, 1, 127};
v4i8 r;
r = __builtin_mips_absq_s_qb (a); // r will be {1, 127, 1, 127}
/* Note that the absolute value of -128 is 128 that is represented by the maximum
value as 127. */
```

Unsigned Add and Right Shift to Halve Results with Optional Rounding

```
v4i8 __builtin_mips_adduh_qb (v4i8, v4i8); // DSPR2
v4i8 __builtin_mips_adduh_r_qb (v4i8, v4i8); // DSPR2
#---------------------------------------------------------------------------
Ex:
v4i8 a = {1, 2, 3, 4};
v4i8 b = {0x80, 0x80, 0x80, 0x80};
v4i8 r1, r2;
r1 = __builtin_mips_adduh_qb (a, b); // r1 will be {0x40, 0x41, 0x41, 0x42}
r2 = __builtin_mips_adduh_r_qb (a, b); // r2 will be {0x41, 0x41, 0x44, 0x42}
```

Shift Right Arithmetic with Optional Rounding

```
  v4i8 __builtin_mips_shra_qb (v4i8, imm0_7); // DSPR2
  v4i8 __builtin_mips_shra_r_qb (v4i8, imm0_7); // DSPR2
  v4i8 __builtin_mips_shra_qb (v4i8, i32); // DSPR2
  v4i8 __builtin_mips_shra_r_qb (v4i8, i32); // DSPR2
#---------------------------------------------------------------------------
Ex:
v4i8 a = {0x40, 0x20, 0x10, 0x0F};
v4i8 r1, r2, r3, r4;
int shift_amount = 2;
r1 = __builtin_mips_shra_qb (a, 2); // r1 will be {0x10, 0x08, 0x04, 0x3}
r2 = __builtin_mips_shra_r_qb (a, 2); // r2 will be {0x10, 0x08, 0x04, 0x4}
r3 = __builtin_mips_shra_qb (a, shift_amount); // r3 will be {0x10, 0x08, 0x04, 0x3}
r4 = __builtin_mips_shra_r_qb (a, shift_amount); // r4 will be {0x10, 0x08, 0x04, 0x4}
```

Unsigned Subtract and Right Shift to Halve Results with Optional Rounding

```
v4i8 __builtin_mips_subuh_qb (v4i8, v4i8); // DSPR2
v4i8 __builtin_mips_subuh_r_qb (v4i8, v4i8); // DSPR2
#---------------------------------------------------------------------------
Ex:
v4i8 a = {0x80, 0x80, 0x80, 0x80};
v4i8 b = {1, 2, 3, 4};
v4i8 r1, r2;
r1 = __builtin_mips_subuh_qb (a, b); // r1 will be {0x3F, 0x3F, 0x3E, 0x3E}
r2 = __builtin_mips_subuh_r_qb (a, b); // r2 will be {0x40, 0x3F, 0x3F, 0x3E}
```

13.7.9.3 Using Intrinsics for Q15 Data Type

This section introduces intrinsics that operate on Q15 data present in register SIMD fashion by using a “v2q15” data type. If the programmer wants to perform the specified operation on a single data in the register, then these intrinsics can still be used while ignoring the other element inside the “v2q15” variable.

Add/Subtract with Optional Saturation

```
v2q15 __builtin_mips_addq_ph (v2q15, v2q15);
v2q15 __builtin_mips_addq_s_ph (v2q15, v2q15);
v2q15 __builtin_mips_subq_ph (v2q15, v2q15);
v2q15 __builtin_mips_subq_s_ph (v2q15, v2q15);
#----------------------------------------------------------------------------
Ex:
v2q15 a = {0x0000, 0x8000};
v2q15 b = {0x8000, 0x8000};
v2q15 r1, r2, r3, r4;
r1 = __builtin_mips_addq_ph (a, b); // r1 will be {0x8000, 0x0000}
r2 = __builtin_mips_addq_s_ph (a, b); // r2 will be {0x8000, 0x8000}
r3 = __builtin_mips_subq_ph (a, b); // r3 will be {0x8000, 0x0000}
r4 = __builtin_mips_subq_s_ph (a, b); // r4 will be {0x7FFF, 0x0000}
```

Find Absolute Value

```
v2q15 __builtin_mips_absq_s_ph (v2q15);
#----------------------------------------------------------------------------
Ex:
v2q15 a = {0xFFFF, 0x8000};
v2q15 r;
r = __builtin_mips_absq_s_ph (a); // r will be {0x0001, 0x7FFF}
/* Note that the value of 0x8000 is -1 in Q15.  The absolute value of -1 is 1 that
is represented by the maximum value as 0x7FFF in Q15.  */
```

Shift Left Logical with Optional Saturation

```
v2q15 __builtin_mips_shll_ph (v2q15, imm0_15);
v2q15 __builtin_mips_shll_ph (v2q15, i32);
v2q15 __builtin_mips_shll_s_ph (v2q15, imm0_15);
v2q15 __builtin_mips_shll_s_ph (v2q15, i32);
#----------------------------------------------------------------------------
Ex:
v2q15 a = {0x0001, 0x8000};
v2q15 r1, r2, r3, r4;
int shift_amount = 2;
r1 = __builtin_mips_shll_ph (a, 1); // r1 will be {0x0002, 0x0000}
r2 = __builtin_mips_shll_ph (a, shift_amount); // r2 will be {0x0004, 0x0000}
r3 = __builtin_mips_shll_s_ph (a, 1); // r3 will be {0x0002, 0x8000}
r4 = __builtin_mips_shll_s_ph (a, shift_amount); // r4 will be {0x0004, 0x8000}
```

Shift Right Arithmetic with Optional Rounding

```
v2q15 __builtin_mips_shra_ph (v2q15, imm0_15);
v2q15 __builtin_mips_shra_ph (v2q15, i32);
v2q15 __builtin_mips_shra_r_ph (v2q15, imm0_15);
v2q15 __builtin_mips_shra_r_ph (v2q15, i32);
#----------------------------------------------------------------------------
Ex:
```

```
v2q15 a = {0x7FFF, 0x8000};
v2q15 r1, r2, r3, r4;
int shift_amount = 2;
r1 = __builtin_mips_shra_ph (a, 1); // r1 will be {0x3FFF, 0xC000}
r2 = __builtin_mips_shra_ph (a, shift_amount); // r2 will be {0x1FFF, 0xE000}
r3 = __builtin_mips_shra_r_ph (a, 1); // r3 will be {0x4000, 0xC000}
r4 = __builtin_mips_shra_r_ph (a, shift_amount); // r4 will be {0x2000, 0xE000}
```

Multiply with Rounding and Saturation (Q15 x Q15 => Q15)

```
v2q15 __builtin_mips_mulq_rs_ph (v2q15, v2q15);
#----------------------------------------------------------------------------
Ex:
v2q15 a = {0x7FFF, 0x8000};
v2q15 b = {0x7FFF, 0x8000};
v2q15 r;
r = __builtin_mips_mulq_rs_ph (a, b); // r will be {0x7FFE, 0x7FFF}
```

Dot Product with Accumulate/Subtract (Q15 x Q15 => Q32.31)

```
a64 __builtin_mips_dpaq_s_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpsq_s_w_ph (a64, v2q15, v2q15);
```

NOTES:
1. The result will be in the Q32.31 format.
2. Using the same “a64” variable for the target and the first parameter could lead to better performance.

```
#----------------------------------------------------------------------------
Ex:
v2q15 a = {0x0001, 0x8000};
v2q15 b = {0x0002, 0x8000};
a64 ac1, ac2;
ac1 = ac2 = 0;
ac1 = __builtin_mips_dpaq_s_w_ph (ac1, a, b); // ac1 will be 0 + (1*2)<<1 +
                                              // 0x7FFFFFFF = 0x0000000080000003
ac2 = __builtin_mips_dpsq_s_w_ph (ac2, a, b); // ac2 will be 0 - (1*2)<<1 -
                                              // 0x7FFFFFFF = 0xFFFFFFFF7FFFFFFD
```

Multiply and Subtract and Accumulate (Q15 x Q15 => Q32.31)

```
a64 __builtin_mips_mulsaq_s_w_ph (a64, v2q15, v2q15);
```

NOTES:
1. The result will be in the Q32.31 format.
2. The processor endianness affects the format of the result.
3. Using the same “a64” variable for the target and the first parameter could lead to better performance.

```
#----------------------------------------------------------------------------
Ex: /* Assume a big-endian CPU */
v2q15 a = {0x0001, 0x8000};
v2q15 b = {0x0002, 0x8000};
a64 ac1 = 0;
ac1 = __builtin_mips_mulsaq_s_w_ph (ac1, a, b); // ac1 will be 0 + (1*2)<<1 -
                                                // 0x7FFFFFFF = 0xFFFFFFFF80000005
```

Multiply with Accumulate a Single Element (Q15 x Q15 => Q31)

```
a64 __builtin_mips_maq_s_w_phl (a64, v2q15, v2q15);
a64 __builtin_mips_maq_s_w_phr (a64, v2q15, v2q15);
```

```
a64 __builtin_mips_maq_sa_w_phl (a64, v2q15, v2q15);
a64 __builtin_mips_maq_sa_w_phr (a64, v2q15, v2q15);
```

NOTES:
1. The result will be in the Q31 format.
2. The processor endianness affects the format of the result.
3. Using the same “a64” variable for the target and the first parameter could lead to better performance.

```
#--------------------------------------------------------------------------
Ex: /* Assume a big-endian CPU */
v2q15 a = {0x0001, 0x8000};
v2q15 b = {0x0002, 0x8000};
a64 ac1, ac2, ac3, ac4;
ac1 = ac2 = 0;
ac3 = ac4 = 0x7FFFFFF0;
ac1 = __builtin_mips_maq_s_w_phl (ac1, a, b); // ac1 will be 0 + (1*2)<<1 =
                                              // 0x4
ac2 = __builtin_mips_maq_s_w_phr (ac2, a, b); // ac2 will be 0 + 0x7FFFFFFF =
                                              // 0x7FFFFFFF
ac3 = __builtin_mips_maq_sa_w_phl (ac3, a, b); // ac3 will be 0x7FFFFFF0 +
                                               // (1*2)<<1 = 0x7FFFFFF4
ac4 = __builtin_mips_maq_sa_w_phr (ac4, a, b); // ac4 will be 0x7FFFFFF0 +
                                               // 0x7FFFFFFF = 0x7FFFFFFF
```

Multiply Vector Fractional Left/Right Half-Words to Expanded Width Product with Saturation (Q15 x Q15 => Q31)

```
q31 __builtin_mips_muleq_s_w_phl (v2q15, v2q15);
q31 __builtin_mips_muleq_s_w_phr (v2q15, v2q15);
```

NOTES:
1. The result will be in the Q31 format.
2. The processor endianness affects the format of the result.

```
#--------------------------------------------------------------------------
Ex: /* Assume a big-endian CPU */
v2q15 a = {0x1234, 0x8000};
v2q15 a = {0x5678, 0x8000};
q31 r1, r2;
r1 = __builtin_mips_muleq_s_w_phl (a, b); // r1 will be 0x0C4C00C0
r2 = __builtin_mips_muleq_s_w_phr (a, b); // r2 will be 0x7FFFFFFF
```

Replicate a Fixed Half-word into Elements

```
v2q15 __builtin_mips_repl_ph (imm_n512_511);
v2q15 __builtin_mips_repl_ph (i32);
```

Note that for the immediate version, imm_n512_511 will be sign-extended to a 16-bit value and replicated into each SIMD element.

```
#--------------------------------------------------------------------------
Ex:
v2q15 r1, r2;
int value = 0x1234;
r1 = __builtin_mips_repl_ph (-512); // r1 will be {0xFE00, 0xFE00};
r2 = __builtin_mips_repl_ph (value); // r2 will be {0x1234, 0x1234};
```

Compare

```
void __builtin_mips_cmp_eq_ph (v2q15, v2q15);
void __builtin_mips_cmp_lt_ph (v2q15, v2q15);
void __builtin_mips_cmp_le_ph (v2q15, v2q15);
Ex:
v2q15 a = {0x1111, 0x1234};
v2q15 b = {0x4444, 0x1234};
__builtin_mips_cmp_eq_ph (a, b); // CCOND bits will be 1 (= 01b)
__builtin_mips_cmp_lt_ph (a, b); // CCOND bits will be 2 (= 10b)
__builtin_mips_cmp_le_ph (a, b); // CCOND bits will be 3 (= 11b)
```

Pick Based on Condition Code Bits

```
v2q15 __builtin_mips_pick_ph (v2q15, v2q15);
```

Note that this intrinsic is usually used together with the compare intrinsics in Section .

```
#---------------------------------------------------------------------------
Ex:
v2q15 a = {0x1111, 0x1234};
v2q15 b = {0x4444, 0x1234};
v2q15 r;
__builtin_mips_cmp_eq_ph (a, b); // CCOND bits will be 1 (= 01b)
r = __builtin_mips_pick_ph (a, b); // r will be {0x4444, 0x1234}
```

Pack from the Right and Left

```
v2q15 __builtin_mips_packrl_ph (v2q15, v2q15);
```

Note that the endianness affects the result.

```
#---------------------------------------------------------------------------
Ex: /* Assume a big-endian CPU */
v2q15 a = {0x1111, 0x2222};
v2q15 b = {0x3333, 0x4444};
v2q15 r;
r = __builtin_mips_packrl_ph (a, b); // r will be {0x2222, 0x3333}
```

Multiply with Saturation (Q15 x Q15 => Q15)

```
v2q15 __builtin_mips_mulq_s_ph (v2q15, v2q15); // DSPR2
#---------------------------------------------------------------------------
Ex:
v2q15 a = {0x7FFF, 0x8000};
v2q15 b = {0x7FFF, 0x8000};
v2q15 r;
r = __builtin_mips_mulq_s_ph (a, b); // r will be {0x7FFE, 0x7FFF}
```

Add and Right Shift to Halve Results with Optional Rounding

```
v2q15 __builtin_mips_addqh_ph (v2q15, v2q15); // DSPR2
v2q15 __builtin_mips_addqh_r_ph (v2q15, v2q15); // DSPR2
#---------------------------------------------------------------------------
Ex:
v2q15 a = {0x1000, 0x1000};
v2q15 b = {0x1001, 0x1000};
v2q15 r1, r2;
r1 = __builtin_mips_addqh_ph (a, b); // r1 will be {0x1000, 0x1000}
```

```
r2 = __builtin_mips_addqh_r_ph (a, b); // r1 will be {0x1001, 0x1000}
```

Subtract and Right Shift to Halve Results with Optional Rounding

```
v2q15 __builtin_mips_subqh_ph (v2q15, v2q15); // DSPR2
v2q15 __builtin_mips_subqh_r_ph (v2q15, v2q15); // DSPR2
#----------------------------------------------------------------------------
Ex:
v2q15 a = {0x1000, 0x1000};
v2q15 b = {0x1001, 0x1000};
v2q15 r1, r2;
r1 = __builtin_mips_subqh_ph (a, b); // r1 will be {0xFFFF, 0x0000}
r2 = __builtin_mips_subqh_r_ph (a, b); // r1 will be {0x0000, 0x0000}
```

Cross Dot Product with Accumulate/Subtract (Q15 x Q15 => Q32.31)

```
a64 __builtin_mips_dpaqx_s_w_ph (a64, v2q15, v2q15); // DSPR2
a64 __builtin_mips_dpaqx_sa_w_ph (a64, v2q15, v2q15); // DSPR2
a64 __builtin_mips_dpsqx_s_w_ph (a64, v2q15, v2q15); // DSPR2
a64 __builtin_mips_dpsqx_sa_w_ph (a64, v2q15, v2q15); // DSPR2
```

NOTES:
1. The result will be in the Q32.31 format.
2. Using the same “a64” variable for the target and the first parameter could lead to better performance.

```
#----------------------------------------------------------------------------
Ex:
v2q15 a = {0x0002, 0x8000};
v2q15 b = {0x8000, 0x0003};
a64 ac1, ac2, ac3, ac4;
ac1 = ac2 = ac3 = ac4 = 0;
ac1 = __builtin_mips_dpaqx_s_w_ph (ac1, a, b);  // ac1 will be
                                              //  0 + (2*3)<<1 + 0x7FFFFFFF
                                                   // = 0x000000008000000B
ac2 = __builtin_mips_dpaqx_sa_w_ph (ac2, a, b); // ac2 will be saturated to
                                                // 0x000000007FFFFFFF
ac3 = __builtin_mips_dpsqx_s_w_ph (ac3, a, b); // ac3 will be
                                               // 0 - (2*3)<<1 - 0x7FFFFFFF
                                               // = 0xFFFFFFFF7FFFFFF5
ac4 = __builtin_mips_dpsqx_sa_w_ph (ac4, a, b); // ac4 will be saturated
                                                // to 0xFFFFFFFF80000000
```

13.7.9.4 Using Intrinsics for Q31 Data Type

Add/Subtract with Saturation

```
q31 __builtin_mips_addq_s_w (q31, q31);
q31 __builtin_mips_subq_s_w (q31, q31);
#----------------------------------------------------------------------------
Ex:
q31 a = 0x12345678;
q31 b = 0x7FFFFFFF;
q31 r1, r2;
r1 = __builtin_mips_addq_s_w (a, b); // r1 will be 0x7FFFFFFF
r2 = __builtin_mips_subq_s_w (a, b); // r2 will be 0x92345679
```

Find Absolute Value

```
q31 __builtin_mips_absq_s_w (q31);
#----------------------------------------------------------------------------
Ex:
q31 a = 0x80000000;
q31 r;
r = __builtin_mips_absq_s_w (a); // r will be 0x7FFFFFFF
/* Note that the value of 0x80000000 is -1 in Q31.  The absolute value of -1 is 1
that is represented by the maximum value as 0x7FFFFFFF in Q31.  */
```

Shift Left Logical with Saturation

```
q31 __builtin_mips_shll_s_w (q31, imm0_31);
q31 __builtin_mips_shll_s_w (q31, i32);
#----------------------------------------------------------------------------
Ex:
q31 a = 0x70000000;
q31 r1, r2;
int shift_amount = 2;
r1 = __builtin_mips_shll_s_w (a, 1); // r1 will be 0x7FFFFFFF
r2 = __builtin_mips_shll_s_w (a, shift_amount); // r2 will be 0x7FFFFFFF
```

Shift Right Arithmetic with Rounding

```
q31 __builtin_mips_shra_r_w (q31, imm0_31);
q31 __builtin_mips_shra_r_w (q31, i32);
#----------------------------------------------------------------------------
Ex:
q31 a = 0x7FFFFFFF;
q31 r1, r2;
int shift_amount = 2;
r1 = __builtin_mips_shra_r_w (a, 1); // r1 will be 0x40000000
r2 = __builtin_mips_shra_r_w (a, shift_amount); // r2 will be 0x20000000
```

Dot Product with Accumulate/Subtract (Q31 x Q31 => Q63)

```
a64 __builtin_mips_dpaq_sa_l_w (a64, q31, q31);
a64 __builtin_mips_dpsq_sa_l_w (a64, q31, q31);
```

NOTES:
1.The result will be in the Q63 format.
2. Using the same “a64” variable for the target and the first parameter could lead to better performance.

```
#----------------------------------------------------------------------------
Ex:
q31 a = 0x80000000;
q31 b = 0x80000000;
a64 ac1, ac2;
ac1 = ac2 = 1;
ac1 = __builtin_mips_dpaq_sa_l_w (ac1, a, b); // ac1 will be 0x7FFFFFFFFFFFFFFF
ac2 = __builtin_mips_dpaq_sa_l_w (ac2, a, b); // ac2 will be 0x8000000000000002
```

Multiply with Rounding and Saturation (Q31 x Q31 => Q31)

```
q31 __builtin_mips_mulq_rs_w (q31, q31); // DSPR2
#----------------------------------------------------------------------------
Ex:
```

```
q31 a = 0x7FFFFFFF;
q31 b = 0x00000001;
q31 r;
r = __builtin_mips_mulq_rs_w (a, b); // r will be 0x00000001
```

Multiply with Saturation (Q31 x Q31 => Q31)

```
q31 __builtin_mips_mulq_s_w (q31, q31); // DSPR2
#----------------------------------------------------------------------------
Ex:
q31 a = 0x80000000;
q31 b = 0x00000001;
q31 r;
r = __builtin_mips_mulq_s_w (a, b); // r will be 0x00000000
```

Add and Right Shift to Halve Results with Optional Rounding

```
q31 __builtin_mips_addqh_w (q31, q31); // DSPR2
q31 __builtin_mips_addqh_r_w (q31, q31); // DSPR2
#----------------------------------------------------------------------------
Ex:
q31 a = 0x10000000;
q31 b = 0x10000001;
q31 r1, r2;
r1 = __builtin_mips_addqh_w (a, b); // r1 will be 0x10000000
r2 = __builtin_mips_addqh_r_w (a, b); // r2 will be 0x10000001
```

Subtract and Right Shift to Halve Results with Optional Rounding

```
q31 __builtin_mips_subqh_w (q31, q31); // DSPR2
q31 __builtin_mips_subqh_r_w (q31, q31); // DSPR2
#----------------------------------------------------------------------------
Ex:
q31 a = 0x10000000;
q31 b = 0x10000001;
q31 r1, r2;
r1 = __builtin_mips_subqh_w (a, b); // r1 will be 0xFFFFFFFF
r2 = __builtin_mips_subqh_r_w (a, b); // r2 will be 0x00000000
```

13.7.9.5 Using Intrinsics for Mixed Data Types: 8-bit Integers and Q15/16-bit Integers

Precision Reduce Four Fractional Half-words to Four Bytes

```
v4i8 __builtin_mips_precrq_qb_ph (v2q15, v2q15);
```

Note that the processor endianness affects the format of the result.

```
#----------------------------------------------------------------------------
Ex: /* Assume a big-endian CPU */
v2q15 a = {0x1234, 0x5678};
v2q15 b = {0x1111, 0x2222};
v4i8 r;
r = __builtin_mips_precrq_qb_ph (a, b); // r will be {0x12, 0x56, 0x11, 0x22}
```

Precision Reduce Unsigned Four Fractional Half-words to Four Bytes with Saturation

```
v4i8 __builtin_mips_precrqu_s_qb_ph (v2q15, v2q15);
```

Note that the processor endianness affects the format of the result.

```
#----------------------------------------------------------------------------
Ex: /* Assume a big-endian CPU */
v2q15 a = {0x7F79, 0xFFFF};
v2q15 b = {0x7F81, 0x2000};
v4i8 r;
r = __builtin_mips_precrqu_s_qb_ph (a, b); // r will be {0xFE, 0x00, 0xFF, 0x40}
```

Precision Expand Two Unsigned Bytes to Fractional Half-word Values

```
v2q15 __builtin_mips_precequ_ph_qbl (v4i8);
v2q15 __builtin_mips_precequ_ph_qbr (v4i8);
v2q15 __builtin_mips_precequ_ph_qbla (v4i8);
v2q15 __builtin_mips_precequ_ph_qbra (v4i8);
```

Note that the processor endianness affects the format of the result.

```
#----------------------------------------------------------------------------
Ex: /* Assume a big-endian CPU */
v4i8 a = {0x12, 0x34, 0x56, 0x78};
v2q15 r1, r2, r3, r4;
r1 = __builtin_mips_precequ_ph_qbl (a, b); // r1 will be {0x0900, 0x1A00}
r2 = __builtin_mips_precequ_ph_qbr (a, b); // r2 will be {0x2B00, 0x3C00}
r3 = __builtin_mips_precequ_ph_qbla (a, b); // r3 will be {0x0900, 0x2B00}
r4 = __builtin_mips_precequ_ph_qbra (a, b); // r4 will be {0x1A00, 0x3C00}
```

Precision Expand Two Unsigned Bytes to Unsigned Integer Half-words

```
v2q15 __builtin_mips_preceu_ph_qbl (v4i8);
v2q15 __builtin_mips_preceu_ph_qbr (v4i8);
v2q15 __builtin_mips_preceu_ph_qbla (v4i8);
v2q15 __builtin_mips_preceu_ph_qbra (v4i8);
```

Note that the processor endianness affects the format of the result.

```
#----------------------------------------------------------------------------
Ex: /* Assume a big-endian CPU */
v4i8 a = {0x12, 0x34, 0x56, 0x78};
v2q15 r1, r2, r3, r4;
r1 = __builtin_mips_preceu_ph_qbl (a, b); // r1 will be {0x0012, 0x0034}
r2 = __builtin_mips_preceu_ph_qbr (a, b); // r2 will be {0x0056, 0x0078}
r3 = __builtin_mips_preceu_ph_qbla (a, b); // r3 will be {0x0012, 0x0056}
r4 = __builtin_mips_preceu_ph_qbra (a, b); // r4 will be {0x0034, 0x0078}
```

Multiply Unsigned Vector Left/Right Bytes with Half-Words to Half Word Products with Saturation (Int8 x Q15 => Q15)

```
v2q15 __builtin_mips_muleu_s_ph_qbl (v4i8, v2q15);
v2q15 __builtin_mips_muleu_s_ph_qbr (v4i8, v2q15);
```

Note that the processor endianness affects the format of the result.

```
#----------------------------------------------------------------------------
Ex: /* Assume a big-endian CPU */
v4i8 a = {0x1, 0x3, 0x5, 0x7};
v2q15 b = {0x1234, 0x5678};
v2q15 r1, r2;
r1 = __builtin_mips_muleu_s_ph_qbl (a, b); // r1 will be {0x1234, 0xFFFF}
r2 = __builtin_mips_muleu_s_ph_qbr (a, b); // r2 will be {0x5B04, 0xFFFF}
```

Precision Reduce Four Integer Half-words to Four Bytes

```
v4i8 __builtin_mips_precr_qb_ph (v2i16, v2i16); // DSPR2
```

Note that the processor endianness affects the format of the result.

```
#-----------------------------------------------------------------------------
Ex: /* Assume a big-endian CPU */
v2i16 a = {0x7F79, 0xFFFF};
v2i16 b = {0x7F81, 0x2000};
v4i8 r;
r = __builtin_mips_precr_qb_ph (a, b); // r will be {0x79, 0xFF, 0x81, 0x00}
```

13.7.9.6 Using Intrinsics for Mixed Data Types: Q15 and Q31

Precision Reduce Two Fractional Words to Two Half-Words

```
v2q15 __builtin_mips_precrq_ph_w (q31, q31);
#-----------------------------------------------------------------------------
Ex:
q31 a = {0x12345678};
q31 b = {0x11112222};
v2q15 r;
r = __builtin_mips_precrq_ph_w (a, b); // r will be {0x1234, 0x1111}
```

Precision Reduce Two Fractional Words to Two Half-Words with Rounding and Saturation

```
v2q15 __builtin_mips_precrq_rs_ph_w (q31, q31);
#-----------------------------------------------------------------------------
Ex:
q31 a = {0x7000FFFF};
q31 b = {0x80000000};
v2q15 r;
r = __builtin_mips_precrq_rs_ph_w (a, b); // r will be {0x7001, 0x8000}
```

Precision Expand a Fractional Half-word to a Fractional Word Value

```
q31 __builtin_mips_preceq_w_phl (v2q15);
q31 __builtin_mips_preceq_w_phr (v2q15);
```

Note that the endianness affects the result.

```
#-----------------------------------------------------------------------------
Ex: /* Assume a big-endian CPU */
v2q15 a = {0x1234, 0x5678};
q31 r1, r2;
r1 = __builtin_mips_preceq_w_phl (a, b); // r1 will be 0x12340000
r2 = __builtin_mips_preceq_w_phr (a, b); // r2 will be 0x56780000
```

13.7.9.7 Using Intrinsics for 64-bit Accumulators

Extract a Value with Right Shift

```
i32 __builtin_mips_extr_w (a64, imm0_31);
i32 __builtin_mips_extr_w (a64, i32);
i32 __builtin_mips_extr_r_w (a64, imm0_31);
i32 __builtin_mips_extr_r_w (a64, i32);
i32 __builtin_mips_extr_rs_w (a64, imm0_31);
i32 __builtin_mips_extr_rs_w (a64, i32);
```

```
#----------------------------------------------------------------------------
Ex:
a64 ac1 = 0x8123456712345678;
i32 shift_amount = 31;
i32 r1, r2, r3, r4, r5, r6;
r1 = __builtin_mips_extr_w (ac1, 1); // r1 will be 0x891A2B3C
r2 = __builtin_mips_extr_w (ac1, shift_amount); // r2 will be 0x02468ACE
r3 = __builtin_mips_extr_r_w (ac1, 4); // r3 will be 0x71234568
r4 = __builtin_mips_extr_r_w (ac1, shift_amount); // r4 will be 0x02468ACE
r5 = __builtin_mips_extr_rs_w (ac1, 4); // r5 will be 0x80000000
r6 = __builtin_mips_extr_rs_w (ac1, shift_amount); // r6 will be 0x80000000
```

Extract Half-word with Right Shift and Saturate

```
i32 __builtin_mips_extr_s_h (a64, imm0_31);
i32 __builtin_mips_extr_s_h (a64, i32);
```

Note that the 16-bit result is sign-extended to a 32-bit result.

```
#----------------------------------------------------------------------------
Ex:
a64 ac1 = 0xFFFFF81230000000;
i32 shift_amount = 4;
i32 r1, r2;
r1 = __builtin_mips_extr_s_h (ac1, 28); // r1 will be 0xFFFF8123
r2 = __builtin_mips_extr_s_h (ac1, shift_amount); // r2 will be 0xFFFF8000
```

Extract Bit from an Arbitrary Position

```
i32 __builtin_mips_extp (a64, imm0_31);
i32 __builtin_mips_extp (a64, i32);
```

Note that the “imm0_31” + 1 bits between “POS” and “POS” - “imm0_31” are extracted and zero-extended to a 32-bit result. So, if X bits are extracted, X-1 should be used as the second parameter. The POS field can be set by using “__builtin_mips_wrdsp”.

```
#----------------------------------------------------------------------------
Ex:
a64 ac1 = 0x1234567887654321;
i32 r1, r2;
int the_size = 3;
__builtin_mips_wrdsp (35, 1); // Write 35 to the POS field
r1 = __builtin_mips_extp (ac1, 31); // r1 will be 0x88765432
r2 = __builtin_mips_extp (ac1, the_size); // r2 will be 0x8
```

Extract Bit from an Arbitrary Position and Decrement POS

```
i32 __builtin_mips_extpdp (a64, imm0_31);
i32 __builtin_mips_extpdp (a64, i32);
```

Note that this intrinsic is the same as the ones in Section , except that in addition, the POS field is decremented by the number of extracted bits, “imm0_31” + 1 or “i32” + 1.

```
#----------------------------------------------------------------------------
Ex:
a64 ac1 = 0x123456789ABCDEF0;
i32 r1, r2;
int the_size = 7;
__builtin_mips_wrdsp (35, 1); // Write 35 to the POS field
r1 = __builtin_mips_extpdp (ac1, 3); // r1 will be 0x8, and POS will be 31
```

```
r2 = __builtin_mips_extpdp (ac1, the_size); // r2 will be 0x9a, and POS will be 23
```

Shift an Accumulator Value

```
a64 __builtin_mips_shilo (a64, imm_n32_31);
a64 __builtin_mips_shilo (a64, i32);
```

Note that using the same a64 variable for the target and the first parameter could lead to better performance.

```
#----------------------------------------------------------------------------
Ex:
a64 ac1 = 0x1234567887654321;
int shift_amount = -8;
ac1 = __builtin_mips_shilo (ac1, 8); // ac1 will be 0x0012345678876543
ac1 = __builtin_mips_shilo (ac1, shift_amount); // ac1 will be 0x1234567887654300
```

Copy the LO to HI and a Value to LO and Increment POS by 32

```
a64 __builtin_mips_mthlip (a64, i32);
```

Note that using the same a64 variable for the target and the first parameter could lead to better performance.

```
#----------------------------------------------------------------------------
Ex:
a64 ac1 = 0x1234567887654321;
int b = 0x11112222
__builtin_mips_wrdsp (0, 1); // Write 0 to the POS field
ac1 = __builtin_mips_mthlip (ac1, b); // ac1 will be 0x8765432111112222,
                                      // and POS will be 32
```

13.7.9.8 Using Intrinsics for 32-bit Integers

Add and Set Carry/Add with Carry

```
i32 __builtin_mips_addsc (i32, i32);
i32 __builtin_mips_addwc (i32, i32);
```

Note that these two intrinsics can be used to add two 64-bit operands, each spread across two GPRs. The lower 32 bits are calculated first, then the carry from this addition is fed to the add of the upper 32 bit values.

```
#----------------------------------------------------------------------------
Ex:
int i1 = 0;
int i2 = 0xFFFFFFFF;
int j1 = 1;
int j2 = 1;
int r1, r2;
r2 = __builtin_mips_addsc (i2, j2); // r2 will be 0xFFFFFFFF+1 = 0 and C will be 1
r1 = __builtin_mips_addwc (i1, j1); // r1 will be 0 + 1 + 1(C) = 2
```

Modular Subtraction on an Index Value

```
i32 __builtin_mips_modsub (i32, i32);
```

Note that this intrinsic can be used to implement a circular buffer. The first parameter is the current index, that will be checked against zero. If the index is zero, the new index will be rolled back to the top of the buffer, assigned from the bit 23 to 8 of the second parameter. If the index is not zero, the new index will be decremented by the size of the element, assigned from the bit 7 to 0 of the second parameter.

```
#----------------------------------------------------------------------------
```

```
Ex:
int index = 20;
int element = 0x1402;
while (1)
{
  index = __builtin_mips_modsub (index, element);
}
/* ‘index’ will be 20, 18, 16, ..., 4, 2, 0, 20, 18, 16, ... */
```

Bit Reverse a Half-word

```
i32 __builtin_mips_bitrev (i32);
#--------------------------------------------------------------------------
Ex:
int a = 0x1234; // 0001 0010 0011 0100
int r;
r = __builtin_mips_bitrev (a); // r will be 0x2c48 (0010 1100 0100 1000)
```

Insert Bit Field Variable

```
i32 __builtin_mips_insv (i32, i32);
```

Note that using the same variable for the target and the first parameter could lead to better performance. This intrinsic inserts the value of the second parameter to the first parameter. The size to be extracted from the second parameter is specified in the SCOUNT field. The position to be inserted in the first parameter is specified in the POS field.

```
#--------------------------------------------------------------------------
Ex:
int a = 0x12345678;
int r = 0xFFFFFFFF;
__builtin_mips_wrdsp ((16<<7)+4, 3); // set SCOUNT to 16, and set POS to 4
r = __builtin_mips_insv (r, a); // The lowest 16-bit value of a is inserted to r
                                // at bit 4. r will be 0xFFF5678F.
```

Load Unsigned Byte/Halfword/Word Indexed

```
i32 __builtin_mips_lbux (void *, i32);
i32 __builtin_mips_lhx (void *, i32);
i32 __builtin_mips_lwx (void *, i32);
```

NOTES:
1, The first parameter is the base of the array, and the second parameter is the offset in byte.
2.The returned value is zero-extended for “__builtin_mips_lbux”, and sign-extended for “__builtin_mips_lhx” and “__builtin_mips_lwx” to 32-bit integers.

```
#--------------------------------------------------------------------------
Ex:
char array_a[100];
short array_b[100];
int array_c[100];
int offset = 20;
int r1, r2, r3;
r1 = __builtin_mips_lbux ((void *)array_a, offset);
r2 = __builtin_mips_lhx ((void *)array_b, offset);
r3 = __builtin_mips_lwx ((void *)array_c, offset);
```

Signed Multiply and Add

```
a64 __builtin_mips_madd (a64, i32, i32);
#---------------------------------------------------------------------------
Ex:
a64 a = 1;
i32 b = 2;
i32 c = -3;
a = __builtin_mips_madd (a, b, c); // a will be 1 + 2 * (-3) = -5
```

Unsigned Multiply and Add

```
a64 __builtin_mips_maddu (a64, ui32, ui32);
#---------------------------------------------------------------------------
a64 a = 1;
ui32 b = 2;
ui32 c = 3;
a = __builtin_mips_maddu (a, b, c); // a will be 1 + 2 * 3 = 7
```

Signed Multiply and Subtract

```
a64 __builtin_mips_msub (a64, i32, i32);
#---------------------------------------------------------------------------
Ex:
a64 a = 1;
i32 b = 2;
i32 c = -3;
a = __builtin_mips_msub (a, b, c); // a will be 1 - 2 * (-3) = 7
```

Unsigned Multiply and Subtract

```
a64 __builtin_mips_msubu (a64, ui32, ui32);
#---------------------------------------------------------------------------
Ex:
a64 a = 1;
ui32 b = 2;
ui32 c = 3;
a = __builtin_mips_msubu (a, b, c); // a will be 1 - 2 * 3 = -5
```

Signed Multiply

```
a64 __builtin_mips_mult (i32, i32);
#---------------------------------------------------------------------------
a64 a;
i32 b = 2;
i32 c = -3;
a = __builtin_mips_mullt (b, c); // a will be 2 * (-3) = -6
```

Unsigned Multiply

```
a64 __builtin_mips_multu (ui32, ui32);
#---------------------------------------------------------------------------
Ex:
a64 a;
u32 b = 2;
u32 c = 3;
```

```
a = __builtin_mips_mulltu (b, c); // a will be 2 * 3 = 6
```

Left Shift and Append Bits

```
i32 __builtin_mips_append (i32, i32, imm0_31); // DSPR2
#----------------------------------------------------------------------------
Ex:
i32 a = 0x8765ABCD;
i32 b = 0x12345678;
i32 r;
r = __builtin_mips_append (a, b, 4); // r will be 0x765ABCD8
```

Byte Align Contents from Two Registers

```
i32 __builtin_mips_balign (i32, i32, imm0_3); // DSPR2
#----------------------------------------------------------------------------
Ex:
i32 a = 0x8765ABCD;
i32 b = 0x12345678;
i32 r;
r = __builtin_mips_balign (a, b, 3); // r will be 0xCD123456
```

Right Shift and Prepend Bits

```
i32 __builtin_mips_prepend (i32, i32, imm0_31); // DSPR2
#----------------------------------------------------------------------------
Ex:
i32 a = 0x8765ABCD;
i32 b = 0x12345678;
i32 r;
r = __builtin_mips_prepend (a, b, 4); // r will be 0x88765ABC
```

13.7.9.9 Using Intrinsics for 16-bit Integers

Unsigned Add/Subtract with Optional Saturation

```
v2i16 __builtin_mips_addu_ph (v2i16, v2i16); // DSPR2
v2i16 __builtin_mips_addu_s_ph (v2i16, v2i16); // DSPR2
v2i16 __builtin_mips_subu_ph (v2i16, v2i16); // DSPR2
v2i16 __builtin_mips_subu_s_ph (v2i16, v2i16); // DSPR2
#----------------------------------------------------------------------------
Ex:
v2i16 a = {0x0000, 0x8000};
v2i16 b = {0x8000, 0x8000};
v2i16 r1, r2, r3, r4;
r1 = __builtin_mips_addu_ph (a, b); // r1 will be {0x8000, 0x0000}
r2 = __builtin_mips_addu_s_ph (a, b); // r2 will be {0x8000, 0xFFFF}
r3 = __builtin_mips_subu_ph (a, b); // r3 will be {0x8000, 0x0000}
r4 = __builtin_mips_subu_s_ph (a, b); // r4 will be {0x0000, 0x0000}
```

Dot Product with Accumulate/Subtract

```
a64 __builtin_mips_dpa_w_ph (a64, v2i16, v2i16); // DSPR2
a64 __builtin_mips_dps_w_ph (a64, v2i16, v2i16); // DSPR2
```

NOTES:
1. The result will be a 64-bit integer.
2. Using the same “a64” variable for both the target and the first parameter could result in better performance.

```
#----------------------------------------------------------------------------
Ex:
v2i16 a = {0x0001, 0x8000};
v2i16 b = {0x0002, 0x8000};
a64 ac1, ac2;
ac1 = ac2 = 0;
ac1 = __builtin_mips_dpa_w_ph (ac1, a, b); // ac1 will be 0 + 1*2 +
                                           // 0x40000000 = 0x0000000040000002
ac2 = __builtin_mips_dps_w_ph (ac2, a, b); // ac2 will be 0 - 1*2 -
                                           // 0x40000000 = 0xFFFFFFFFBFFFFFFE
```

Multiply with Optional Saturation

```
v2i16 __builtin_mips_mul_ph (v2i16, v2i16); // DSPR2
v2i16 __builtin_mips_mul_s_ph (v2i16, v2i16); // DSPR2
#----------------------------------------------------------------------------
Ex:
v2i16 a = {0x7FFF, 0x8000};
v2i16 b = {0x7FFF, 0x8000};
v2i16 r1, r2;
r1 = __builtin_mips_mul_ph (a, b); // r1 will be {0x0001, 0x0000}
r2 = __builtin_mips_mul_s_ph (a, b); // r2 will be {0x7FFF, 0x7FFF}
```

Multiply and Subtract and Accumulate

```
a64 __builtin_mips_mulsa_w_ph (a64, v2i16, v2i16); // DSPR2
```

NOTES:
1. The result will be a 64-bit integer.
2. The processor endianness affects the format of the result.
3. Using the same “a64” variable for the target and the first parameter could lead to better performance.

```
#----------------------------------------------------------------------------
Ex:
v2i16 a = {0x0001, 0x8000};
v2i16 b = {0x0002, 0x8000};
a64 ac1 = 0;
ac1 = __builtin_mips_mulsa_w_ph (ac1, a, b); // ac1 will be 0 + 1*2 -
                                               // 0x40000000 = 0xFFFFFFFFC0000002
```

Shift Right Logical

```
v2i16 __builtin_mips_shrl_ph (v2i16, imm0_15); // DSPR2
v2i16 __builtin_mips_shrl_ph (v2i16, i32); // DSPR2
#----------------------------------------------------------------------------
Ex:
v2i16 a = {0x8000, 0x4000};
v2i16 r1, r2;
int shift_amount = 4;
r1 = __builtin_mips_shrl_ph (a, 4); // r1 will {0x0800, 0x0400};
r2 = __builtin_mips_shrl_ph (a, shift_amount); // r2 will {0x0800, 0x0400};
```

Cross Dot Product with Accumulate/Subtract

```
a64 __builtin_mips_dpax_w_ph (a64, v2i16, v2i16); // DSPR2
a64 __builtin_mips_dpsx_w_ph (a64, v2i16, v2i16); // DSPR2
```

NOTES:
1. The result will be a 64-bit integer.
2. Using the same "a64" variable for both the target and the first parameter could result in better performance.

```
#----------------------------------------------------------------------------
Ex:
v2i16 a = {0x0001, 0x0003};
v2i16 b = {0x0002, 0x0004};
a64 ac1, ac2;
ac1 = ac2 = 0;
ac1 = __builtin_mips_dpax_w_ph (ac1, a, b);  // ac1 will be
                                             //0 + 1*4 + 2*3 = 0x000000000000000A
ac2 = __builtin_mips_dpsx_w_ph (ac1, a, b); // ac2 will be
                                            // 0 - 1*4 - 2*3 = 0xFFFFFFFFFFFFFFF6
```

13.7.9.10 Using Intrinsics for Mixed Data Types: 16-bit and 32-bit Integers

Precision Reduce Two Integer Words to Halfwords After a Right Shift with Optional Rounding

```
v2i16 __builtin_mips_precr_sra_ph_w (i32, i32, imm0_31); // DSPR2
v2i16 __builtin_mips_precr_sra_r_ph_w (i32, i32, imm0_31); // DSPR2
#----------------------------------------------------------------------------
Ex:
i32 a = 0x80000000;
i32 b = 0x7FFFFFFF;
v2i16 r1, r2;
r1 = __builtin_mips_precr_sra_ph_w (a, b, 4); // r1 will be {0x0000, 0xFFFF}
r2 = __builtin_mips_precr_sra_r_ph_w (a, b, 4); // r2 will be {0x0000, 0x0000}
```

13.8 Code Optimizations for the MIPS DSP-R2

This section provides information on writing efficient C programs using the MIPS32 DSP-R2 ASE. Note that the level of optimization is dependent on the specifics of how the particular GCC compiler works.

Programmers must select proper optimization levels to compile C code to suit their purposes. For example, for maximum speed: "-O3 -funroll-loops". For good speed with moderate code sizes: "-O2". For minimum code sizes: "-Os". Note that, to allow the GCC compiler to efficiently schedule instructions based on the latency information, programmers must supply correct architecture and CPU options.

13.8.1 The Use of Intrinsics Versus ASM Macros

The assembler has no knowledge of the pipeline and any code written using ASM macros will be treated as a single cycle latency instruction by the GCC compiler. This can lead to poor code scheduling and a lot of stalls in the resulting execution. On the other hand, the GCC compiler has knowledge of the pipeline latency of instructions and can schedule the DSP-R2 instructions correctly when programmers use intrinsics, that is "**__builtin_mips_***". Hence it is important to try to avoid the use of ASM macros whenever possible.

13.8.2 Using Accumulators

To access only HI or LO of an accumulator, programmers are recommended to use a union type as follows.

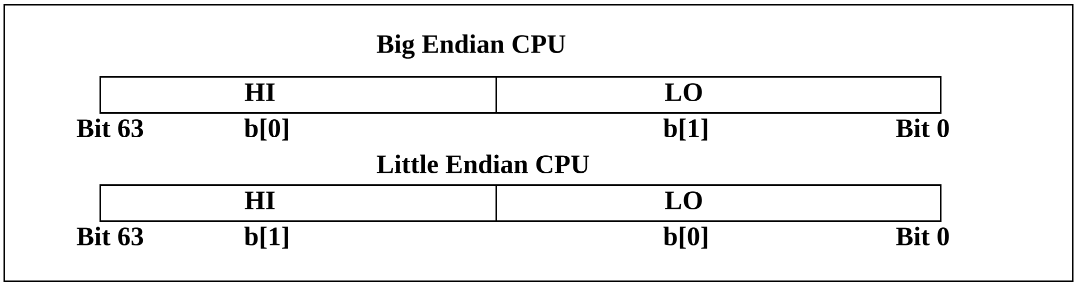
```
typedef union
{
  long long a;   // One 64-bit accumulator
  int b[2];      // 32-bit HI and LO
} a64_union;
```

Note that CPU endianness affects how to access the accumulator as shown in Figure 13.4. To access HI, b[0] is used in a Big Endian CPU, but b[1] is used in a Little Endian CPU. To access LO, b[1] is used in a Big Endian CPU, but b[0] is used in a Little Endian CPU.

```
Ex:
int test10 (long long a, v2q15 b, v2q15 c)
{
  a64_union temp;
  temp.a = __builtin_mips_dpaq_s_w_ph (a, b, c);
  return temp.b[0];  // Assume in a little-endian CPU we want to access LO.
}

# Generated Assembly
test10:
   mtlo   $4
   mthi   $5
   dpaq_s.w.ph  $ac0, $6, $7
   j      $31
   mflo   $2
```

Figure 13.4 Accessing HI and LO of Accumulators



13.8.3 Multiply “32-bit * 32-bit = 64-bit”

To multiply 32 bits by 32 bits to obtain a 64-bit result, we must cast the 32-bit integer to a 64-bit integer (long long) and then perform the multiplication operation as follows.

```
Ex:
long long test11 (int a, int b)
{
  return (long long) a * b;  // Same as (long long) a * (long long) b
                             // NOT the same as (long long) (a * b)
}

# Generated Assembly
test11:
    mult    $4,$5
    mflo    $2
    j       $31
    mfhi    $3
```

Combined with Section 13.8.2 we can multiply 32-bit by 32-bit integers and get the highest 32-bit result from HI as follows.

```
Ex:
int test12 (int a, int b)
{
  a64_union temp;
  temp.a = (long long) a * b;
  return temp.b[1];  // Assume a little-endian CPU
}

# Generated Assembly
test12:
    mult    $4,$5
    j       $31
    mfhi    $2
```

13.8.4 Multiply and Add “32-bit * 32-bit + 64-bit = 64-bit”

To perform multiplication and addition, we must cast the 32-bit integer to 64-bit (long long) and then perform multiplication and addition as follows.

```
Ex:
long long test13 (int a, int b, long long c)
{
  return c + (long long) a * b;
}

# Generated Assembly
test13:
    mtlo   $6
    mthi   $7
    madd   $4, $5
    mflo   $2
    j      $31
    mfhi   $3
```

13.8.5 Array Alignment and Data Layout

The GCC compiler provides a mechanism to specify the alignment of variables by using “__attribute__ ((aligned (*bytes*)))”. The alignment is important to loading or storing SIMD variables: “v4i8” and “v2q15”. If an array is aligned to a 4-byte boundary, that is, word-aligned, the GCC compiler can load or store four 8-bit data for v4i8 variables (or two 16-bit data for v2q15 variables) at a time using the load word class of instructions. The following example shows that when a char array A is aligned to a 4-byte boundary, we can cast this array to a v4i8 array and load four items to a v4i8 variable at a time by using the “lwx” instruction. However, if this char array A is not aligned to a 4-byte boundary, executing the following code will result in an address exception due to a mis-aligned load.

```
Ex: /* v4i8 Example */
char A[128] __attribute__ ((aligned (4)));
v4i8 test14 (int i)
{
  v4i8 a;
  v4i8 *myA = (v4i8 *)A;
  a = myA[i];
  return a;
}
# Generated Assembly
test14:
   lui     $2,%hi(A)
   sll     $4,$4,2
   addiu   $2,$2,%lo(A)
   j       $31
   lwx     $2,$2($4)
```

After SIMD data is loaded from memory into a register, it is best if the SIMD variables in the register are ready for use without requiring any rearrangement of the data. To avoid such data rearrangement which can reduce the benefit of parallelism, programmers must design their arrays with efficient data layout that is favorable for SIMD calculations.

13.8.6 GP-Relative Addressing

The GCC compiler provides an option "-G *num*" to put global or static data that is at most "*num*" bytes to the small data or BSS sections. This allows using only one instruction to access data via GP-relative addressing to improve the performance. Programmers can try to increase "*num*" to include more data into small data or BSS sections until these sections are full. Note that all ASEs should be compiled with the same "-G *num*". The following example shows how the GCC compiler accesses a 16-byte array. When compiling the example with "-G 4", calculating the base address of the array "C" needs two instructions: "`lui $3,%hi(C)`" and "`addiu $3,$3,%lo(C)`". But, when compiling with "-G 16" to put the whole array of "C" into the small data section, only one instruction "`addiu $3,$28,%gp_rel(C)`" is required to get the base address of "C".

```
Ex:
int C[4];
void test15 (int index, int value)
{
  C[index] = value;
}

# Generated Assembly when compiling with -G 4
test15:
   lui     $3,%hi(C)
   addiu   $3,$3,%lo(C)
   sll     $4,$4,2
   addu    $4,$4,$3
   j       $31
   sw      $5,0($4)
# -----------------------------------------------------------

# Generated Assembly when compiling with -G 16
test15:
   addiu   $3,$28,%gp_rel(C)
   sll     $4,$4,2
   addu    $4,$4,$3
   j       $31
   sw      $5,0($4)
```

13.8.7 Fixed Registers and Register Variables

Register usage is defined by the Application Binary Interface (ABI). For example, the ABI defines that some registers are caller-saved, some are callee-saved, and a few registers are fixed (or called global) and not saved at all. When conforming to the ABI, functions are guaranteed to work with each other.

However, in very special cases where performance may be very critical, programmers may want to improve performance by avoiding the saving and restoring of registers and hence violating the ABI convention. This undertaking should be taken with caution and not normally recommended as general practice. The GCC compiler allows programmers to treat a register as fixed by using the command-line option: "-ffixed-*reg*" where *reg* must be the name of a register. When a register is fixed, the register allocation process does not touch the fixed register.

For example, the ABI defines that four accumulators ($ac0 - $ac3) are caller-saved registers, but programmers may want to dedicate one accumulator, $ac1, for a special purpose. Note that because $ac1is a 64-bit register that consists of $ac1hi and $ac1lo, "-ffixed-\$ac1hi -ffixed-\$ac1lo" is specified in the command-line options to fix HI and LO of $ac1.

The following example demonstrates that under the original ABI, the GCC compiler register allocator will allocate 64-bit variables to all accumulators. However, when $ac1 is specified to be fixed, The GCC compiler only allocates 64-bit variables to $ac0, $ac2, and $ac3.

```
Ex:
typedef long long a64;
typedef short v2q15 __attribute__ ((vector_size(4)));
void test16 (a64 a[4], v2q15 b[4], v2q15 c[4])
{
  a[0] = __builtin_mips_dpaq_s_w_ph (a[0], b[0], c[0]);
  a[1] = __builtin_mips_dpaq_s_w_ph (a[1], b[1], c[1]);
  a[2] = __builtin_mips_dpaq_s_w_ph (a[2], b[2], c[2]);
  a[3] = __builtin_mips_dpaq_s_w_ph (a[3], b[3], c[3]);
}

# Generated Assembly without using "-ffixed-\$ac1hi --fixed-\$ac1lo"
# Note that $ac0, $ac1, $ac2, and $ac3 are all used.
test16:
        lw      $15,4($4)
        lw      $14,0($4)
        lw      $13,12($4)
        lw      $10,8($4)
        lw      $9,20($4)
        lw      $8,16($4)
        lw      $3,28($4)
        lw      $7,24($4)
        lw      $2,0($6)
        lw      $12,12($5)
        lw      $11,12($6)
        mtlo    $15,$ac1
        mthi    $14,$ac1
        mtlo    $13,$ac2
        lw      $25,0($5)
        lw      $15,4($5)
        lw      $24,4($6)
        lw      $13,8($5)
        lw      $14,8($6)
        mthi    $10,$ac2
        mtlo    $9,$ac3
        mthi    $8,$ac3
        mtlo    $3
        mthi    $7
        dpaq_s.w.ph     $ac1,$25,$2
        dpaq_s.w.ph     $ac2,$15,$24
        dpaq_s.w.ph     $ac3,$13,$14
        dpaq_s.w.ph     $ac0,$12,$11
        mflo    $10
        mfhi    $9
        mflo    $8,$ac1
        mfhi    $7,$ac1
        mflo    $6,$ac2
        mfhi    $5,$ac2
        mflo    $3,$ac3
        mfhi    $2,$ac3
        sw      $10,28($4)
        sw      $9,24($4)
        sw      $8,4($4)
        sw      $7,0($4)
        sw      $6,12($4)
        sw      $5,8($4)
        sw      $3,20($4)
        j       $31
        sw      $2,16($4)
# ---------------------------------------------------------------------------

# Generated Assembly when using “-ffixed-\$ac1hi --fixed-\$ac1lo”
# Note that $ac0, $ac2, and $ac3 are used.  But, $ac1 is not touched at all by the
# compiler.
test16:
        lw      $3,4($4)
        lw      $2,0($4)
        lw      $25,12($4)
        lw      $24,8($4)
        lw      $9,20($4)
        lw      $8,16($4)
        lw      $15,12($5)
        lw      $13,0($5)
        lw      $11,0($6)
        lw      $12,4($5)
        lw      $7,4($6)
        lw      $10,8($5)
        lw      $5,8($6)
        mtlo    $3,$ac2
        mthi    $2,$ac2
        lw      $3,28($4)
        lw      $2,24($4)
        mtlo    $25,$ac3
        mthi    $24,$ac3
        mtlo    $9
        mthi    $8
        dpaq_s.w.ph     $ac2,$13,$11
        lw      $14,12($6)
        dpaq_s.w.ph     $ac3,$12,$7
        dpaq_s.w.ph     $ac0,$10,$5
        mflo    $9
        mfhi    $8
        mtlo    $3
        mthi    $2
        dpaq_s.w.ph     $ac0,$15,$14
        mflo    $25
        mfhi    $24
        mflo    $6,$ac2
        mfhi    $5,$ac2
        mflo    $3,$ac3
        mfhi    $2,$ac3
        sw      $25,28($4)
        sw      $24,24($4)
        sw      $6,4($4)
        sw      $5,0($4)
        sw      $3,12($4)
        sw      $2,8($4)
        sw      $9,20($4)
        j       $31
        sw      $8,16($4)
```

To use a fixed register, programmers must associate a register variable with the explicit name of the fixed register. For example, when $ac1 is fixed, we can declare “register a64 MYACC ASM (“$ac1lo”)” in a Little Endian CPU, or “register a64 MYACC ASM (“$ac1hi”)” in a Big Endian CPU. Then, the global variable “MYACC” is ready to be used across all functions via directly accessing $ac1.

The following example shows that when no global register variable is used, The GCC compiler needs to load or store a 64-bit global variable from or to memory.

```
Ex:
typedef long long a64;
typedef short v2q15 __attribute__ ((vector_size(4)));
a64 MYACC;
void test17 (v2q15 b, v2q15 c)
{
  MYACC = __builtin_mips_dpaq_s_w_ph (MYACC, b, c);
}

# Generated Assembly
test17:
        lw      $2,%gp_rel(MYACC)($28)
        lw      $3,%gp_rel(MYACC+4)($28)
        mtlo    $2
        mthi    $3
        dpaq_s.w.ph     $ac0,$4,$5
        mflo    $2
        mfhi    $3
        sw      $2,%gp_rel(MYACC)($28)
        j       $31
        sw      $3,%gp_rel(MYACC+4)($28)
```

However, when a register variable is used for a global variable, the overhead of storing and loading to and from memory is eliminated as follows, reducing the above 10 instructions to only 2.

```
Ex:
typedef long long a64;
typedef short v2q15 __attribute__ ((vector_size(4)));
register a64 MYACC asm ("$ac1lo"); /* Assume a little-endian CPU */
void test18 (v2q15 b, v2q15 c)
{
  MYACC = __builtin_mips_dpaq_s_w_ph (MYACC, b, c);
}

# Generated Assembly by
# “sde-gcc -mips32r2 -mdsp -O4 -S -ffixed-\$ac1hi --fixed-\$ac1lo -EL 18.c”
test18:
        j       $31
        dpaq_s.w.ph     $ac1,$4,$5
```

There are a few things to note when using the technique of fixing registers for global variables.

1. When fixing accumulators, because $ac0 is the original HI and LO registers for multiplication and division instructions in MIPS32, $ac0 cannot be fixed by using “-ffixed-hi -ffixed-lo”. The rest of the accumulators, that is $ac1, $ac2, and $ac3 can be fixed.

2. When fixing $ac1, $ac2, or $ac3, programmers must ensure that no third-party or library functions that can clobber $ac1, $ac2 or $ac3 are called between accessing fixed accumulators. To practice safe programming methods, it is probably advisable to restrict the use of fixed accumulators inside an optimized kernel that consist of only internal functions.

3. The technique of fixing registers for use as global variables can be directly applied to callee-saved registers that are $16 to $23 (s0 to s7). Programmers do not need to change s0 to s7 to be fixed registers.

13.8.8 Conditional Moves

Typically conditional move instructions are used instead of branch instructions to avoid the penalty from branch delay slots and mis-predicted branches. For example, the GCC compiler can generate conditional move instructions for simple C code as follows.

```
Ex:
int test19 (int true_value, int false_value, int cond)
{
if (cond)
  return true_value;
else
  return false_value;
}

# Generated Assembly
test19:
        move    $2,$4
        j       $31
        movz    $2,$5,$6
# ------------------------------------------------------------

Ex:
int test20 (int true_value, int false_value, int cond)
{
  return cond ? true_value : false_value;
}

# Generated Assembly
test20:
        move    $2,$4
        j       $31
        movz    $2,$5,$6
```

However, for complicated C code, the GCC compiler may not recognize the C patterns to generate conditional move instructions. Programmers can then use ASM macros to force the GCC compiler to use conditional move instructions. The following example shows how to use an ASM macro for a “movz” instruction. First, we need to assign a value “true_value” (when the condition is true) to a resultant variable “result”. Then, we pass “result”, “false_value” and “cond” to the ASM macro of “movz”. Note that the ASM macro uses “+d” for the output format, because the output register is also used as an input register.

```
Ex:
int test21 (int true_value, int false_value, int cond)
{
  int result = true_value;
  asm ("movz %0, %1, %2": "+d" (result): "d" (false_value), "d" (cond));
  return result;
}

# Generated Assembly
test21:
 #APP
        movz $4, $5, $6
 #NO_APP
        .set    noreorder
        .set    nomacro
        j       $31
        move    $2,$4
```

13.9 Programming Examples

This section describes the programming example for a 16-point FIR filter in three ways:

- FIR filter in traditional C code without SIMD variables and DSP-R2 intrinsics
- Hand-tuned assembly version
- FIR filter in efficient C code

13.9.1 The FIR Filter in Traditional C

The following C code implements a 16-point FIR filter without using SIMD variables and DSP-R2 intrinsics. The arrays of “coeffs” and “delay” store sixteen Q15 coefficients and sixteen Q15 delayed inputs.

```
Ex:
int i;
short x, y;
long long ac0 = 0;
for (i = 0; i < 16; i++)
{
   x = coeffs[i];
   y = delay[i];
   if ((unsigned short) x == 0x8000 && (unsigned short) y == 0x8000)
      ac0 += 0x7fffffff;
   else
      ac0 += ((x * y) << 1);
}
```

Inside a loop, a saturation condition needs to be detected when both values of “coeffs” and “delay” are 0x8000 (-1 in Q15). Moreover, to perform “Q15 x Q15” multiplication, a left shift is required after integer multiplication. This version of the FIR filter takes 536 cycles to calculate one result. (The tools used for this experiment are SDE 6.03.00-rc3 and MIPSsim 4.6.23.) The traditional C code produces inefficient binary code, so DSP programmers would write it in assembly code.

13.9.2 The FIR Filter in Hand-Tuned Assembly

DSP programmers pack two coefficients to a register and pack two delayed inputs to a register, so a SIMD DSP-R2 instruction “dpaq_s.w.ph” that performs saturation and “Q15 x Q15” multiplication can be applied efficiently. Instruction scheduling is performed by hand to avoid pipeline stalls. This FIR implementation can generate one result in 39 cycles which are much faster than the traditional C version in Section 13.9.1. The hand-tuned assembly code for the FIR filter is as follows.

```
Ex:
    mult  $0, $0
    lw    $8, 0($5)
    lw    $10, 0($6)
    lw    $9, 4($5)
    lw    $11, 4($6)
    dpaq_s.w.ph $ac0, $8, $10
    dpaq_s.w.ph $ac0, $9, $11
    lw    $12, 8($5)
    lw    $10, 8($6)
    lw    $13, 12($5)
    lw    $11, 12($6)
    dpaq_s.w.ph $ac0, $12, $10
    dpaq_s.w.ph $ac0, $13, $11
    lw    $14, 16($5)
    lw    $10, 16($6)
    lw    $15, 20($5)
    lw    $11, 20($6)
    dpaq_s.w.ph $ac0, $14, $10
    dpaq_s.w.ph $ac0, $15, $11
    lw    $16, 24($5)
    lw    $10, 24($6)
    lw    $4, 28($5)
    lw    $11, 28($6)
    dpaq_s.w.ph $ac0, $16, $10
    dpaq_s.w.ph $ac0, $4, $11
```

13.9.3 The FIR Filter in Efficient C

Although the hand-tuned assembly code yields good performance, it requires the programmer to manually do register allocation and code scheduling. A compromise is to write C code that uses SIMD variables and DSP-R2 intrinsics as shown.

```
Ex:
    v2q15 *my_delay = (v2q15 *)delay;
    v2q15 *my_coeffs = (v2q15 *)coeffs;
    long long ac0 = 0;
    ac0 = __builtin_mips_dpaq_s_w_ph (ac0, my_delay[0], my_coeffs[0]);
    ac0 = __builtin_mips_dpaq_s_w_ph (ac0, my_delay[1], my_coeffs[1]);
    ac0 = __builtin_mips_dpaq_s_w_ph (ac0, my_delay[2], my_coeffs[2]);
    ac0 = __builtin_mips_dpaq_s_w_ph (ac0, my_delay[3], my_coeffs[3]);
    ac0 = __builtin_mips_dpaq_s_w_ph (ac0, my_delay[4], my_coeffs[4]);
    ac0 = __builtin_mips_dpaq_s_w_ph (ac0, my_delay[5], my_coeffs[5]);
    ac0 = __builtin_mips_dpaq_s_w_ph (ac0, my_delay[6], my_coeffs[6]);
```

```
        ac0 = __builtin_mips_dpaq_s_w_ph (ac0, my_delay[7], my_coeffs[7]);
```

This C code does not look as clean or as readable as the traditional C version in Section 13.9.1, but it is efficient and calculates one result in 42 cycles which is only 7.69% slower than the hand-tuned assembly version in Section 13.9.2. Compared to the hand-tuned assembly code, the efficient C code has three significant advantages as follows.

1. Register allocation is done by the compiler.

2. Code scheduling is done by the compiler.

3. Load and store of SIMD data is taken care of by the compiler.

Other DSP kernels can similarly benefit from C code.

13.10 MIPS32 DSP-R2 Intrinsics

This section lists the MIPS32 DSP-R2 instrinsics. Note that some parameters of intrinsics are immediate types. Programmers must pass a constant that is within the specific range in order to invoke these intrinsics.

13.10.1 Immediate Intrinsics

The immediate types are as follows:

```
imm0_3: the parameter must be a constant in the range 0 to 3.
imm0_7: the parameter must be a constant in the range 0 to 7.
imm0_15: the parameter must be a constant in the range 0 to 15.
imm0_31: the parameter must be a constant in the range 0 to 31.
imm0_63: the parameter must be a constant in the range 0 to 63.
imm0_255: the parameter must be a constant in the range 0 to 255.
imm_n512_511: the parameter must be a constant in the range -512 to 511.
imm_n32_31: the parameter must be a constant in the range -32 to 31.
```

13.10.2 Intrinsics for DSPControl Register

```
void __builtin_mips_wrdsp (i32, imm0_63);
i32 __builtin_mips_rddsp (imm0_63);
i32 __builtin_mips_bposge32 ();
```

13.10.3 Intrinsics for Signed and Unsigned 8-bit Integers

```
v4i8 __builtin_mips_addu_qb (v4i8, v4i8);
v4i8 __builtin_mips_addu_s_qb (v4i8, v4i8);
v4i8 __builtin_mips_subu_qb (v4i8, v4i8);
v4i8 __builtin_mips_subu_s_qb (v4i8, v4i8);
i32 __builtin_mips_raddu_w_qb (v4i8);
v4i8 __builtin_mips_shll_qb (v4i8, imm0_7);
v4i8 __builtin_mips_shll_qb (v4i8, i32);
v4i8 __builtin_mips_shrl_qb (v4i8, imm0_7);
v4i8 __builtin_mips_shrl_qb (v4i8, i32);
a64 __builtin_mips_dpau_h_qbl (a64, v4i8, v4i8);
a64 __builtin_mips_dpau_h_qbr (a64, v4i8, v4i8);
a64 __builtin_mips_dpsu_h_qbl (a64, v4i8, v4i8);
a64 __builtin_mips_dpsu_h_qbr (a64, v4i8, v4i8);
```

```
v4i8 __builtin_mips_repl_qb (imm0_255);
v4i8 __builtin_mips_repl_qb (i32);
void __builtin_mips_cmpu_eq_qb (v4i8, v4i8);
void __builtin_mips_cmpu_lt_qb (v4i8, v4i8);
void __builtin_mips_cmpu_le_qb (v4i8, v4i8);
i32 __builtin_mips_cmpgu_eq_qb (v4i8, v4i8);
i32 __builtin_mips_cmpgu_lt_qb (v4i8, v4i8);
i32 __builtin_mips_cmpgu_le_qb (v4i8, v4i8);
i32 __builtin_mips_cmpgdu_eq_qb (v4i8, v4i8); // DSPR2
i32 __builtin_mips_cmpgdu_lt_qb (v4i8, v4i8); // DSPR2
i32 __builtin_mips_cmpgdu_le_qb (v4i8, v4i8); // DSPR2
v4i8 __builtin_mips_pick_qb (v4i8, v4i8);
v4i8 __builtin_mips_absq_s_qb (v4i8); // DSPR2
v4i8 __builtin_mips_adduh_qb (v4i8, v4i8); // DSPR2
v4i8 __builtin_mips_adduh_r_qb (v4i8, v4i8); // DSPR2
v4i8 __builtin_mips_shra_qb (v4i8, imm0_7); // DSPR2
v4i8 __builtin_mips_shra_r_qb (v4i8, imm0_7); // DSPR2
v4i8 __builtin_mips_shra_qb (v4i8, i32); // DSPR2
v4i8 __builtin_mips_shra_r_qb (v4i8, i32); // DSPR2
v4i8 __builtin_mips_subuh_qb (v4i8, v4i8); // DSPR2
v4i8 __builtin_mips_subuh_r_qb (v4i8, v4i8); // DSPR2
```

13.10.4 Intrinsics for Q15

```
v2q15 __builtin_mips_addq_ph (v2q15, v2q15);
v2q15 __builtin_mips_addq_s_ph (v2q15, v2q15);
v2q15 __builtin_mips_subq_ph (v2q15, v2q15);
v2q15 __builtin_mips_subq_s_ph (v2q15, v2q15);
v2q15 __builtin_mips_absq_s_ph (v2q15);
v2q15 __builtin_mips_shll_ph (v2q15, imm0_15);
v2q15 __builtin_mips_shll_ph (v2q15, i32);
v2q15 __builtin_mips_shll_s_ph (v2q15, imm0_15);
v2q15 __builtin_mips_shll_s_ph (v2q15, i32);
v2q15 __builtin_mips_shra_ph (v2q15, imm0_15);
v2q15 __builtin_mips_shra_ph (v2q15, i32);
v2q15 __builtin_mips_shra_r_ph (v2q15, imm0_15);
v2q15 __builtin_mips_shra_r_ph (v2q15, i32);
v2q15 __builtin_mips_mulq_rs_ph (v2q15, v2q15);
a64 __builtin_mips_dpaq_s_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpsq_s_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_mulsaq_s_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_maq_s_w_phl (a64, v2q15, v2q15);
a64 __builtin_mips_maq_s_w_phr (a64, v2q15, v2q15);
a64 __builtin_mips_maq_sa_w_phl (a64, v2q15, v2q15);
a64 __builtin_mips_maq_sa_w_phr (a64, v2q15, v2q15);
q31 __builtin_mips_muleq_s_w_phl (v2q15, v2q15);
q31 __builtin_mips_muleq_s_w_phr (v2q15, v2q15);
v2q15 __builtin_mips_repl_ph (imm_n512_511);
v2q15 __builtin_mips_repl_ph (i32);
void __builtin_mips_cmp_eq_ph (v2q15, v2q15);
void __builtin_mips_cmp_lt_ph (v2q15, v2q15);
void __builtin_mips_cmp_le_ph (v2q15, v2q15);
v2q15 __builtin_mips_pick_ph (v2q15, v2q15);
v2q15 __builtin_mips_packrl_ph (v2q15, v2q15);
v2q15 __builtin_mips_mulq_s_ph (v2q15, v2q15); // DSPR2
v2q15 __builtin_mips_addqh_ph (v2q15, v2q15); // DSPR2
v2q15 __builtin_mips_addqh_r_ph (v2q15, v2q15); // DSPR2
```

```
v2q15 __builtin_mips_subqh_ph (v2q15, v2q15); // DSPR2
v2q15 __builtin_mips_subqh_r_ph (v2q15, v2q15); // DSPR2
a64 __builtin_mips_dpaqx_s_w_ph (a64, v2q15, v2q15); // DSPR2
a64 __builtin_mips_dpaqx_sa_w_ph (a64, v2q15, v2q15); // DSPR2
a64 __builtin_mips_dpsqx_s_w_ph (a64, v2q15, v2q15); // DSPR2
a64 __builtin_mips_dpsqx_sa_w_ph (a64, v2q15, v2q15); // DSPR2
```

13.10.5 Intrinsics for Q31

```
q31 __builtin_mips_addq_s_w (q31, q31);
q31 __builtin_mips_subq_s_w (q31, q31);
q31 __builtin_mips_absq_s_w (q31);
q31 __builtin_mips_shll_s_w (q31, imm0_31);
q31 __builtin_mips_shll_s_w (q31, i32);
q31 __builtin_mips_shra_r_w (q31, imm0_31);
q31 __builtin_mips_shra_r_w (q31, i32);
a64 __builtin_mips_dpaq_sa_l_w (a64, q31, q31);
a64 __builtin_mips_dpsq_sa_l_w (a64, q31, q31);
q31 __builtin_mips_mulq_rs_w (q31, q31); // DSPR2
q31 __builtin_mips_mulq_s_w (q31, q31); // DSPR2
q31 __builtin_mips_addqh_w (q31, q31); // DSPR2
q31 __builtin_mips_addqh_r_w (q31, q31); // DSPR2
q31 __builtin_mips_subqh_w (q31, q31); // DSPR2
q31 __builtin_mips_subqh_r_w (q31, q31); // DSPR2
```

13.10.6 Intrinsics for Mixed Data Types: 8-bit Integers and Q15/16-bit Integers

```
v4i8 __builtin_mips_precrq_qb_ph (v2q15, v2q15);
v4i8 __builtin_mips_precrqu_s_qb_ph (v2q15, v2q15);
v4i8 __builtin_mips_precr_qb_ph (v2i16, v2i16); // DSPR2
v2q15 __builtin_mips_precequ_ph_qbl (v4i8);
v2q15 __builtin_mips_precequ_ph_qbr (v4i8);
v2q15 __builtin_mips_precequ_ph_qbla (v4i8);
v2q15 __builtin_mips_precequ_ph_qbra (v4i8);
v2q15 __builtin_mips_preceu_ph_qbl (v4i8);
v2q15 __builtin_mips_preceu_ph_qbr (v4i8);
v2q15 __builtin_mips_preceu_ph_qbla (v4i8);
v2q15 __builtin_mips_preceu_ph_qbra (v4i8);
v2q15 __builtin_mips_muleu_s_ph_qbl (v4i8, v2q15);
v2q15 __builtin_mips_muleu_s_ph_qbr (v4i8, v2q15);
```

13.10.7 Intrinsics for Mixed Data Types: Q15 and Q31

```
v2q15 __builtin_mips_precrq_ph_w (q31, q31);
v2q15 __builtin_mips_precrq_rs_ph_w (q31, q31);
q31 __builtin_mips_preceq_w_phl (v2q15);
q31 __builtin_mips_preceq_w_phr (v2q15);
```

13.10.8 Intrinsics for 64-bit Accumulators

```
i32 __builtin_mips_extr_w (a64, imm0_31);
i32 __builtin_mips_extr_w (a64, i32);
i32 __builtin_mips_extr_r_w (a64, imm0_31);
i32 __builtin_mips_extr_r_w (a64, i32);
i32 __builtin_mips_extr_rs_w (a64, imm0_31);
```

```
i32 __builtin_mips_extr_rs_w (a64, i32);
i32 __builtin_mips_extr_s_h (a64, imm0_31);
i32 __builtin_mips_extr_s_h (a64, i32);
i32 __builtin_mips_extp (a64, imm0_31);
i32 __builtin_mips_extp (a64, i32);
i32 __builtin_mips_extpdp (a64, imm0_31);
i32 __builtin_mips_extpdp (a64, i32);
a64 __builtin_mips_shilo (a64, imm_n32_31);
a64 __builtin_mips_shilo (a64, i32);
a64 __builtin_mips_mthlip (a64, i32);
```

13.10.9 Intrinsics for 32-bit Integers

```
i32 __builtin_mips_addsc (i32, i32);
i32 __builtin_mips_addwc (i32, i32);
i32 __builtin_mips_modsub (i32, i32);
i32 __builtin_mips_bitrev (i32);
i32 __builtin_mips_insv (i32, i32);
i32 __builtin_mips_lbux (void *, i32);
i32 __builtin_mips_lhx (void *, i32);
i32 __builtin_mips_lwx (void *, i32);
i32 __builtin_mips_append (i32, i32, imm0_31); // DSPR2
i32 __builtin_mips_balign (i32, i32, imm0_3); // DSPR2
i32 __builtin_mips_prepend (i32, i32, imm0_31); // DSPR2
a64 __builtin_mips_madd (a64, i32, i32);
a64 __builtin_mips_maddu (a64, ui32, ui32);
a64 __builtin_mips_msub (a64, i32, i32);
a64 __builtin_mips_msubu (a64, ui32, ui32);
a64 __builtin_mips_mult (i32, i32);
a64 __builtin_mips_multu (ui32, ui32);
```

13.10.10 Intrinsics for 16-bit Integers

```
v2i16 __builtin_mips_addu_ph (v2i16, v2i16); // DSPR2
v2i16 __builtin_mips_addu_s_ph (v2i16, v2i16); // DSPR2
v2i16 __builtin_mips_subu_ph (v2i16, v2i16); // DSPR2
v2i16 __builtin_mips_subu_s_ph (v2i16, v2i16); // DSPR2
a64 __builtin_mips_dpa_w_ph (a64, v2i16, v2i16); // DSPR2
a64 __builtin_mips_dps_w_ph (a64, v2i16, v2i16); // DSPR2
v2i16 __builtin_mips_mul_ph (v2i16, v2i16); // DSPR2
v2i16 __builtin_mips_mul_s_ph (v2i16, v2i16); // DSPR2
a64 __builtin_mips_mulsa_w_ph (a64, v2i16, v2i16); // DSPR2
v2i16 __builtin_mips_shrl_ph (v2i16, imm0_15); // DSPR2
v2i16 __builtin_mips_shrl_ph (v2i16, i32); // DSPR2
a64 __builtin_mips_dpax_w_ph (a64, v2i16, v2i16); // DSPR2
a64 __builtin_mips_dpsx_w_ph (a64, v2i16, v2i16); // DSPR2
```

13.10.11 Intrinsics for Mixed Data Types: 16-bit and 32-bit Integers

```
v2i16 __builtin_mips_precr_sra_ph_w (i32, i32, imm0_31); // DSPR2
v2i16 __builtin_mips_precr_sra_r_ph_w (i32, i32, imm0_31); // DSPR2
```

13.11 DSP-R2 ASE Instruction Groups

The following tables list the DSP-R2 instructions per function group. The input and output data type for each instruction is included, as well as the intended application. Refer to Section 13.12, "Listing of DSP-R2 ASE Instruction Groups" for an alphabetical listing of DSP-R2 instructions and associated links to the corresponding instruction. Refer to Section “Repeat rate is measured as number of independent instructions that can be sent in 1 cycle.” for a definition and encoding of each individual DSP-R2 instruction.

Table 13.4 through Table 13.11 in this section list all the instructions in the DSP-R2 ASE. In each table below, the column entitled “Writes GPR / ac / DSPControl“, indicates the explicit write performed by each instruction. This column indicates the writing of a field in the *DSPControl* register other than the *ouflag* field (which is written by a large number of instructions).

Table 13.4 List of Instructions in MIPS® DSP-R2 ASE in Arithmetic Sub-class

Instruction Mnemonics	Input Data Type	Output Data Type	Writes GPR / ac / DSPControl	App	Description
ADDQ.PH rd,rs,rt ADDQ_S.PH rd,rs,rt	Pair Q15	Pair Q15	GPR	VoIP SoftM	Element-wise addition of two vectors of Q15 fractional values, with optional saturation.
ADDQ_S.W rd,rs,rt	Q31	Q31	GPR	Audio	Add two Q31 fractional values with saturation.
ADDU.QB rd,rs,rt ADDU_S.QB rd,rs,rt	Quad Unsigned Byte	Quad Unsigned Byte	GPR	Video	Element-wise addition of vectors of four unsigned byte values. Results may be optionally saturated to 255.
ADDUH.QB rd,rs,rt ADDUH_R.QB rd,rs,rt	Quad Unsigned Byte	Quad Unsigned Byte	GPR	Video	Element-wise addition of vectors of four unsigned byte values, halving each result by right-shifting by one bit position. Results may be optionally rounded up in the least-significant bit.
ADDU.PH rd,rs,rt ADDU_S.PH rd,rs,rt	Pair Unsigned Halfword	Pair Unsigned Halfword	GPR	Video	Element-wise addition of vectors of two unsigned halfword values, with optional saturation on overflow.
ADDQH.PH rd,rs,rt ADDQH_R.PH rd,rs,rt	Pair Signed Halfword	Pair Signed Halfword	GPR	Misc	Element-wise addition of vectors of two signed halfword values, halving each result with right-shifting by one bit position. Results may be optionally rounded up in the least-significant bit.
ADDQH.W rd,rs,rt ADDQH_R.W rd,rs,rt	Signed Word	Signed Word	GPR	Misc	Add two signed word values, halving the result with right-shifting by one bit position. Result may be optionally rounded up in the least-significant bit.
SUBQ.PH rd,rs,rt SUBQ_S.PH rd,rs,rt	Pair Q15	Pair Q15	GPR	VoIP	Element-wise subtraction of two vectors of Q15 fractional values, with optional saturation.
SUBQ_S.W rd,rs,rt	Q31	Q31	GPR	Audio	Subtraction with Q31 fractional values, with saturation.

Table 13.4 List of Instructions in MIPS® DSP-R2 ASE in Arithmetic Sub-class *(continued)*

Instruction Mnemonics	Input Data Type	Output Data Type	Writes GPR / ac / DSPControl	App	Description
SUBU.QB rd,rs,rt SUBU_S.QB rd,rs,rt	Quad Unsigned Byte	Quad Unsigned Byte	GPR	Video	Element-wise subtraction of unsigned byte values, with optional unsigned saturation.
SUBUH.QB rd,rs,rt SUBUH_R.QB rd,rs,rt	Quad Unsigned Byte	Quad Unsigned Byte	GPR	Video	Element-wise subtraction of unsigned byte values, shifting the results right one bit position (halving). The results may be optionally rounded up by adding 1 to each result at the most-significant discarded bit position before shifting.
SUBU.PH rd,rs,rt SUBU_S.PH rd,rs,rt	Pair Unsigned Halfword	Pair Unsigned Halfword	GPR	Video	Element-wise subtraction of vectors of two unsigned halfword values, with optional saturation on overflow.
SUBQH.PH rd,rs,rt SUBQH_R.PH rd,rs,rt	Pair Signed Halfword	Pair Signed Halfword	GPR	Misc	Element-wise subtraction of vectors of two signed halfword values, halving each result with right-shifting by one bit position. Results may be optionally rounded up in the least-significant bit.
SUBQH.W rd,rs,rt SUBQH_R.W rd,rs,rt	Signed Word	Signed Word	GPR	Misc	Subtract two signed word values, halving the result with right-shifting by one bit position. Result may be optionally rounded up in the least-significant bit.
ADDSC rd,rs,rt	Signed Word	Signed Word	GPR & *DSPControl*	Audio	Add two signed words and set the carry bit in the *DSPControl* register.
ADDWC rd,rs,rt	Signed Word	Signed Word	GPR	Audio	Add two signed words with the carry bit from the *DSPControl* register.
MODSUB rd,rs,rt	Signed Word	Signed Word	GPR	Misc	Modulo addressing support: update a byte index into a circular buffer by subtracting a specified decrement (in bytes) from the index, resetting the index to a specified value if the subtraction results in underflow.
RADDU.W.QB rd,rs	Quad Unsigned Byte	Unsigned Word	GPR	Misc	Reduce (add together) the 4 unsigned byte values in *rs*, zero-extending the sum to32 bits before writing to the destination register. For example, if all 4 input values are 0x80 (decimal 128), then the result in *rd* is 0x200 (decimal 512).
ABSQ_S.QB rd,rt	Quad Q7	Quad Q7	GPR	Misc	Find the absolute value of each of four Q7 fractional byte elements in the source register, saturating values of -1.0 to the maximum positive Q7 fractional value.
ABSQ_S.PH rd,rt	Pair Q15	Pair Q15	GPR	Misc	Find the absolute value of each of two Q15 fractional halfword elements in the source register, saturating values of -1.0 to the maximum positive Q15 fractional value.

Table 13.4 List of Instructions in MIPS® DSP-R2 ASE in Arithmetic Sub-class *(continued)*

Instruction Mnemonics	Input Data Type	Output Data Type	Writes GPR / ac / DSPControl	App	Description
ABSQ_S.W rd,rt	Q31	Q31	GPR	Misc	Find the absolute value of the Q31 fractional element in the source register, saturating the value -1.0 to the maximum positive Q31 fractional value.
PRECR.QB.PH rd,rs,rt	Two Pair Integer Halfwords	Four Integer Bytes	GPR	Misc	Reduce the precision of four signed integer halfword input values by discarding the eight most-significant bits from each to create four signed integer byte output values. The two halfword values from register *rs* are used to create the two left-most byte results, allowing an endian-agnostic implementation.
PRECRQ.QB.PH rd,rs,rt	2 Pair Q15	Quad Byte	GPR	Misc	Reduce the precision of four Q15 fractional input values by truncation to create four Q7 fractional output values. The two Q15 values from register *rs* are written to the two left-most byte results, allowing an endian-agnostic implementation.
PRECR_SRA.PH.W rt,rs,sa PRECR_SRA_R.PH.W rt,rs,sa	Two Integer Words	Pair Integer Halfword	GPR	Misc	Reduce the precision of two integer word values to create a pair of integer halfword values. Each word value is first shifted right arithmetically by *sa* bit positions, and optionally rounded up by adding 1 at the most-significant discard bit position. The 16 least-significant bits of each word are then written to the corresponding halfword elements of destination register *rt*.
PRECRQ.PH.W rd,rs,rt PRECRQ_RS.PH.W rd,rs,rt	2 Q31	Pair halfword	GPR	Misc	Reduce the precision of two Q31 fractional input values by truncation to create two Q15 fractional output values. The Q15 value obtained from register *rs* creates the left-most result, allowing an endian-agnostic implementation. Results may be optionally rounded up and saturated before being written to the destination.
PRECRQU_S.QB.PH rd,rs,rt	2 Pair Q15	Quad Unsigned Byte	GPR	Misc	Reduce the precision of four Q15 fractional values by saturating and truncating to create four unsigned byte values.
PRECEQ.W.PHL rd,rt PRECEQ.W.PHR rd,rt	Q15	Q31	GPR	Misc	Expand the precision of a Q15 fractional value to create a Q31 fractional value by adding 16 least-significant bits to the input value.
PRECEQU.PH.QBL rd,rt PRECEQU.PH.QBR rd,rt PRECEQU.PH.QBLA rd,rt PRECEQU.PH.QBRA rd,rt	Unsigned Byte	Q15	GPR	Video	Expand the precision of two unsigned byte values by prepending a sign bit and adding seven least-significant bits to each to create two Q15 fractional values.

Table 13.4 List of Instructions in MIPS® DSP-R2 ASE in Arithmetic Sub-class *(continued)*

Instruction Mnemonics	Input Data Type	Output Data Type	Writes GPR / ac / DSPControl	App	Description
PRECEU.PH.QBL rd,rt PRECEU.PH.QBR rd,rt PRECEU.PH.QBLA rd,rt PRECEU.PH.QBRA rd,rt	Unsigned Byte	Unsigned half-word	GPR	Video	Expand the precision of two unsigned byte values by adding eight least-significant bits to each to create two unsigned halfword values.

Table 13.5 List of Instructions in MIPS® DSP ASE in GPR-Based Shift Sub-class

Instruction Mnemonics	Input Data Type	Output Data Type	Writes GPR / ac / DSPControl	App	Description
SHLL.QB rd, rt, sa SHLLV.QB rd, rt, rs	Quad Unsigned Byte	Quad Unsigned Byte	GPR	Misc	Element-wise left shift of eight signed bytes. Zeros are inserted into the bits emptied by the shift. The shift amount is specified by the three least-significant bits of sa or *rs*.
SHLL.PH rd, rt, sa SHLLV.PH rd, rt, rs SHLL_S.PH rd, rt, sa SHLLV_S.PH rd, rt, rs	Pair Signed halfword	Pair Signed halfword	GPR	Misc	Element-wise left shift of two signed halfwords, with optional saturation on overflow. Zeros are inserted into the bits emptied by the shift. The shift amount is specified by the four least-significant bits of sa or *rs*.
SHLL_S.W rd, rt, sa SHLLV_S.W rd, rt, rs	Signed Word	Signed Word	GPR	Misc	Left shift of a signed word, with saturation on overflow. Zeros are inserted into the bits emptied by the shift. The shift amount is specified by the five least-significant bits of sa or *rs*. Use the MIPS32 instructions SLL or SLLV for non-saturating shift operations.
SHRL.QB rd, rt, sa SHRLV.QB rd, rt, rs	Quad Unsigned Byte	Quad Unsigned Byte	GPR	Video	Element-wise logical right shift of four byte values. Zeros are inserted into the bits emptied by the shift. The shift amount is specified by the three least-significant bits of sa or *rs*.
SHRL.PH rd, rt, sa SHRLV.PH rd, rt, rs	Pair Half-words	Pair Half-words	GPR	Video	Element-wise logical right shift of two halfword values. Zeros are inserted into the bits emptied by the shift. The shift amount is specified by the four least-significant bits of *rs* or the *sa* argument.
SHRA.QB rd,rt,sa SHRA_R.QB rd,rt,sa SHRAV.QB rd,rt,rs SHRAV_R.QB rd,rt,rs	Quad Byte	Quad Byte	GPR	Misc	Element-wise arithmetic (sign preserving) right shift of four byte values. Optional rounding may be performed, adding 1 at the most-significant discard bit position. The shift amount is specified by the three least-significant bits of *rs* or by the argument *sa*.

Table 13.5 List of Instructions in MIPS® DSP ASE in GPR-Based Shift Sub-class *(continued)*

Instruction Mnemonics	Input Data Type	Output Data Type	Writes GPR / ac / DSPControl	App	Description
SHRA.PH rd, rt, sa SHRAV.PH rd, rt, rs SHRA_R.PH rd, rt, sa SHRAV_R.PH rd, rt, rs	Pair Signed halfword	Pair Signed halfword	GPR	Misc	Element-wise arithmetic (sign preserving) right shift of two halfword values. Optionally, rounding may be performed, adding 1 at the most-significant discard bit position. The shift amount is specified by the four least-significant bits of *rs* or by the argument *sa*.
SHRA_R.W rd, rt, sa SHRAV_R.W rd, rt, rs	Signed Word	Signed Word	GPR	Video	Arithmetic (sign preserving) right shift of a word value. Optionally, rounding may be performed, adding 1 at the most-significant discard bit position. The shift amount is specified by the five least-significant bits of *rs* or the argument *sa*.

Table 13.6 List of Instructions in MIPS® DSP-R2 ASE in Multiply Sub-class

Instruction Mnemonics	Input Data Type	Output Data Type	Writes GPR / ac / DSPControl	App	Description
MULEU_S.PH.QBL rd,rs,rt MULEU_S.PH.QBR rd,rs,rt	Pair Unsigned Byte, Pair Unsigned Halfword,	Pair Unsigned Halfword	GPR	Still Image	Element-wise multiplication of two unsigned byte values from register *rs* with two unsigned halfword values from register *rt*. Each 24-bit product is truncated to 16 bits, with saturation if the product exceeds 0xFFFF, and written to the corresponding element in the destination register.
MULQ_RS.PH rd,rs,rt	Pair Q15	Pair Q15	GPR	Misc	Element-wise multiplication of two Q15 fractional values to create two Q15 fractional results, with rounding and saturation. After multiplication, each 32-bit product is rounded up by adding 0x00008000, then truncated to create a Q15 fractional value that is written to the destination register. If both multiplicands are -1.0, the result is saturated to the maximum positive Q15 fractional value. To stay compliant with the base architecture, this instruction leaves the base *HI-LO* pair **UNPREDICTABLE** after the operation. The other DSP-R2 ASE accumulators *ac1-ac3* are untouched.

Table 13.6 List of Instructions in MIPS® DSP-R2 ASE in Multiply Sub-class *(continued)*

Instruction Mnemonics	Input Data Type	Output Data Type	Writes GPR / ac / DSPControl	App	Description
MULEQ_S.W.PHL rd,rs,rt MULEQ_S.W.PHR rd,rs,rt	Pair Q15	Q31	GPR	VoIP	Multiplication of two Q15 fractional values, shifting the product left by 1 bit to create a Q31 fractional result. If both multiplicands are -1.0 the result is saturated to the maximum positive Q31 value. To stay compliant with the base architecture, this instruction leaves the base *HI-LO* pair **UNPREDICTABLE** after the operation. The other DSP-R2 ASE accumulators *ac1-ac3* must beareuntouched.
DPAU.H.QBL DPAU.H.QBR	Pair Bytes	Halfword	Acc	Image	Dot-product accumulation. Two pairs of corresponding unsigned byte elements from source registers *rt* and *rs* are separately multiplied, and the two 16-bit products are then summed together. The summed products are then added to the accumulator.
DPSU.H.QBL DPSU.H.QBR	Pair Bytes	Halfword	Acc	Image	Dot-product subtraction. Two pairs of corresponding unsigned byte elements from source registers *rt* and *rs* are separately multiplied, and the two 16-bit products are then summed together. The summed products are then subtracted from the accumulator.
DPA.W.PH ac,rs,rt	Pair Signed Halfword	Pair Signed Halfword	ac	VoIP / SoftM	Dot-product accumulation. The two pairs of corresponding signed integer halfword values from source registers *rt* and *rs* are separately multiplied to create two separate integer word products. The products are then summed and accumulated into the specified accumulator.
DPAX.W.PH ac,rs,rt	Pair Signed Halfword	Doubleword	ac	VoIP	Dot-product with crossed operands and accumulation. The two crossed pairs of signed integer halfword values from source registers *rt* and *rs* are separately multiplied to create two separate integer word products. The products are then summed and accumulated into the specified accumulator.

Table 13.6 List of Instructions in MIPS® DSP-R2 ASE in Multiply Sub-class *(continued)*

Instruction Mnemonics	Input Data Type	Output Data Type	Writes GPR / ac / DSPControl	App	Description
DPAQ_S.W.PH ac,rs,rt	Pair Q15	Q32.31	ac	VoIP / SoftM	Dot-product accumulation. Two pairs of corresponding Q15 fractional values from source registers *rt* and *rs* are separately multiplied and left-shifted 1 bit to create two Q31 fractional products. For each product, if both multiplicands are equal to -1.0 the product is clamped to the maximum positive Q31 fractional value. The products are then summed, and the sum is then sign extended to the width of the accumulator and accumulated into the specified accumulator. This instruction may be used to compute the imaginary component of a 16-bit complex multiplication operation after first swapping the operands to place them in the correct order.
DPAQX_S.W.PH ac,rs,rt	Pair Signed Halfword	Q32.31	ac	VoIP	Dot-product with saturating fractional multiplication and using crossed operands, with a final accumulation. The two crossed pairs of signed fractional halfword values from source registers *rt* and *rs* are separately multiplied to create two separate fractional word products. The products are then summed and accumulated into the specified accumulator.
DPAQX_SA.W.PH ac,rs,rt	Pair Signed Halfword	Q32.31	ac	VoIP	Dot-product with saturating fractional multiplication and using crossed operands, with a final saturating accumulation. The two crossed pairs of signed fractional halfword values from source registers *rt* and *rs* are separately multiplied to create two separate fractional word products. The products are then summed and accumulated with saturation into the specified accumulator.
DPS.W.PH ac,rs,rt	Pair Signed Halfword	Doubleword	ac	VoIP / SoftM	Dot-product subtraction. The two pairs of corresponding signed integer halfword values from source registers *rt* and *rs* are separately multiplied to create two separate integer word products. The products are then summed and subtracted from the specified accumulator.

Table 13.6 List of Instructions in MIPS® DSP-R2 ASE in Multiply Sub-class *(continued)*

Instruction Mnemonics	Input Data Type	Output Data Type	Writes GPR / ac / DSPControl	App	Description
DPSX.W.PH ac,rs,rt	Pair Signed Half-word	Q32.31	ac	VoIP	Dot-product with crossed operands and subtraction. The two crossed pairs of signed integer halfword values from source registers *rt* and *rs* are separately multiplied to create two separate integer word products. The products are then summed and subtracted into the specified accumulator.
DPSQ_S.W.PH ac,rs,rt	Pair Q15	Q32.31	ac	VoIP / SoftM	Dot-product subtraction. Two pairs of corresponding Q15 fractional values from source registers *rt* and *rs* are separately multiplied and left-shifted 1 bit to create two Q31 fractional products. For each product, if both multiplicands are equal to -1.0 the product is clamped to the maximum positive Q31 fractional value. The products are then summed, and the sum is then sign extended to the width of the accumulator and subtracted from the specified accumulator. This instruction may be used to compute the imaginary component of a 16-bit complex multiplication operation after first swapping the operands to place them in the correct order.
DPSQX_S.W.PH ac,rs,rt	Pair Signed Half-word	Q32.31	ac	VoIP	Dot-product with saturating fractional multiplication and using crossed operands, with a final subtraction. The two crossed pairs of signed fractional halfword values from source registers *rt* and *rs* are separately multiplied to create two separate fractional word products. The products are then summed and subtracted from the specified accumulator.
DPSQX_SA.W.PH ac,rs,rt	Pair Signed Half-word	Q32.31	ac	VoIP	Dot-product with saturating fractional multiplication and using crossed operands, with a final saturating subtraction. The two crossed pairs of signed fractional halfword values from source registers *rt* and *rs* are separately multiplied to create two separate fractional word products. The products are then summed and subtracted with saturation into the specified accumulator.

Table 13.6 List of Instructions in MIPS® DSP-R2 ASE in Multiply Sub-class *(continued)*

Instruction Mnemonics	Input Data Type	Output Data Type	Writes GPR / ac / DSPControl	App	Description
MULSAQ_S.W.PH ac,rs,rt	Pair Q15	Q32.31	ac	SoftM	Complex multiplication step. Performs element-wise fractional multiplication of the two Q15 fractional values from registers *rt* and *rs*, subtracting one product from the other to create a Q31 fractional result that is added to accumulator *ac*. The intermediate products are saturated to the maximum positive Q31 fractional value if both multiplicands are equal to -1.0.
DPAQ_SA.L.W ac,rs,rt	Q31	Q63	ac	Audio	Fractional multiplication of two Q31 fractional values to produce a Q63 fractional product. If both multiplicands are -1.0 the product is saturated to the maximum positive Q63 fractional value. The product is then added to accumulator *ac*. If the addition results in overflow or underflow, the accumulator is saturated to the maximum positive or minimum negative value.
DPSQ_SA.L.W ac,rs,rt	Q31	Q63	ac	Audio	Fractional multiplication of two Q31 fractional values to produce a Q63 fractional product. If both multiplicands are -1.0 the product is saturated to the maximum positive Q63 fractional value. The product is then subtracted from accumulator *ac*. If the addition results in overflow or underflow, the accumulator is saturated to the maximum positive or minimum negative value.
MAQ_S.W.PHL ac,rs,rt MAQ_S.W.PHR ac,rs,rt	Q15	Q32.31	ac	SoftM	Fractional multiply-accumulate. The product of two Q15 fractional values is sign extended to the width of the accumulator and added to accumulator *ac*. The intermediate product is saturated to the maximum positive Q31 fractional value if both multiplicands are equal to -1.0.
MAQ_SA.W.PHL ac,rs,rt MAQ_SA.W.PHR ac,rs,rt	Q15	Q31	ac	speech	Fractional multiply-accumulate with saturation after accumulation. The product of two Q15 fractional values is sign extended to the width of the accumulator and added to accumulator *ac*. The intermediate product is saturated to the maximum positive Q31 fractional value if both multiplicands are equal to -1.0. If the accumulation results in overflow or underflow, the accumulator value is saturated to the maximum positive or minimum negative Q31 fractional value.

Table 13.6 List of Instructions in MIPS® DSP-R2 ASE in Multiply Sub-class *(continued)*

Instruction Mnemonics	Input Data Type	Output Data Type	Writes GPR / ac / DSPControl	App	Description
MUL.PH rd,rs,rt MUL_S.PH rd,rs,rt	Pair Signed Halfword	Pair Signed Halfword	GPR	speech	Element-wise multiplication of two vectors of signed integer halfwords, writing the 16 least-significant bits of each 32-bit product to the corresponding element of the destination register. Optional saturation clamps each 16-bit result to the maximum positive or minimum negative value if the product cannot be accurately represented in 16 bits.
MULQ_S.PH rd,rs,rt	Pair Q15	Pair Q15	GPR	speech	Element-wise multiplication of two vectors of Q15 fractional halfwords, writing the 16 most-significant bits of each Q31-format product to the corresponding element of the destination register. Each result is saturated to the maximum positive Q15 value if both multiplicands were equal to -1.0 (0x8000 hexadecimal).
MULQ_S.W rd,rs,rt	Q31	Q31	GPR	speech	Fractional multiplication of two Q31 format words to create a Q63 format result that is truncated by discarding the 32 least-significant bits before being written to the destination register. The result is saturated to the maximum positive Q31 value if both multiplicands were equal to -1.0 (0x80000000 hexadecimal).
MULQ_RS.W rd,rs,rt	Q31	Q31	GPR	speech	Multiplication of two Q31 fractional words to create a Q63-format intermediate product that is rounded up by adding a 1 at bit position 31. The 32 most-significant bits of the rounded result are then written to the destination register. If both multiplicands were equal to -1.0 (0x80000000 hexadecimal), rounding is not performed and the result is clamped to the maximum positive Q31 value before being written to the destination.
MULSA.W.PH ac,rs,rt	Pair Signed Halfword	Doubleword	ac	speech	Element-wise multiplication of two vectors of signed integer halfwords to create two 32-bit word intermediate results. The right intermediate result is subtracted from the left intermediate result, and the resulting sum is accumulated into the specified accumulator.
MADD, MADDU, MSUB, MSUBU, MULT, MULTU	Word	DoubleWord	ac	Misc	Allows these instructions to target accumulators *ac1*, *ac2*, and *ac3* (in addition to the original ac0 destination).

Table 13.7 List of Instructions in MIPS® DSP-R2 ASE in Bit Manipulation Sub-class

Instruction Mnemonics	Input Data Type	Output Data Type	Writes GPR / ac / DSPControl	App	Description
BITREV rd,rt	Unsigned Word	Unsigned Word	GPR	Audio / FFT	Reverse the order of the 16 least-significant bits of register *rt*, writing the result to register *rd*. The 16 most-significant bits are set to zero.
INSV rt,rs	Unsigned Word	Unsigned Word	GPR	Misc	Like the Release 2 INS instruction, except that the 5 bits for *pos* and *size* values are obtained from the *DSPControl* register. *size* = scount[14:10], and *pos* = pos[20:16].
REPL.QB rd,imm REPLV.QB rd,rt	Byte	Quad Byte	GPR	Video / Misc	Replicate a signed byte value into the four byte elements of register *rd*. The byte value is given by the 8 least-significant bits of the specified 10-bit immediate constant or by the 8 least-significant bits of register *rt*.
REPL.PH rd,imm REPLV.PH rd,rt	Signed halfword	Pair Signed halfword	GPR	Misc	Replicate a signed halfword value into the two halfword elements of register *rd*. The halfword value is given by the 16 least-significant bits of register *rt*, or by the value of the 10-bit immediate constant, sign-extended to 16 bits.

Table 13.8 List of Instructions in MIPS® DSP-R2 ASE in Compare-Pick Sub-class

Instruction Mnemonics	Input Data Type	Output Data Type	Writes GPR / ac / DSPControl	App	Description
CMPU.EQ.QB rs,rt CMPU.LT.QB rs,rt CMPU.LE.QB rs,rt	Quad Unsigned Byte	Quad Unsigned Byte	DSPControl	Video	Element-wise unsigned comparison of the four unsigned byte elements of *rs* and *rt*, recording the boolean comparison results to the four right-most bits in the *ccond* field of the *DSPControl* register.
CMPGDU.EQ.QB rd,rs,rt CMPGDU.LT.QB rd,rs,rt CMPGDU.LE.QB rd,rs,rt	Quad Unsigned Byte	Quad Unsigned Byte	GPR DSPControl	Video	Element-wise unsigned comparison of the four right-most unsigned byte elements of *rs* and *rt*, recording the boolean comparison results to the four least-significant bits of register *rd* and to the four right-most bits in the *ccond* field of the *DSPControl* register.
CMPGU.EQ.QB rd,rs,rt CMPGU.LT.QB rd,rs,rt CMPGU.LE.QB rd,rs,rt	Quad Unsigned Byte	Quad Unsigned Byte	GPR	Video	Element-wise unsigned comparison of the four right-most unsigned byte elements of *rs* and *rt*, recording the boolean comparison results to the four least-significant bits of register *rd*.
CMP.EQ.PH rs,rt CMP.LT.PH rs,rt CMP.LE.PH rs,rt	Pair Signed halfword	Pair Signed halfword	DSPControl	Misc	Element-wise signed comparison of the two halfword elements of *rs* and *rt*, recording the boolean comparison results to the two right-most bits in the *ccond* field of the *DSPControl* register.

Table 13.8 List of Instructions in MIPS® DSP-R2 ASE in Compare-Pick Sub-class*(continued)*

Instruction Mnemonics	Input Data Type	Output Data Type	Writes GPR / ac / DSPControl	App	Description
PICK.QB rd,rs,rt	Quad Unsigned Byte	Quad Unsigned Byte	GPR	Video	Element-wise selection of unsigned bytes from the four bytes of registers *rs* and *rt* into the corresponding elements of register *rd*, based on the value of the four right-most bits of the *ccond* field in the *DSPControl* register. If the corresponding *ccond* bit is 1, the byte value is copied from register *rs*, otherwise it is copied from *rt*.
PICK.PH rd,rs,rt	Pair Signed halfword	Pair Signed halfword	GPR	Misc	Element-wise selection of signed halfwords from the two halfwords in registers *rs* and *rt* into the corresponding elements of register *rd*, based on the value of the two right-most bits of the *ccond* field in the *DSPControl* register. If the corresponding *ccond* bit is 1, the halfword value is copied from register *rs*, otherwise it is copied from *rt*.
APPEND rt,rs,sa	Two Words	Word	GPR	Misc	Shifts the 32-bit word in register *rt* left by *sa* bits, inserting the *sa* least-significant bits from register *rs* into the bit positions emptied by the shift. The 32-bit result is then written to register *rt*.
PREPEND rt,rs,sa	Two Words	Word	GPR	Misc	Shifts the 32-bit word in register *rt* right by *sa* bits, inserting the *sa* least-significant bits from register *rs* into the bit positions emptied by the shift. The 32-bit result is then written to register *rt*.
BALIGN rt,rs,bp	Two Words	Word	GPR	Misc	Packs *bp* bytes from register *rt* and (4-*bp*) bytes from register *rs* into a 32-bit word and writes it to register *rt*.
PACKRL.PH rd,rs,rt	Pair Signed Halfwords	Pair Signed Halfword	GPR	Misc	Pack two halfwords taken from registers *rs* and *rt* into destination register *rd*.

Table 13.9 List of Instructions in Accumulator and DSPControl Access Sub-class

Instruction Mnemonics	Input Data Type	Output Data Type	Writes GPR / ac / DSPControl	App	Description
EXTR.W rt,ac,shift EXTR_R.W rt,ac,shift EXTR_RS.W rt,ac,shift	Q63	Q31	GPR	Misc	Extract a Q31 fractional value from the 32 least-significant bits of 64-bit accumulator *ac*. The accumulator value may be shifted right logically by *shift* bits prior to the extraction, and the extracted value may be optionally rounded or rounded and saturated before being written to register *rt*. The *shift* argument value ranges from 0 to 31. The optional rounding step adds 1 at the most-significant bit position discarded by the shift. The optional saturation clamps the extracted value to the maximum positive Q31 value if the rounding step results in overflow.
EXTR_S.H rt,ac,shift	Q63	Q15	GPR	Misc	Extract a Q15 fractional value from the 16 least-significant bits of 64-bit accumulator *ac*. The accumulator value may be shifted right logically by *shift* bits prior to the extraction, and the extracted value is saturated before being written to register *rt*. The *shift* argument value ranges from 0 to 31. The saturation clamps the extracted value to the maximum positive or minimum negative Q15 value if the shifted accumulator value cannot be represented accurately as a Q15 format value.
EXTRV_S.H rt,ac,rs	Q63	Q15	GPR	Misc	Extract a Q15 fractional value from the 16 least-significant bits of 64-bit accumulator *ac*. The accumulator value may be shifted right logically by *shift* bits prior to the extraction, and the extracted value is saturated before being written to register *rt*. The *shift* argument ranges from 0 to 31 and is given by the five least-significant bits of register *rs*. The saturation clamps the extracted value to the maximum positive or minimum negative Q15 value if the shifted accumulator value cannot be represented accurately as a Q15 format value.

Table 13.9 List of Instructions in Accumulator and DSPControl Access Sub-class*(continued)*

Instruction Mnemonics	Input Data Type	Output Data Type	Writes GPR / ac / DSPControl	App	Description
EXTRV.W rt,ac,rs EXTRV_R.W rt,ac,rs EXTRV_RS.W rt,ac,rs	Q63	Q31	GPR	Misc	Extract a Q31 fractional value from the 32 least-significant bits of 64-bit accumulator *ac*. The accumulator value may be shifted right logically by *shift* bits prior to the extraction, and the extracted value may be optionally rounded or rounded and saturated before being written to register *rt*. The *shift* argument value is provided by the five least-significant bits of *rs* and ranges from 0 to 31. The optional rounding step adds 1 at the most-significant bit position discarded by the shift. The optional saturation clamps the extracted value to the maximum positive Q31 value if the rounding step results in overflow.
EXTP rt,ac,size EXTPV rt,ac,rs EXTPDP rt,ac,size EXTPDPV rt,ac,rs	Unsigned DWord	Unsigned Word	GPR / *DSPControl*	Audio / Video	Extract a set of *size*+1 contiguous bits from accumulator *ac*, right-justifying and sign-extending the result to32 bits before writing the result to register *rt*. The position of the left-most bit to extract is given by the value of the *pos* field in the *DSPControl* register. The number of bits (less one) to extract is provided either by the *size* immediate operand or by the five least-significant bits of *rs*. The EXTPDP and EXTPDPV instructions also decrement the *pos* field by *size*+1 to facilitate sequential bit field extraction operations.
SHILO ac,shift SHILOV ac,rs	Unsigned DWord	Unsigned DWord	ac	Misc	Shift accumulator *ac* left or right by the specified number of bits, writing the shifted value back to the accumulator. The signed shift argument is specified either by the immediate operand *shift* or by the six least-significant bits of register *rs*. A negative shift argument results in a right shift of up to 32 bits, and a positive shift argument results in a left shift of up to 31 bits.
MTHLIP rs, ac	Unsigned Word	Unsigned Word	ac / *DSPControl*	Audio / Video	Copy the *LO* register of the specified accumulator to the *HI* register, copy *rs* to *LO*, and increment the *pos* field in *DSPcontrol* by 32.
MFHI/MFLO/MTHI/MTLO	Unsigned Word	Unsigned Word	GPR/ac	Misc	Copy an unsigned word to or from the specified accumulator *HI* or *LO* register to the specified GPR.
WRDSP rt,mask	Unsigned Word	Unsigned Word	*DSPControl*	Misc	Overwrite specific fields in the *DSPControl* register using the corresponding bits from the specified GPR. Bits in the *mask* argument correspond to specific fields in *DSPControl*; a value of 1 causes the corresponding *DSPControl* field to be overwritten using the corresponding bits in *rt*, otherwise the field is unchanged.

Table 13.9 List of Instructions in Accumulator and DSPControl Access Sub-class*(continued)*

Instruction Mnemonics	Input Data Type	Output Data Type	Writes GPR / ac / DSPControl	App	Description
RDDSP rt,mask	Unsigned Word	Unsigned Word	GPR	Misc	Copy the values of specific fields in the *DSPControl* register to the specified GPR. Bits in the *mask* argument correspond to specific fields in *DSPControl*; a value of 1 causes the corresponding *DSPControl* field to be copied to the corresponding bits in *rt*, otherwise the bits in *rt* are unchanged.

Table 13.10 List of Instructions in MIPS® DSP-R2 ASE in Indexed-Load Sub-class

Instruction Mnemonics	Input Data Type	Output Data Type	Writes GPR / ac / DSPControl	App	Description
LBUX rd,index(base)	-	Unsigned byte	GPR	Misc	Index byte load from address base+(index). Loads the byte in the low-order bits of the destination register and zero-extends the result.
LHX rd,index(base)	-	Signed halfword	GPR	Misc	Index halfword load from address base+(index). Loads the halfword in the low-order bits of the register and sign-extends the result.
LWX rd, index(base)	-	Signed Word	GPR	Misc	Indexed word load from address base+(index).

Table 13.11 List of Instructions in MIPS® DSP-R2 ASE in Branch Sub-class

Instruction Mnemonics	Input Data Type	Output Data Type	Writes GPR / ac / DSPControl	App	Description
BPOSGE32 offset	-	-	-	Audio / Video	Branch if the *pos* value is greater than or equal to integer 32.

13.12 Listing of DSP-R2 ASE Instruction Groups

Table 13.12 shows an alphabetical listing of the DSP-R2 instruction set, along with the associated instruction group, the page number location of the actual instruction. The actual instruction can be viewed by clicking on either the instruction of the page number reference in the table below. For the definition of each instruction, refer to Table 13.4 through Table 13.11 above.

Table 13.12 Alphabetical Listing of DSP-R2 Instructions

Instruction Name	Instruction Group
ABSQ_S.PH	Arithmetic

Table 13.12 Alphabetical Listing of DSP-R2 Instructions *(continued)*

Instruction Name	Instruction Group
ABSQ_S.W	Arithmetic
ADDQ_S.W	Arithmetic
ADDQH[_R].PH	Arithmetic
ADDQH[_R].W	Arithmetic
ADDSC	Arithmetic
ADDU[_S].PH	Arithmetic
ADDU[_S].QB	Arithmetic
ADDUH[_R].QB	Arithmetic
ADDWC	Arithmetic
APPEND	Compare-Pick
BALIGN	Compare-Pick
BPOSGE32	Branch
BITREV	Bit Manipulation
CMP.cond.PH	Compare-Pick
CMPGDU.cond.QB	Compare-Pick
CMPGU.cond.QB	Compare-Pick
CMPU.cond.QB	Compare-Pick
DPA.W.PH	Multiply
DPAQ_S.W.PH	Multiply
DPAQ_SA.L.W	Multiply
DPAQX_S.W.PH	Multiply
DPAQX_SA.W.PH	Multiply
DPAU.H.QBL	Multiply
DPAU.H.QBT	Multiply
DPAX.W.PH	Multiply
DPS.W.PH	Multiply
DPSQ_S.W.PH	Multiply
DPSQ_SA.L.W	Multiply
DPSQX_S.W.PH	Multiply
DPSQX_SA.W.PH	Multiply
DPSU.H.QBL	Multiply
DPSU.H.QBR	Multiply
DPSX.W.PH	Multiply
EXTPV	Accumulator/DSPControl Access
EXTPDP	Accumulator/DSPControl Access
EXTPDPV	Accumulator/DSPControl Access
EXTPV	Accumulator/DSPControl Access

Table 13.12 Alphabetical Listing of DSP-R2 Instructions *(continued)*

Instruction Name	Instruction Group
EXTR[_RS].W	Accumulator/DSPControl Access
EXTR_S.H	Accumulator/DSPControl Access
EXTRV[_RS].W	Accumulator/DSPControl Access
EXTRV_S.H	Accumulator/DSPControl Access
INSV	Bit Manipulation
LBUX	Indexed Load
LHX	Indexed Load
LWX	Indexed Load
MADD	Multiply
MADDU	Multiply
MAQ_S[A].W.PHL	Multiply
MAQ_S[A].W.PHR	Multiply
MFHI	Accumulator/DSPControl Access
MFLO	Accumulator/DSPControl Access
MODSUB	Arithmetic
MSUB	Multiply
MSUBU	Multiply
MTHI	Accumulator/DSPControl Access
MTHILIP	Accumulator/DSPControl Access
MTLO	Accumulator/DSPControl Access
MUL[_S].PH	Arithmetic
MULEQ_S.W.PHL	Arithmetic
MULEQ_S.W.PHR	Arithmetic
MULEU_S.PH.QBL	Arithmetic
MULEU_S.PH.QBR	Arithmetic
MULQ_RS.W	Arithmetic
MULQ_S.PH	Arithmetic
MULQ_RS.PH	Arithmetic
MULQ_S.W	Arithmetic
MULSA.W.PH	Arithmetic
MULSAQ_S.W.PH	Arithmetic
MULT	Arithmetic
MULTU	Arithmetic
PACKRL.PH	Compare-Pick
PICK.PH	Compare-Pick
PICK.QB	Compare-Pick
PRECEQ.W.PHL	Arithmetic

Table 13.12 Alphabetical Listing of DSP-R2 Instructions *(continued)*

Instruction Name	Instruction Group
PRECEQ.W.PHR	Arithmetic
PRECEQU.PH.QBL	Arithmetic
PRECEQU.PH.QBLA	Arithmetic
PRECEQU.PH.QBR	Arithmetic
PRECEQU.PH.QBRA	Arithmetic
PRECEU.PH.QBL	Arithmetic
PRECEU.PH.QBLA	Arithmetic
PRECEU.PH.QBR	Arithmetic
PRECEU.PH.QBRA	Arithmetic
PRECR.QB.PH	Arithmetic
PRECRQ.PH.W	Arithmetic
PRECRQ_RS.PH.W	Arithmetic
PRECRQU_S.QB.PH	Arithmetic
PRECR_SRA[_R].PH.W	Arithmetic
PREPEND	Compare-Pick
RADDU.W.QB	Arithmetic
RDDSP	Accumulator/DSPControl Access
REPL.PH	Bit Manipulation
REPL.QB	Bit Manipulation
REPLV.PH	Bit Manipulation
REPL.QB	Bit Manipulation
SHILO	Accumulator/DSPControl Access
SHILOV	Accumulator/DSPControl Access
SHLL[_S].PH	GPR-Based Shift
SHLL.QB	GPR-Based Shift
SHLLV.QB	GPR-Based Shift
SHLL_S.W	GPR-Based Shift
SHLLV[_S].PH	GPR-Based Shift
SHLLV_S.W	GPR-Based Shift
SHRA[_R].PH	GPR-Based Shift
SHRAV[_R].PH	GPR-Based Shift
SHRA[_R].QB	GPR-Based Shift
SHRAV[_R].QB	GPR-Based Shift
SHRAV[_R].W	GPR-Based Shift
SHRL.PH	GPR-Based Shift
SHRLV.PH	GPR-Based Shift
SHRL.QB	GPR-Based Shift

Table 13.12 Alphabetical Listing of DSP-R2 Instructions *(continued)*

Instruction Name	Instruction Group
SHRLV.QB	GPR-Based Shift
SUBQ[_S].PH	Arithmetic
SUBQ_S.W	Arithmetic
SUBQH[_R].PH	Arithmetic
SUBQH[_R].W	Arithmetic
SUBU[_S].PH	Arithmetic
SUBU[_S].QB	Arithmetic
SUBUH[_R].QB	Arithmetic
WRDSP	Accumulator/DSPControl Access

13.13 DSP Instruction Latencies and Repeat Rates

Latency is defined with respect to instruction pair, but for ease of documenting they are defined for the instruction. If the behavior per instruction differs from that of an instruction pair, this difference is mentioned in the Notes column. If the instruction does not loads any general purpose register (GPR) then it is shown as not applicable (n/a).

Repeat rate is measured as number of independent instructions that can be sent in 1 cycle.

Table 13.13 proAptiv DSP Instruction Latencies and Repeat Rates

Instruction	Latency	Repeat Rate	Notes
ADDQ{_S}.PH, ADDQ_S.W,	2	1	
ADDU{_S}.PH, ADDU{_S}.QB	2	1	
ADDUH{_R}.QB,	2	1	
ADDQH{_R}.PH, ADDQH{_R}.W	2	1	
ADDSC, ADDWC	2	1	
SUBQ{_S}.PH, SUB_S.W	2	1	
SUBU{_S}.PH, SUBU{_S}.QB	2	1	
SUBUH{_R}.QB	2	1	
SUBQH{_R}.PH, SUBQH{_R}.W	2	1	
MODSUB, RADDU.W.QB	2	1	
ABSQ_S.QB, ABSQ_S.PH, ABSQ_S.W	2	1	
PRECR.QB.PH	2	1	
PRECRQ.QB.PH	2	1	
PRECR_SRA{_R}.PH.W	2	1	
PRECRQ{_RS}.PH.W	2	1	
PRECRQU_S.QB.PH	2	1	
PRECEQ.W.PHL, PRECEQ.W.PHR	2	1	
PRECEQU.PH.QBL{A}, PRECEQU.PH.QBR{A}	2	1	
PRECEU.PH.QBL{A}, PRECEU.PH.QBR{A}	2	1	
SHLL.QB, SHLLV.QB	2	1	
SHLL{_S}.PH, SHLLV{_S}.PH	2	1	
SHLL_S.W, SHLLV_S.W	2	1	
SHRL.QB, SHRLV.QB	2	1	
SHRL.PH, SHRLV.PH	2	1	
SHRA{_R}.QB, SHRAV{_R}.QB	2	1	
SHRA{_R}.PH, SHRAV{_R}.PH	2	1	
SHRA_R.W, SHRAV_R.W	2	1	
MULEU_S.PH.QBL, MULEU_S.PH.QBR	6	1	
MULQ_RS.PH	6	1	
MULEQ_S.W.PHL, MULEQ_S.W.PHR	6	1	

Table 13.13 proAptiv DSP Instruction Latencies and Repeat Rates *(continued)*

Instruction	Latency	Repeat Rate	Notes
DPAU.H.QBL, DPAU.H.QBR	6/1	1	DPA to MADD/DPA is 1 while MFHI/MFLO is 6.
DPSU.H.QBL, DPSU.H.QBR	6/1	1	
DPA.W.PH, DPAX.W.PH	6/1	1	
DPAQ_S.W.PH, DPAQX_S.W.PH, DPAQX_SA.W.PH	6/1	1	
DPS.W.PH, DPSX.W.PH	6/1	1	
DPSQ_S.W.PH, DPSQX_S.W.PH, DPSQX_SA.W.PH	6/1	1	
DPAQ_SA.L.W, DPSQ_SA.L.W	6/1	1	
MAQ_S{A}.W.PHL, MAQ_S{A}.W.PHR	6/1	1	
MADD, MADDU, MSUB, MSUBU, MULT, MULTU	6/1	1	
MULSAQ_S.W.PH	6	1	
MUL{_S}.PH	6	1	
MULQ_S.PH, MULQ_S.W, MULQ_RS.W	6	1	
MULSA.W.PH	6	1	
BITREV	2	1	
INSV	2	1	
REPL{V}.QB, REPL{V}.PH	2	1	
CMPU.EQ.QB, CMPU.LT.QB, CMPU.LE.QB	2	1	
CMPGDU.EQ.QB, CMPGDU.LT.QB, CMPGDU.LE.QB	2	1	
CMPGU.EQ.QB, CMPGU.LT.QB, CMPGU.LE.QB	2	1	
CMP.EQ.PH, CMP.LT.PH, CMP.LE.PH	2	1	
PICK.QB, PICK.PH	2	1	
APPEND, PREPEND	2	1	
BALIGN	2	1	
PACKRL.PH	2	1	
EXTR.W, EXTR_R.W, EXTR_RS.W	6	1	
EXTR_S.H, EXTRV_S.H	6	1	
EXTRV{_R,_RS}.W	6	1	
EXTP, EXTPV, EXTPDP, EXTPDPV	2	1	
SHILO, SHILOV	5	5	
MTHLIP	5/13	5	5 cycles for acc register, while 13 cycles for DSPCTL.POS register.
MFHI	2	1	
MFLO	2	1	
MTHI	5	5	
MTLO	5	5	
WRDSP	n/a	1	
RDDSP	2	1	
LBUX, LHX, LWX	4	1	Assuming L1 data cache hit.
BPOSGE32	n/a	1	

Chapter 14

EJTAG Debug Support

The EJTAG block of the proAptiv Multiprocessing System provides a system debug facility for the device. The EJTAG functions are not normally controlled by the end user, but rather are controlled by a debugger. This chapter is meant to be read in conjunction with the MIPS EJTAG Specification that was included as part of the release.

An EJTAG debug block is present in all cores available from MIPS Technologies, Inc. It contains support for things like hardware and software breakpoints, hardware single-step, and a JTAG based debug TAP for debug probe connection.

This chapter is used for debug of the proAptiv CPU core. For more information on the debugging of the Multiprocessing System, including the CM2 and CPC, refer to the next chapter entitled “Multi-CPU Debug”.

This chapter contains the following sections:

- Section 14.1 “Overview”
- Section 14.2 “Trace Funnel and Trace Types”
- Section 14.3 “Detecting Debug Mode”
- Section 14.4 “Ways of Entering Debug Mode”
- Section 14.5 “Exiting Debug Mode”
- Section 14.6 “EJTAG and PDTrace Revisions”
- Section 14.7 “Connection Options”
- Section 14.8 “Hardware Breakpoints”
- Section 14.10 “Test Access Port (TAP)”
- Section 14.11 “PDTrace”
- Section 14.12 “PDtrace Cycle-by-Cycle Behavior”
- Section 14.13 “PC Sampling”
- Section 14.14 “EJTAG Registers”
- Section 14.15 “Fast Debug Channel”
- Section 14.16 “TCB Trigger Logic”

14.1 Overview

The EJTAG debug logic in the proAptiv Multiprocessing System core is compliant with EJTAG Specification 5.0 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 PDtrace capability for program counter/data address/data value trace to On-chip memory or to Trace probe

EJTAG debug resources are often controlled via high level debugger commands. The following is a brief overview of some EJTAG features.

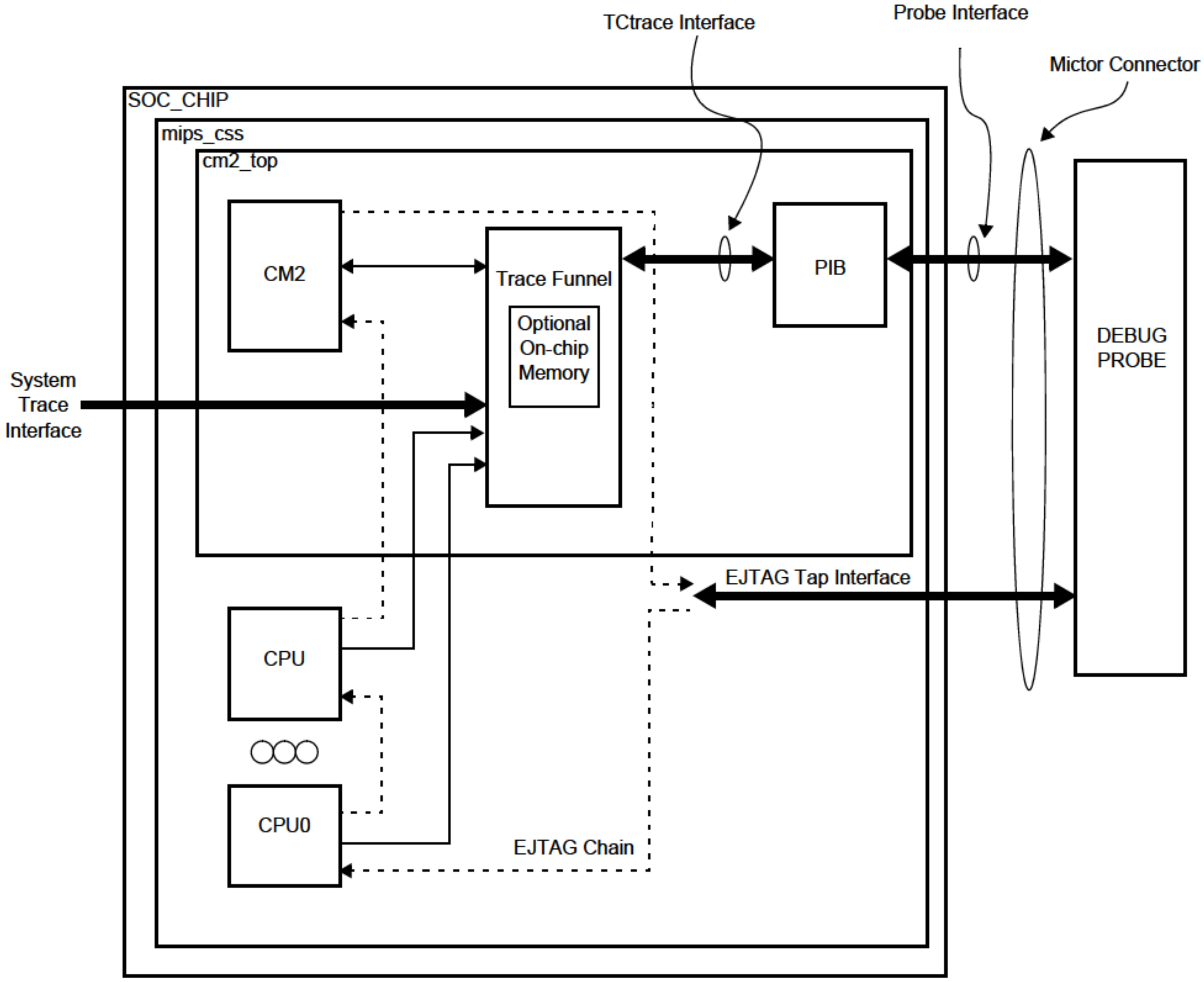
- PCSAMPLE: A feature allowing for non-intrusive reading of recently completed instruction addresses. The PCSAMPLE TAP instruction selects the TAP data register “PCSAMPLE” which contains an execution address and a flag indicating whether or not a new instruction has completed since the last read of the PCSAMPLE TAP data register.
- EJTAG TAP: The optional JTAG TAP associated with an EJTAG debug block used for communications with an EJTAG probe and debugger.
- ECR (EJTAG Control Register): This register is used mostly by probe developers and can only be accessed via a probe.
- DCR (Debug Control Register): This register is located in the drseg memory segment and can only be accessed in Debug mode.
- DINT (Debug Interrupt): an interrupt which causes a debug exception and entry into debug mode.
- DRSEG (Debug Register Segment): A memory overlay, present only while executing in debug mode, that allows access to registers controlling various EJTAG debug features.
- DMSEG (Debug Memory Segment): A memory overlay, present only while in debug mode and ECR.ProbEn is set, that an EJTAG probe emulates by satisfying processor accesses (fetches, loads, and stores.) The emulation is carried out via TAP data registers CONTROL, ADDRESS, and DATA.
- Single-Step: A debug setting that results in a debug exception after execution of a single12 non-debug mode instruction has completed.
- Hardware Breakpoint: A hardware resource capable of detecting execution or data access at virtual addresses.
- Software Breakpoint: The instruction “sdbbp” which causes a debug exception on execution. Debuggers will temporarily replace an instruction of your program with this instruction on setting a breakpoint in writeable memory.

14.2 Trace Funnel and Trace Types

The proAptiv Multiprocessing System implements a trace funnel that is used to communicate with the debug probe via the probe interface block. The trace funnel can accept trace information from either the CM2, the core, or the MIPS system trace.

The trace funnel and its connections are shown in Figure 14.1. Refer to Section 14.2.1 "Trace Types" for more information on the types of traces shown.

Figure 14.1 Trace Connections in the MIPS Debug Architecture



14.2.1 Trace Types

The proAptiv Multiprocessing System support three types of trace:

1. Core Trace
2. CM2 Trace
3. System Trace

Core Trace — Core trace allows CPU signals to be traced and routed to the trace funnel for processing. The functionality of core trace and the registers used to control it are described throughout this chapter.

CM2 Trace — The CM2 has its own trace and also manages the trace funnel. The functionality of CM2 trace and the registers used to control it are described in the CM2 chapter. Refer to 8 of this manual for more information.

MIPS System Trace — The MIPS System trace is a new feature to the proAptiv Multiprocessing System and allows the SoC designer to place signals from their non-probe SoC logic directly into the trace funnel for PDTrace to capture. The logic and registers that controls System Trace are handled by the CM2. Refer to Chapter 8 of the proAptiv Multiprocessing System Hardware User’s Manual for more information on MIPS System Trace. For additional information, refer to Section 7.6.2 of the proAptiv Multiprocessing System Hardware User’s Manual.

14.2.2 EJTAG TAP Interface

Every TAP register access (also referred to as a “scan”) is a read-before-write operation. A TAP register access captures (reads) a register value from the target and then that value is serially shifted out to the tool as a new value is simultaneously shifted in. After all of the bits of the register have been shifted the input value is updated (written.)

There are two main paths through an EJTAG TAP state machine. One provides access to the single, 5-bit instruction register and the other provides access to the currently selected data register(s). Every TAP instruction access should result in the 5 bit binary value “00001” being read. Most EJTAG TAP instructions’ sole purpose is to select which data register is accessed during a data scan. EJTAG TAP instructions not intended to select specific TAP data registers will select the BYPASS data register.

In a multi-device target system, the term “scan chain” is used to describe the serial (daisy-chained) set of TAPS which are read/written in a single scan.

14.2.3 EJTAGBOOT vs NORMALBOOT

The EJTAGBOOT TAP instruction modifies the reset value of the *ECR.ProbTrap*, *ECR.ProbEn*, and *ECR.EjtagBrk,* thereby changing device reset behavior. Subsequent warm resets result in a debug exception after release from reset. Any EJTAG TAP reset will clear the EJTAGBOOT indication as will sending a NORMALBOOT TAP instruction.

14.3 Detecting Debug Mode

The DM bit of the CP0 Debug register (CP0 Register 23, Select 0) indicates if the processor is operating in debug mode. If this bit is set, the processor is operating in debug mode. This bit is set on any debug exception and is cleared by executing a *DERET* instruction. Refer to Chapter 2, CP0 Registers, for more information on the *Debug* register.

This bit is available to both probe and non-probe related configurations and can be read at any time. The user does not need to be in Debug mode in order to read this bit. This bit, along with the associated fields in this register, can be used by software to determine the conditions under which Debug mode was entered.

14.4 Ways of Entering Debug Mode

There are five ways to enter Debug mode. Each of these ways can be entered from either software, or from a debug probe. All of these ways cause the *DM* bit in the *CP0 Debug* register to be set.

1. EJTAG Debug Single Step
2. EJTAG Debug Interrupt. Caused by the assertion of the external EJ_DINT input, or by setting the EJTAGBrk bit in the ECR register.
3. EJTAG debug hardware data breakpoint match
4. EJTAG debug hardware instruction breakpoint match
5. EJTAG Breakpoint (execution of SDBBP instruction)

14.4.1 EJTAG Debug Single Step

To enter Debug single step mode, the core must implement the single step mode. This can be determined by reading the *NoSST* bit (9) of the *CP0 Debug* register. If this bit is zero, the debug single step feature is implemented in the core. In the proAptiv Multiprocessing System, this bit is always zero to indicate that the single step feature is implemented by the core.

Single step mode can be enabled or disabled by writing to the *SST* bit (8) of the *CP0 Debug* register. If the *SST* bit is set, the single step function is available once the core enters debug mode using any of the ways listed above. For implementation that include a probe, the common way is to generate the EJTAG DINT signal, which causes a debug interrupt to the core. For non-probe implementations, software can set the EJTAGBRK bit. Both of these methods are described in the following subsection.

14.4.2 EJTAG Debug Interrupt

The EJTAG DINT signal is an implementation dependent feature that is determined at build time. The *DINTsup* bit (24) in the *Implementation* register indicates whether the DINT signal is supported. This bit is written by the *EJ_DINTsup* signal at reset depending on whether this option is selected at build time. This is a common way for probe or logic analyzer implementations to enter debug mode. Refer to Section 14.14.4.5 “Implementation Register” for more information.

Software can enter debug mode by setting the *EJTAGbrk* bit (12) or the *EJTAG Control* register. 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. Refer to Section 14.14.4.6 “EJTAG Control Register” for more information.

14.4.3 EJTAG Hardware Data Breakpoint Match

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. Refer to Section 14.8 “Hardware Breakpoints” for more information and a list of registers used to set up a data breakpoint.

14.4.4 EJTAG Hardware Instruction Breakpoint Match

Instruction breaks occur on instruction fetch operations and the break is set on the virtual address used by the instruction fetch unit. Instruction breaks can also be made on the ASID value used by the MMU. Finally, a mask can be applied to the virtual address to set breakpoints on a range of instructions. Instruction breakpoints compare the virtual address of the executed instructions (PC) and the ASID with the registers for each instruction breakpoint including masking of address and ASID. When an instruction breakpoint matches, a trigger is generated and a debug exception is optionally signalled. An internal bit in the instruction breakpoint registers is set to indicate that the match occurred.

Refer to Section 14.8 "Hardware Breakpoints" for more information and a list of register used to set up an instruction breakpoint.

14.4.5 EJTAG Software Breakpoint

Software can execute a software debug breakpoint using the SDBBP instruction. When this instruction is executed, the debugger temporarily replaces the program instruction with the SDBBP instruction when setting a breakpoint in memory.

14.5 Exiting Debug Mode

As described above, there are five basic ways to enter debug mode. On in debug mode, the mode can only be exited in one of three ways:

- Execution of a Debug Exception Return (DERET) instruction.
- Reset the core
- Power cycle the core

During normal operation, execptions are taken by the core and processed. Once the exception processing is complete, software executes an Exception Return (ERET) instruction. When in debug mode, software executes a Debug Exception Return (DERET) instruction. This causes the core to exit debug mode and return to previous mode as determined by the programmer (normal, kernel, supervisor, etc.).

Note that for a DERET instruction to be executed, the core must be in a state where it is fetching instructions. If for any reason the instruction stream has been halted and cannot resume, then the DERET instruction cannot be executed. In this case, the only other options are resetting the core, or cycling the power to the proAptiv Multiprocessing System.

14.6 EJTAG and PDTrace Revisions

This chapter is intended to be used in conjunction with the EJTAG specification (MIPS document number MD00047) and the MIPS PDTrace specification (MIPS document number MD00439). These documents contain information for multiple types of MIPS cores, so the EJTAG and PDTrace versions of the core in question must be known in order to use these documents.

- ***EJTAG version with probe***: When using the MIPS Debug facility with a debug probe, the EJTAG version used in the proAptiv core can be determined by reading the EJTAGver field in bits 31:29 of the *Implementation* register. This is a TAP controller register that is only accessible through an EJTAG probe. The proAptiv core implements EJTAG revision 5.0. Refer to Section 14.14.4.5 "Implementation Register" for more information. Note that the probe can read either the *Implementation* register of the CP0 *Debug* register described below to determine the EJTAG revision number.
- ***EJTAG version without probe***: When using the MIPS Debug facility without a debug probe, the EJTAG version used in the proAptiv core can be determined by reading the *EJTAGver* field in bits 17:15 of the CP0 *Debug* register located at CP0 register 23, select 0. The proAptiv core implements EJTAG revision 5.0. Refer to Chapter 2 of this manual for more information on the CP0 *Debug* register. Note that the kernel can only read the CP0 *Debug* register to determine the EJTAG version and does not have access to the EJTAG *Implementation* register described above.
- ***PDTrace version with probe***: When using the MIPS Debug facility with a debug probe, the PDTrace version used in the proAptiv core can be determined by reading the REV field in bits 3:0 of the *Trace Buffer Configuration* (TCBCONFIG) register located in the EJTAG TAP controller. Refer to the Section 14.14.11.2 "TCBCONFIG Register (Reg 0)" for more information on this register. The current revision is 3.0 as noted by the default value. Note that this register can only be read when an EJTAG probe is connected to the device.
- ***PDTrace version without probe***: When using the MIPS Debug facility without a debug probe, the PDTrace version used in the proAptiv core can be determined by reading the REV field in bits 3:0 of the *Trace Buffer Configuration* (TCBCONFIG) register located in the EJTAG TAP controller.

  However, since a probe is not attached in this case, the core must be in Debug mode in order to read this register. Debug mode can be entered in any of the ways described in Section 14.4 "Ways of Entering Debug Mode". Refer to the Section 14.14.11.2 "TCBCONFIG Register (Reg 0)" for more information on this register.

It should be noted that the *Device Identification* register located in Section 14.14.4.4 on page 713 contains version and part number information. This register is only accessible when an EJTAG probe is attached, but does not provide EJTAG or PDTrace revision information. This register is used to by the manufacturer for their own device identification purposes and should not be used in an attempt to determine the EJTAG or PDTrace revisions.

14.7 Connection Options

The EJTAG debug port of the proAptiv core can be accessed either via a TAP (five JTAG pins), or the EJTAG debug block through the CP0 Debug register, the DCR, and drseg space. If the TAP is used, no ROM monitor is required and there is no interference with the customers code. If there is no TAP, then the user must write their own ROM monitor.

There are two ways to connect to access the EJTAG debug facility:

- Software via the General Control Registers (GCR)
- Debug probe via the EJTAG Test Access Port (TAP)

The DCR (Debug Control Register) can be used to access the EJTAG debug port via software. This register is located in the drseg memory segment and can only be accessed in Debug mode. This register can be accessed by anyone that enters Debug mode and does not require that a probe be attached.

Access via software would mostly be performed during normal operation. As described in Section 14.4 "Ways of Entering Debug Mode" above, the CP0 Debug register (CP0 Register 23, Select 0) indicates whether or not the device is in Debug mode and the cause as to how it got there. Bit 30 of this register indicates if the core has entered Debug mode. If the core is not in Debug mode, the other bits have no meaning. If the core is in Debug mode, the other bits are used to provide additional information about how the device got into Debug mode. For example, setting a software breakpoint allows thc core to enter Debug mode.

The ECR (EJTAG Control Register) is used mostly by probe developers and can only be accessed via a probe. Refer to Section 14.14.4.6 "EJTAG Control Register" for more information.

14.8 Hardware Breakpoints

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

There are two types of simple hardware breakpoints implemented in the proAptiv Multiprocessing System core; Instruction breakpoints and Data breakpoints.

A core may be configured with the following breakpoint options:

- Zero, two, or four instruction breakpoints
- Zero, one, or two data breakpoints

14.8.1 Instruction Breakpoints

Instruction breaks occur on instruction fetch operations and the break is set on the virtual address used by the instruction fetch unit. 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 trigger is generated and a debug exception is optionally signalled. An internal bit in the instruction breakpoint registers is set to indicate that the match occurred.

14.8.2 Data Breakpoints

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 trigger is generated and a debug exception is optionally signalled. An internal bit in the data breakpoint registers is set to indicate that the match occurred.

14.8.3 Instruction Breakpoint Registers Overview

The register with implementation indication and status for instruction breakpoints in general is shown in Table 14.1.

Table 14.1 Overview of Status Register for Instruction Breakpoints

Register Mnemonic	Register Name and Description
IBS	Instruction Breakpoint Status

Up to four instruction breakpoints are available and 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 14.2.

Table 14.2 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

14.8.4 Data Breakpoint Registers Overview

The register with implementation indication and status for data breakpoints in general is shown in Table 14.3.

Table 14.3 Overview of Status Register for Data Breakpoints

Register Mnemonic	Register Name and Description
DBS	Data Breakpoint Status

Up to two data breakpoints are available and 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 14.4.

Table 14.4 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*

14.8.5 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, as described in this section. Breakpoints only match for instructions executed in non-debug mode, never on instructions executed in debug mode.

The match of an enabled breakpoint always generates a trigger indication and can also generate a debug exception. 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.

14.8.5.1 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 the bit level, The match can also include an optional compare of the ASID value. The registers for each instruction breakpoint contain 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.

14.8.5.2 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 coprocessor loads/stores 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 contain 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 $DBCn_{\mathrm{BAI}}$ 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 $DBCn_{\mathrm{BLM}}$, $DBCn_{\mathrm{BAI}}$ 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.

14.8.6 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.

14.8.6.1 Debug Exception by Instruction Breakpoint

If the breakpoint is enabled by the *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 the *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 cannot 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.

14.8.6.2 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 14.5 apply with respect to updating the *BS*[n] bits.

Table 14.5 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 a Imprecise Debug Exception

An Debug Data Break Load/Store Imprecise exception occurs when a data breakpoint indicates an imprecise match. Imprecise matches are generated when data value compare is used. In this case, the *DEPC* register and *DBD* bit in the Debug register point to an instruction later in the execution flow rather than at the load/store instruction that caused the `DB_match` equation to be true.

The load/store instruction causing the Debug Data Break Load/Store Imprecise exception always updates the destination register and completes the access to the external memory system. Therefore this load/store instruction is not re-executed on return from the debug handler, because the *DEPC* register and *DBD* bit do not point to that instruction.

Several imprecise data breakpoints can be pending at a given time, if the bus system supports multiple outstanding data accesses. The breakpoints are evaluated as the accesses finalize, and a Debug Data Break Load/Store Imprecise exception is generated only for the first one that matches. Both the first and succeeding matches cause corresponding *BS* bits and *DDBLImpr*/*DDBSImpr* to be set, but no debug exception is generated for succeeding matches, because the processor is already in Debug Mode. Similarly, if a debug exception had already occurred at the time of the first match (for example, due to a precise debug exception), then all matches cause the corresponding *BS* bits and *DDBLImpr*/*DDBSImpr* to be set, but no debug exception is generated because the processor is already in Debug Mode.

The SYNC instruction, followed by appropriate spacing must be executed before the *BS* bits and *DDBLImpr*/*DDBSImpr* bits are accessed for read or write. This delay ensures that these bits are fully updated.

Any *BS* bit set prior to the match and debug exception remains set, because only debug software can clear the *BS* bits.

14.8.7 Breakpoint used as Triggerpoint

When an enabled instruction or data breakpoint matches, the corresponding bit in the *IBS.BS* or *DBS.BS* field is set. These fields are externalized on the *SI_Ibs* and *SI_Dbs* core outputs, respectively. These outputs are intended to be used to trigger external devices such as logic analyzers. Furthermore, breakpoint matches can also be used to start or stop PDtrace. See Section 14.11.8 “Enabling PDtrace” for details.

If the breakpoints are to be used only as trigger events, the signalling of the debug exception can be suppressed by clearing the *IBCn/DBCn.BE* field and setting the *IBCn/DBCn.TE* field.

14.9 Debug Vector Addressing

The debug vector address size is managed by the *Debug Vector Address* register as described in Section 14.14.1.2 “DebugVectorAddr Register”. The *Debug Vector Address* register is a read/write register containing the base address of the debug exception vectors in bits 31:7, and a WG bit that determines whether the bits 31:30 of this field are a fixed value, or are programmable.

Bits 31:12 of the *DebugVectorAddress* register are concatenated with zeros to form the base of the debug exception vector. The exception vector base address comes from the fixed defaults for any EJTAG Debug exception. The reset state of bits 31:12 of the *DebugVectorAddress* register initialize the exception base register to `0xFC00.0480`.

The size of the *DebugVectorAddr* field depends on the state of the WG bit. At reset, the WG bit is cleared by default. In this case, the *DebugVectorAddr* field is comprised of bits 29:7. Bits 31:30 of the *DebugVectorAddr* Register are not writeable and are forced to a value of 2'b10 by hardware so that the debug exception handler will be executed from the *kseg0/kseg1* segments.

When the WG bit is set, bits 31:30 of the *DebugVectorAddr* field become writeable and are used to relocate the *DebugVectorAddr* field to other segments after they have been setup using the *SegCtl0* through *SegCtl2* registers. Note that if the WG bit is set by software (allowing bits 31:30 to become part of the *DebugVectorAddr* field) and then cleared, bits 31:30 can no longer be written by software and the state of these bits remains unchanged for any writes after WG was cleared. Therefore, it is the responsibility of software to write a value of 2'b10 to bits 31:30 of the *DebugVectorAddr* register prior to clearing the WG bit if it wants to ensure that future debug exceptions will be executed from the kseg0 or kseg1 segments.

Note that the WG bit is different from the CV bit in the SegCtl0 register located in Section 2.3.3.1, "Segmentation Control 0 — SegCtl0 (CP0 Register 5, Select 2)". Although their functions are similar, the CV bit applies only to cache error exceptions, whereas the WG bit applies to all exceptions.

If the value of the exception base register is to be changed, this must be done with *StatusBEV* equal to 1. The operation of the processor is **UNDEFINED** if the exception base field is written with a different value when *StatusBEV* is 0.

Table 14.11 shows the different debug exception vector locations that are possible.

Table 14.6 Debug Exception Vectors

ECR$_{ProbTrap}$	DCR$_{RdVec}$	Config5$_{K}$	SI_UseExceptionBase	Cache Error?	Debug Exception Vector
1	x	x	x	x	0xFF20_0200
0	1	0	x	0	2’b10 || DebugVectorAddr[29:0]
0	1	1	x	0	DebugVectorAddr[31:0]
0	1	0	x	1	3’b101 || DebugVectorAddr[28:0]
0	1	1	x	1	DebugVectorAddr[31:0]
0	0	0	1	0	2’b10 || SI_ExceptionBase[29:12] || 0x480
0	0	1	1	0	SI_ExceptionBase[31:12] || 0x480
0	0	0	1	1	3’b101 || SI_ExceptionBase[28:12] || 0x480
0	0	1	1	1	SI_ExceptionBase[31:12] || 0x480
0	0	x	0	x	0xBFC0_0480

As shown in the table above, if the *ECR*$_{ProbeTrap}$ bit (14) is set in the EJTAG Control register, then all other bits or signals that determine the location of the debug vector address have no meaning and the location of the debug exception vector default to 0xFF20_0200. Note that the *ECR*$_{ProbeEn}$ bit (15) must be set in order for this bit to have meaning.

14.10 Test Access Port (TAP)

The TAP is used only when a probe is connected to the proAptiv Multiprocessing System.

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.

14.10.1 EJTAG Internal and External Interfaces

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

Table 14.7 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.

14.10.2 Test Access Port Operation

The TAP controller is controlled by the Test Clock (*TCK*) and Test Mode Select (*TMS*) inputs. These two inputs determine whether an the Instruction register scan or data register scan is performed. The TAP consists of a small controller, driven by the *TCK* input, which responds to the *TMS* input as shown in the state diagram in Figure 14.2. The TAP uses both clock edges of *TCK*. *TMS* and *TDI* are sampled on the rising edge of *TCK*, while *TDO* changes on the falling edge of *TCK*.

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

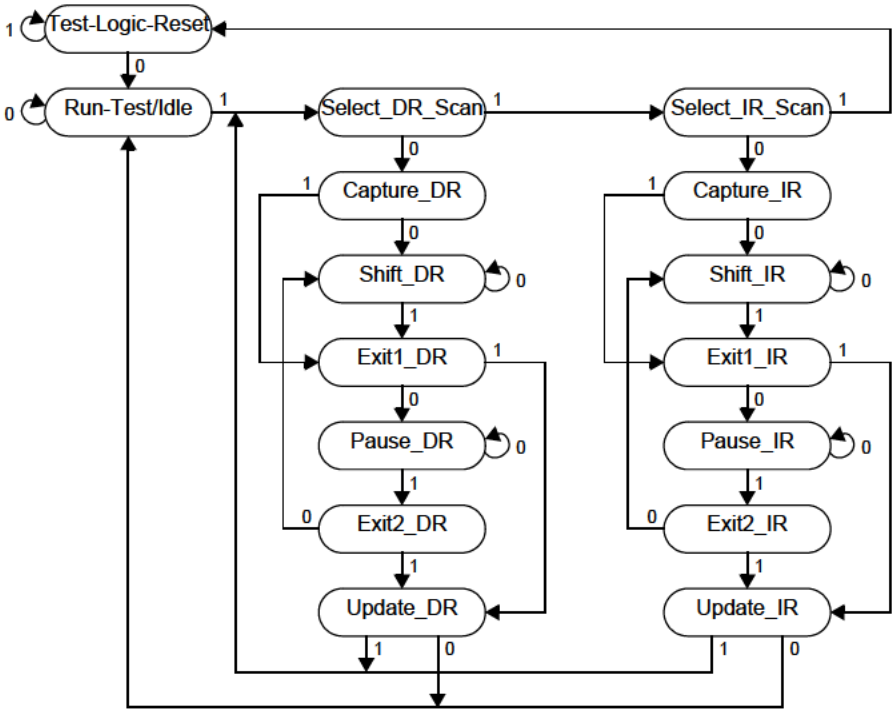
When test access is required, a protocol is applied via the *TMS* and *TCK* inputs, causing the TAP to exit the *Test-Logic-Reset* state and move through the appropriate states. From the *Run-Test/Idle* state, an Instruction register scan or a data register scan can be issued to transition the TAP through the appropriate states shown in Figure 14.2.

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 14.2 TAP Controller State Diagram



14.10.2.1 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.

14.10.2.2 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.

14.10.2.3 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.

14.10.2.4 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.

14.10.2.5 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.

14.10.2.6 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.

14.10.2.7 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.

14.10.2.8 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.

14.10.2.9 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.

14.10.2.10 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.

14.10.2.11 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.

14.10.2.12 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.

14.10.2.13 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.

14.10.2.14 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.

14.10.2.15 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.

14.10.2.16 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.

14.10.3 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 14.8 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
0x15	TCBCONTROLD	Selects the *TCBTCONTROLD* register in the Trace Control Block
0x16	TCBCONTROLE	Selects the *TCBTCONTROLE* register in the Trace Control Block
0x17	FDC	Select Fast Debug Channel
0x1F	BYPASS	Bypass mode

14.10.3.1 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.

14.10.3.2 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.

14.10.3.3 IMPCODE Instruction

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

14.10.3.4 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.

14.10.3.5 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.

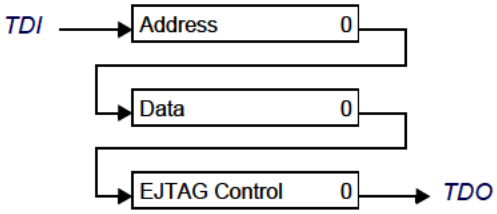
14.10.3.6 CONTROL Instruction

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*.

14.10.3.7 ALL Instruction

This instruction is used to select the concatenation of the Address and Data register, and the EJTAG Control register (ECR) between *TDI* and *TDO*. It can be used in particular to minimize the overhead in switching the instruction in the instruction register. The first bit shifted out is bit 0 of the ECR.

Figure 14.3 Concatenation of the EJTAG Address, Data and Control Registers



14.10.3.8 EJTAGBOOT Instruction

EJTAGBOOT provides a means to enter 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.

When the EJTAGBOOT instruction is given and the Update-IR state is left, the EJTAGBOOT indication will become active. When EJTAGBOOT is active, a core reset will set the ProbTrap, ProbEn and EjtagBrk bits in the EJTAG Control register to 1. This will cause a debug exception that is serviced by the probe immediately after reset is deasserted.

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.

Each CPU has its own TAP controller and thus EJTAGBOOT can be set independently per CPU. Even if a CPU is not activated at cluster reset, EJTAGBOOT on that CPU will still cause a debug exception immediately after reset.

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

14.10.3.9 NORMALBOOT Instruction

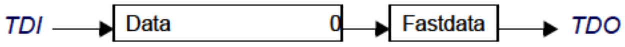
When the NORMALBOOT instruction is given and the Update-IR state is left, then the EJTAGBOOT indication will be cleared. When NORMALBOOT is active (EJTAGBOOT is not active), a core reset will set the ProbTrap, ProbEn and EjtagBrk bits in the EJTAG Control register to 0.

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

14.10.3.10 FASTDATA Instruction

This selects the Data and the Fastdata registers at once, as shown in Figure 14.4.

Figure 14.4 TDI to TDO Path When in Shift-DR State and FASTDATA Instruction is Selected



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 14.9 shows the values of the *PrAcc* and *SPrAcc* bits and the results of a Fastdata access.

Table 14.9 Operation of the FASTDATA Access

Probe Operation	Address Match Check	PrAcc in the Control Register	LSB (SPrAcc) Shifted In	Action in the Data Register	PrAcc Changes to	Lsb Shifted Out	Data Shifted Out
Download using FAST-DATA	Fails	x	x	none	unchanged	0	invalid
	Passes	1	1	none	unchanged	1	invalid
		1	0	write data	0 (SPrAcc)	1	valid (previous) data
		0	x	none	unchanged	0	invalid

Table 14.9 Operation of the FASTDATA Access *(continued)*

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
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.

14.10.3.11 TCBCONTROLA Instruction

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

14.10.3.12 TCBCONTROLB Instruction

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

14.10.3.13 TCBCONTROLC Instruction

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

14.10.3.14 TCBDATA Instruction

This instruction is used to select the TCBDATA register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace Control Block is present. If no TCB is present, then this instruction will select the Bypass register. It should be noted that the TCBDATA register is only an access register to other TCB registers. The width of the TCBDATA register is dependent on the specific TCB register.

14.10.3.15 PCSAMPLE Instruction

This instruction is used to select the PCSAMPLE register to be connected between *TDI* and *TDO*. This register is always implemented.

14.10.3.16 TCBCONTROLD Instruction

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

14.10.3.17 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 Control Block is present. If no TCB is present, then this instruction will select the Bypass register.

14.10.3.18 FDC Instruction

This instruction is used to select the Fast Debug Channel register to be connected between *TDI* and *TDO*. This register is always implemented

14.10.4 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.

14.10.4.1 Fetch/Load and Store From/To the EJTAG Probe Through dmseg

1. The internal hardware latches the requested address into the Address register (in case of the Debug exception: 0xFF20.0200).
2. The internal hardware sets the following bits in the EJTAG Control register:
   PrAcc = 1 (selects Processor Access operation)
   PRnW = 0 (selects processor read operation)
   Psz[1:0] = value depending on the transfer size
3. The EJTAG Probe selects the EJTAG Control register, shifts out this control register’s data and tests the PrAcc status bit (Processor Access): when the PrAcc bit is found 1, it means that the requested address is available and can be shifted out.
4. The EJTAG Probe checks the PRnW bit to determine the required access.
5. The EJTAG Probe selects the Address register and shifts out the requested address.
6. The EJTAG Probe selects the 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 Address register
2. The internal hardware latches the data to be written into the 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 Address register and shifts out the requested address.
7. The EJTAG Probe selects the Data register and shifts out the data that was 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 appropriate address in its memory.
10. The processor detects that PrAcc bit = 0, which means that it is ready to handle a new access.

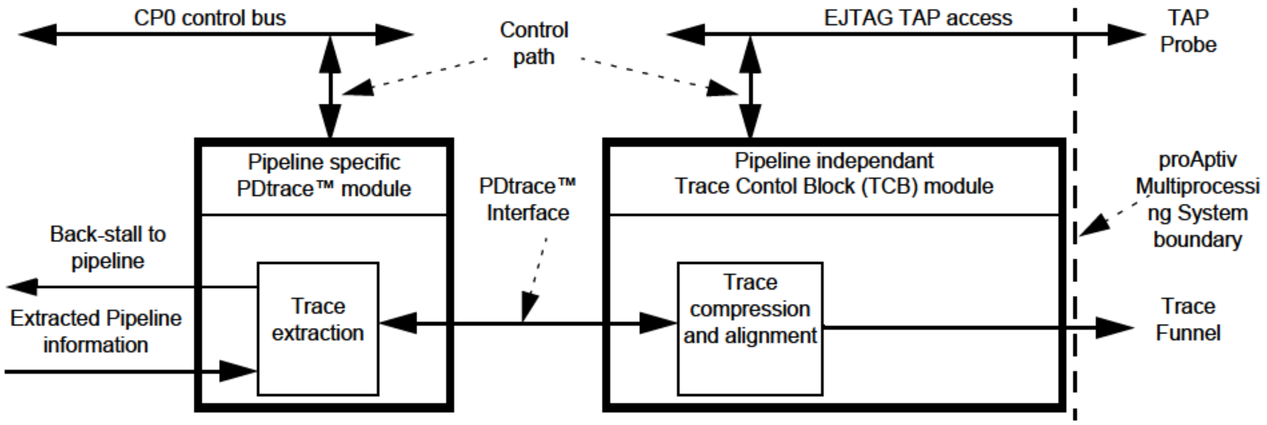
The above examples imply that no reset occurs during the operations, and that Rocc is cleared.

14.11 PDTrace

PDTrace 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). PDtrace is an optional block in the proAptiv Multiprocessing System core. If PDtrace is not implemented, the rest of this chapter is irrelevant. If PDtrace is implemented, the *CP0 Config3*$_{TL}$ bit is set.

There are two primary blocks involved in the PDtrace solution. The pipeline specific part of PDtrace is called the PDtrace module. It extracts the trace information from the processor pipeline, and presents it to a pipeline-independent module called the Trace Control Block (TCB). While working closely together, the two parts of PDtrace are controlled separately by software. Figure 14.5 shows an overview of the PDtrace modules within the core.

Figure 14.5 MIPS® Trace Modules in the proAptiv Multiprocessing System™ Core


CP0 control bus
Control path
EJTAG TAP access
TAP Probe
Pipeline specific PDtrace™ module
PDtrace™ Interface
Pipeline independant Trace Contol Block (TCB) module
proAptiv Multiprocessi ng System boundary
Back-stall to pipeline
Extracted Pipeline information
Trace extraction
Trace compression and alignment
Trace Funnel

To some extent, the two modules both provide similar trace control features, but the access to these features is quite different. The PDtrace controls can only be reached through access to CP0 registers. In general, the PDtrace control registers select what information is captured for tracing.The TCB controls can be reached through EJTAG TAP access or through load/store access to registers mapped in drseg space. The TCB registers control what is traced through the PDtrace™ Interface.

Before describing the PDtrace implemented in the proAptiv Multiprocessing System core, some common terminology and basic features are explained. The remaining sections of this chapter will then provide a more thorough explanation.

14.11.1 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)
```

14.11.2 Software Versus Hardware Control

In some of the specifications and in this text, the terms “software control” and “hardware control” are used to refer to the method for how trace is controlled. Software control is when the CP0 register *TraceControl* is used to select the modes to trace, etc. Hardware control is when the EJTAG register *TCBCONTROLA* in the TCB, via the PDtrace interface, is used to select the trace modes. The *TraceControl*$_{TS}$ bit determines whether software or hardware control is active.

14.11.3 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 PDtrace this is done by providing the minimal cycle-by-cycle information necessary on the PDtrace™ interface for trace regeneration software to reproduce the trace. The following is a summary of the type of information traced:

- Only instructions which complete at the end of the pipeline are traced, and indicated with a completion-flag. The PC is implicitly pointing to the next instruction.

- Load instructions are indicated with a load-flag.

- Store instructions are indicated with a store-flag[1].

- Taken branches are indicated with a branch-taken-flag on the target instruction.

- New PC information for a branch is only traced if the branch target is unpredictable from the static program image.

- When branch targets are unpredictable, only the delta value from current PC is traced, if it is dynamically determined to reduce the number of bits necessary to indicate the new PC. Otherwise the full PC value is traced.

- When a completing instruction is executed in a different processor mode from the previous one, the new processor mode is traced.

- The first instruction is always traced as a branch target, with processor mode and full PC.

- Periodic synchronization instructions are identified with a sync-flag, and traced with the processor mode and full PC.

All the instruction flags above are combined into one 3-bit value, to minimize the bit information to trace. The possible processor modes are explained in Section 14.11.1 "Processor Modes".

The target address is statically predictable for all branch and all jump-immediate instructions. If the branch is taken, then the branch-taken-flag will indicate this. All jump-register instructions and ERET/DERET are instructions which have an unpredictable target address. These will have full/delta PC values included in the trace information. Also treated as unpredictable are PC changes which occur due to exceptions, such as an interrupt, reset, etc.

Trace regeneration software is required to know the static program image in memory, in order to reproduce the dynamic flow with the above information. 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.

14.11.4 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.

[1] A SC (Store Conditional) instruction is not flagged as a store instruction if the load-locked bit prevented the actual store.

- 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.

14.11.5 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 14.11.1 "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*$_{\text{ASID}}$.

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.

14.11.6 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

- Performance counter values, if the optional performance counter trace is enabled.

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.

14.11.6.1 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.

14.11.7 Enable Trace to Probe On-Chip Memory

When trace is On, based on the options listed in Section 14.11.5 “Programmable Processor Trace Mode Options”, the trace information is continuously sent on the PDtrace™ interface to the TCB. The TCB must be enabled to transmit the trace information to the Trace funnel by having the *TCBCONTROLB*$_{EN}$ bit set. It is possible to enable and disable the TCB in a number of 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 store to drseg space.

14.11.8 Enabling PDtrace

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.

14.11.8.1 Trace Trigger from EJTAG Hardware Instruction/Data Breakpoints

If hardware instruction/data simple breakpoints are implemented in the proAptiv Multiprocessing System core, then these breakpoints 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 14.14.2.5 “Instruction Breakpoint Control n (IBCn) Register” and Section 14.14.3.5 “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.

14.11.8.2 Turning On PDtrace™ Trace

Trace enabling and disabling from software is similar to the hardware method, with the exception that the bits in the control register are used instead of the input enable signals from the TCB. The *TraceControl*$_{TS}$ bit controls whether hardware (via the TCB), or software (via the *TraceControl* register) controls tracing functionality.

Trace is turned on when the following expression evaluates true:

```
(
    (
        (TraceControl$_{TS}$ and TraceControl$_{On}$) or
        ((not TraceControl$_{TS}$) and TCBCONTROLA$_{On}$)
    )
    and
    (MatchEnable or TriggerEnable)
)
```

where,

```
MatchEnable ←
(
    TraceControl$_{TS}$
    and
        ((TraceControl2$_{TCV}$ and (TraceControl2$_{TCNUM}$ equal TCIDofCompletedInst)) or
         ((not TraceControl2$_{TCV}$) and TraceControl2$_{CPUIdV}$ and
             (TraceControl2$_{CPUId}$ equal CPUIDofCompletedInst )) or
         (TraceControl2$_{TCV}$ nor TraceControl2$_{CPUIdV}$ ))
    and
    (
        TraceControl$_{G}$ or
        (((TraceControl$_{ASID}$ xor EntryHi$_{ASID}$) and (not TraceControl$_{ASID\_M}$)) = 0)
    )
    and
    (
        (TraceControl$_{U}$ and UserMode)       or
        (TraceControl$_{S}$ and SupervisorMode) or
        (TraceControl$_{K}$ and KernelMode)     or
        (TraceControl$_{E}$ and ExceptionMode)  or
        (TraceControl$_{D}$ and DebugMode)
    )
)
or
(
    (not TraceControl$_{TS}$)
    and
        ((TCBCONTROLC$_{TCV}$ and (TCBCONTROLC$_{TCNUM}$ equal TCIDofCompletedInst)) or
         ((not TCBCONTROLC$_{TCV}$) and TCBCONTROLC$_{CPUIdV}$ and
             (TCBCONTROLC$_{CPUId}$ equal CPUIDofCompletedIns )) or
         (TCBCONTROLC$_{TCV}$ nor TCBCONTROLC$_{CPUIdV}$ ))
    and
    (TCBCONTROLA$_{G}$ or (TCBCONTROLA$_{ASID}$ = EntryHi$_{ASID}$))
    and
    (
        (TCBCONTROLA$_{U}$ and UserMode)        or
        (TCBCONTROLA$_{S}$ and SupervisorMode)  or
        (TCBCONTROLA$_{K}$ and KernelMode)      or
        (TCBCONTROLA$_{E}$ and ExceptionMode)   or
        (TCBCONTROLA$_{DM}$ and DebugMode)
    )
)
```

and where,

```
TriggerEnable ←
(
    DBCi$_{TE}$          and
    DBS$_{BS[i]}$        and
    TraceBPC$_{DE}$      and
    (TraceBPC$_{DBPOn[i]}$ = 1)
)
or
(
    IBCi$_{TE}$          and
    IBS$_{BS[i]}$        and
    TraceBPC$_{IE}$      and
    (TraceBPC$_{IBPOn[i]}$ = 1)
)
```

As seen in the expression above, trace can be turned on only if the master switch *TraceControl*$_{On}$ or *TCBCONTROLA*$_{On}$ is first asserted.

Once this is asserted, there are two ways to turn on tracing. The first way, the *MatchEnable* expression, uses the input enable signals from the TCB or the bits in the *TraceControl* register. This tracing is done over general program areas. For example, all of the user-level code for a particular process (if ASID is specified), and so on.

The second way to turn on tracing, the *TriggerEnable* expression, is from the processor side using the EJTAG hardware breakpoint triggers. If EJTAG is implemented, and hardware breakpoints can be set, then using this method enables finer grain tracing control. It is possible to send a trigger signal that turns on tracing at a particular instruction. For example, it would be possible to trace a single procedure in a program by triggering on trace at the first instruction, and triggering off trace at the last instruction.

The easiest way to unconditionally turn on trace is to assert either hardware or software tracing and the corresponding trace on signal with other enables. For example, with *TraceControl*$_{TS}$=0, i.e., hardware controlled tracing, assert *TCBCONTROLA*$_{On}$, *TCBCONTROLA*$_{G}$, and all the other signals in the second part of expression *MatchEnable*. To only trace when a particular process with a known ASID is executing, assert *TCBCONTROLA*$_{On}$, the correct *TCBCONTROLA*$_{ASID}$ value, and all of *TCBCONTROLA*$_{U}$, *TCBCONTROLA*$_{K}$, *TCBCONTROLA*$_{E}$, and *TCBCONTROLA*$_{DM}$. (If it is known that the particular process is a user-level process, then it would be sufficient to only assert *TCBCONTROLA*$_{U}$ for example). When using the EJTAG hardware triggers to turn trace on and off, it is best if *TCBCONTROLA*$_{On}$ is asserted and all the other processor mode selection bits in *TCBCONTROLA* are turned off. This would be the least confusing way to control tracing with the trigger signals. Tracing can be controlled via software with the *TraceControl* register in a similar manner.

14.11.8.3 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.

14.12 PDtrace 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,

14.12.1 FIFO Logic in PDtrace and TCB Modules

Both the PDtrace module and the TCB module contain a fifo. This might seem like extra overhead, but there are good reasons for this. The vast majority of the information compression happens in the PDtrace module. Any data information, like PC and load/store address values (delta or full), load/store data and processor mode changes, are all sent on the same 32-bit data bus to the TCB on the internal PDtrace™ interface. When an instruction requires more than 32 bits of information to be traced properly, the PDtrace fifo will buffer the information, and send it on subsequent clock cycles.

In the TCB, the on-chip trace memory is defined as a 64-bit wide synchronous memory running at core-clock speed. In this case the FIFO is not needed. For off-chip trace through the Trace Probe, the FIFO comes into play, because only a limited number of pins (4, 8 or 16) exist. Also the speed of the Trace Probe interface can be different (either faster or slower) from that of the proAptiv Multiprocessing System core. So for off-chip tracing, a specific TCB TW FIFO is needed.

14.12.2 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 to overflow from time to time. There are two ways to handle this case:

1. Allow the overflow to happen, and thereby lose some information from the trace data.
2. Prevent the overflow by back-stalling the core until the FIFO has enough empty slots to accept new trace data.

The PDtrace fifo option is controlled by either the *TraceControl*$_{IO}$ or the *TCBCONTROLA*$_{IO}$ bit, depending on the setting of *TraceControl*$_{TS}$ bit.

The first option is free of any cycle-by-cycle change whether trace is turned on or not. This is achieved at the cost of potentially losing trace information. After an overflow, the fifo is completely emptied, and the next instruction is traced as if it was the start of the trace (processor mode and full PC are traced). This guarantees that only the un-traced fifo information is lost.

The second option guarantees that all the trace information is traced to the TCB. In some cases this is then achieved by back-stalling the core pipeline, giving the PDtrace fifo time to empty enough room in the fifo to accept new trace information from a new instruction. This option can obviously change the real-time behavior of the core when tracing is turned on.

If PC trace information is the only thing enabled (in *TraceControl2*$_{MODE}$ or *TCBCONTROLC*$_{MODE}$, depending on the setting of *TraceControl*$_{TS}$), and Trace of all branches is turned off (via *TraceControl*$_{TB}$ or *TCBCONTROLA*$_{TB}$, depending on the setting of *TraceControl*$_{TS}$), then the fifo is unlikely to overflow very often, if at all. This is of course very dependent on the code executed, and the frequency of exception handler jumps, but with this setting there is very little information overhead.

14.12.3 Handling of FIFO Overflow in the TCB

The TCB also holds a fifo, used to buffer the TW's which are sent off-chip through the Trace Probe. The data width of the probe can be either 4, 8 or 16 pins, and the speed of these data pins can be from 16 times the core-clock to 1/4 of the core clock (the trace probe clock always runs at a double data rate multiple to the core-clock). See Section 14.12.3.1 "Probe Width and Clock-ratio Settings" for a description of probe width and clock-ratio options. The combination between the probe width (4, 8 or 16) and the data speed, allows for data rates through the trace probe from 256 bits per core-clock cycle down to only 1 bit per core-clock cycle. The high extreme is not likely to be supported in any implementation, but the low one might be.

The data rate is an important figure when the likelihood of a TCB fifo overflow is considered. The TCB will at maximum produce one full 64-bit TW per core-clock cycle. This is true for any selection of trace mode in *TraceControl2*$_{\mathrm{MODE}}$ or *TCBCONTROLC*$_{\mathrm{MODE}}$. 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. Similar to the PDtrace module FIFO, this can be handled in two ways:

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 (*PDI_StallSending*). This will in turn stall the core pipeline.

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 14.12.2 "Handling of FIFO Overflow in the PDtrace Module" and below in Section 14.12.4 "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.

14.12.3.1 Probe Width and Clock-ratio Settings

Note: the registers called out in this section are located in the Coherence Manager TAP described in Chapter 15, Multi-CPU Debug. All of these fields are reserved in the proAptiv core TAP registers.

The actual number of data pins (4, 8 or 16) is defined by the CM TAP *TCBCONFIG*$_{\mathrm{PW}}$ 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 CM TAP*TCBCONTROLB*$_{\mathrm{CR}}$ field. The legal range for the clock ratio is defined in CM TAP *TCBCONFIG*$_{\mathrm{CRMax}}$ and CM TAP *TCBCONFIG*$_{\mathrm{CRMin}}$ (both values inclusive). If the CM TAP*TCBCONTROLB*$_{\mathrm{CR}}$ bit is set to an unsupported value, the result is UNPREDICABLE. The maximum possible value for CM TAP *TCBCONFIG*$_{\mathrm{CRMax}}$ field is 8:1 (*TR_CLK* is running 8 times faster than core-clock). The minimum possible value for CM TAP *TCBCONFIG*$_{\mathrm{CRMin}}$ field is 1:8 (*TR_CLK* is running at one eighth of the core-clock).

14.12.4 Adding Cycle Accurate Information to the Trace

Depending on the trace regeneration software, it is possible to obtain the exact cycle time relationship between each instruction in the trace. This information is added to the trace, when the *TCBCONTROLB*$_{CA}$ bit is set. The overhead on the trace information is a little more than one extra bit per core-clock cycle.

This setting only affects the TCB module and not the PDtrace module. The extra bit therefore only affects the likelihood of the TCB FIFO overflowing.

14.13 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.

In PC sampling, the PC is sampled periodically and sent to the TAP register. Note that although the PC sampling function can be used both with and without a probe, if a probe is not connected, the sampled information cannot be read out since the TAP registers can only be read when a probe is connected. Therefore, MIPS recommends using the PC sampling capability only when a probe is connected.

The presence or absence of the PC Sampling feature is available in the Debug Control register as bit 9 (PCS).The sampled PC values are written into a TAP register. The old value in the TAP register is overwritten by a new value even if this register has not be read out by the debug probe. The sample rate is specified in a manner similar to the PDtrace synchronization period, with three bits. These bits in the Debug Control register are 8:6 and called PCSR (PC Sample Rate). These three bits take the value 2^5 to 2^{12} similar to SyncPeriod. Note that the processor samples PC even when it is asleep, that is, in a WAIT state. This permits an analysis of the amount of time spent by a processor in WAIT state which may be used for example to revert to a low power mode during the non-execution phase of a real-time application.

The sampled values includes a new data bit, the PC, the ASID of the sampled PC as well as the Enhanced Virtual Address (EVA) K/U bit. Figure 14.6 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 14.6 TAP Register PCsample Format

49 42	41	40 33	32 1	0
R	K/U	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.

14.13.1 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.

14.14 EJTAG Registers

The following subsections describe the EJTAG register interface.

14.14.1 General Purpose Control and Status

The following register provide general control and status information for EJTAG.

14.14.1.1 Debug Control Register

The Debug Control Register (*DCR*) register controls and provides information about debug issues and is always provided with the proAptiv Multiprocessing System 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 proAptiv Multiprocessing System 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 that the probe will service dmseg accesses. The reset value in the table below takes effect on any CPU reset.

Figure 14.7 Debug Control Register Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0		ENM	0		PCIM	PCno ASID	DASQ	DASe	DAS	0			FDC Impl	Data Brk	Inst Brk

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IVM	DVM	0		RD Vec	CBT	PCS	PCR			PCSe	IntE	NMIE	NMI pend	SRstE	Prob En

Table 14.10 Debug Control Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:30	Must be written as zeros; return zeros on reads.	0	0

Table 14.10 Debug Control Register Field Descriptions *(continued)*

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>ENM</td><td>29</td><td>Endianess in which the processor is running in kernel and Debug Mode:<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Little endian</td></tr><tr><td>1</td><td>Big endian</td></tr></table></td><td>R</td><td>Preset</td></tr>
<tr><td>0</td><td>28:27</td><td>Must be written as zeros; return zeros on reads.</td><td>0</td><td>0</td></tr>
<tr><td>PCIM</td><td>26</td><td>Configure PC Sampling to capture all executed addresses or only those that miss the instruction cache
This feature is not supported and this bit will read as 0..<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>All PC’s captured</td></tr><tr><td>1</td><td>Capture only PC’s that miss the cache.</td></tr></table></td><td>R</td><td>0</td></tr>
<tr><td>PCnoASID</td><td>25</td><td>Controls whether the PCSAMPLE scan chain includes or omits the ASID field
ASID is always included so this bit will read as 0.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>ASID included in PCSAMPLE scan</td></tr><tr><td>1</td><td>ASID omitted from PCSAMPLE scan</td></tr></table></td><td>R</td><td>0</td></tr>
<tr><td>DASQ</td><td>24</td><td>Qualifies Data Address Sampling using a data breakpoint.
Data address sampling is not supported so this bit will read as 0<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>All data addresses are sampled</td></tr><tr><td>1</td><td>Sample matches of data breakpoint 0</td></tr></table></td><td>R</td><td>0</td></tr>
<tr><td>DASe</td><td>23</td><td>Enables Data Address Sampling
Data address sampling is not supported so this bit will read as 0<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Data Address sampling disabled.</td></tr><tr><td>1</td><td>Data Address sampling enabled.</td></tr></table></td><td>R</td><td>0</td></tr>
</table>

Table 14.10 Debug Control Register Field Descriptions *(continued)*

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>DAS</td><td>22</td><td>Indicates if the Data Address Sampling feature is implemented.
Data address sampling is not supported so this bit will read as 0.<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No DA Sampling implemented</td></tr><tr><td>1</td><td>DA Sampling implemented</td></tr></table></td><td>R</td><td>0</td></tr>
<tr><td>0</td><td>21:19</td><td>Must be written as zeros; return zeros on reads.</td><td>0</td><td>0</td></tr>
<tr><td>FDCImpl</td><td>18</td><td>Indicates if the fast debug channel is implemented<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No fast debug channel implemented</td></tr><tr><td>1</td><td>Fast debug channel implemented</td></tr></table></td><td>R</td><td>1</td></tr>
<tr><td>DataBrk</td><td>17</td><td>Indicates if data hardware breakpoint is implemented:<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No data hardware breakpoint implemented</td></tr><tr><td>1</td><td>Data hardware breakpoint implemented</td></tr></table></td><td>R</td><td>Preset</td></tr>
<tr><td>InstBrk</td><td>16</td><td>Indicates if instruction hardware breakpoint is implemented:<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No instruction hardware breakpoint implemented</td></tr><tr><td>1</td><td>Instruction hardware breakpoint implemented</td></tr></table></td><td>R</td><td>Preset</td></tr>
<tr><td>IVM</td><td>15</td><td>Indicates if inverted data value match on data hardware breakpoints is implemented:<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No inverted data value match on data hardware breakpoints implemented</td></tr><tr><td>1</td><td>Inverted data value match on data hardware breakpoints implemented</td></tr></table></td><td>R</td><td>0</td></tr>
</table>

Table 14.10 Debug Control Register Field Descriptions *(continued)*

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>DVM</td><td>14</td><td>Indicates if a data value store on a data value breakpoint match is implemented:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No data value store on a data value breakpoint match implemented</td></tr>
<tr><td>1</td><td>Data value store on a data value breakpoint match implemented</td></tr>
</table></td><td>R</td><td>0</td></tr>
<tr><td>0</td><td>13:12</td><td>Must be written as zeros; return zeros on reads.</td><td>0</td><td>0</td></tr>
<tr><td>RDVec</td><td>11</td><td>Enables relocation of the debug exception vector. The value in the DebugVectorAddr register is used for EJTAG exceptions when ProbTrap = 0,and RDVec = 1.</td><td>R/W</td><td>0</td></tr>
<tr><td>CBT</td><td>10</td><td>Indicates if complex breakpoint block is implemented:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No complex breakpoint block implemented</td></tr>
<tr><td>1</td><td>Complex breakpoint block implemented</td></tr>
</table></td><td>R</td><td>0</td></tr>
<tr><td>PCS</td><td>9</td><td>Indicates if the PC Sampling feature is implemented.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No PC Sampling implemented</td></tr>
<tr><td>1</td><td>PC Sampling implemented</td></tr>
</table></td><td>R</td><td>1</td></tr>
<tr><td>PCR</td><td>8:6</td><td>PC Sampling rate. Values 0 to 7 map to values 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 respectively. The external probe or software is allowed to set this value to the desired sample rate.</td><td>R/W</td><td>7</td></tr>
<tr><td>PCSe</td><td>5</td><td>If the PC sampling feature is implemented, then indicates whether PC sampling is initiated or not. That is, a value of 0 indicates that PC sampling is not enabled and when the bit value is 1, then PC sampling is enabled and the counters are operational.</td><td>R/W</td><td>0</td></tr>
<tr><td>IntE</td><td>4</td><td>Hardware and software interrupt enable for Non-Debug Mode, in conjunction with other disable mechanisms:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Interrupt disabled</td></tr>
<tr><td>1</td><td>Interrupt enabled depending on other enabling mechanisms</td></tr>
</table></td><td>R/W</td><td>1</td></tr>
</table>

Table 14.10 Debug Control Register Field Descriptions *(continued)*

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>NMIE</td><td>3</td><td>Non-Maskable Interrupt (NMI) enable for Non-Debug Mode:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>NMI disabled</td></tr>
<tr><td>1</td><td>NMI enabled</td></tr>
</table></td><td>R/W</td><td>1</td></tr>
<tr><td>NMIpend</td><td>2</td><td>Indication for pending NMI:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No NMI pending</td></tr>
<tr><td>1</td><td>NMI pending</td></tr>
</table></td><td>R</td><td>0</td></tr>
<tr><td>SRstE</td><td>1</td><td>Controls soft reset enable:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Soft reset masked for soft reset sources dependent on implementation</td></tr>
<tr><td>1</td><td>Soft reset is fully enabled</td></tr>
</table></td><td>R/W</td><td>1</td></tr>
<tr><td>ProbEn</td><td>0</td><td>Indicates value of the ProbEn value in the ECR register:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No access should occur to the dmseg segment</td></tr>
<tr><td>1</td><td>Probe services accesses to the dmseg segment</td></tr>
</table>
Bit is read-only (R) and reads as zero if not implemented.</td><td>R</td><td>Same value as ProbEn in ECR</td></tr>
</table>

14.14.1.2 DebugVectorAddr Register

This register allows an alternate debug exception vector address to be specified, which can enable placing a debug monitor program into RAM for much faster execution than the default ROM address. This register is memory mapped at an offset of 0x00020 within the DRSEG memory segment.

Figure 14.8 shows the register format and Table 14.12 describes the fields in this register.

Table 14.11 shows the different debug exception vector locations that are possible.

Table 14.11 Debug Exception Vectors

$\text{ECR}_{\text{ProbTrap}}$	$\text{DCR}_{\text{RdVec}}$	$\text{Config5}_{\text{K}}$	SI_UseExceptionBase	Cache Error?	Debug Exception Vector
1	x	x	x	x	0xFF20_0200
0	1	0	x	0	2’b10 || DebugVectorAddr[29:0]
0	1	1	x	0	DebugVectorAddr[31:0]

Table 14.11 Debug Exception Vectors

ECR$_{\text{ProbTrap}}$	DCR$_{\text{RdVec}}$	Config5$_{\text{K}}$	SI_UseExceptionBase	Cache Error?	Debug Exception Vector
0	1	0	x	1	3’b101 || DebugVectorAddr[28:0]
0	1	1	x	1	DebugVectorAddr[31:0]
0	0	0	1	0	2’b10 || SI_ExceptionBase[29:12] || 0x480
0	0	1	1	0	SI_ExceptionBase[31:12] || 0x480
0	0	0	1	1	3’b101 || SI_ExceptionBase[28:12] || 0x480
0	0	1	1	1	SI_ExceptionBase[31:12] || 0x480
0	0	x	0	x	0xBFC0_0480

Figure 14.8 DebugVectorAddr Register Format

31 7	6	5 1	0
DebugVectorAddr	WG	0	ISA

Table 14.12 DebugVectorAddr Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
DebugVectorAddr	31	Programmable Debug Exception Vector Address. Note that bits 31:30 have default values of 1 and 0 respectively and can only be written when the WG bit is set. If the WG bit is cleared, these bits are read-only and retain their previous values. These two bits can be written whenever the WG bit is set, regardless of the state of *Config5*$_{K}$.	R/W	1
	30		R/W	0
	29:7		R/W	0x7f8009 (corresponds to 0xbfc00480)
WG	6	Write gate. When the WG bit is set, the DebugVectorAddr field is expanded to include bits 31:30 to facilitate programmable memory segmentation controlled by the *SegCtl0* through *SegCtl2* registers. When the WG bit is cleared, bits 31:30 of this register are not writeable and remain unchanged from the last time that WG was cleared.	R/W	Externally Set
0	5:1	Ignored on write, returns zero on read.	R	0
ISA	0	ISA mode to be used for debug exception handler. Only used on cores implementing microMIPS.	R	0

14.14.2 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 with addresses as shown in Table 14.13.

Table 14.13 Addresses for Instruction Breakpoint Registers

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

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

14.14.2.1 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 14.9 IBS Register Format

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

Table 14.14 IBS Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>Res</td><td>31</td><td>Must be written as zero; returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>ASIDsup</td><td>30</td><td>Hardware and software interrupt enable for Non-Debug Mode, in conjunction with other disable mechanisms:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>ASID compare not supported</td></tr>
<tr><td>1</td><td>ASID compare supported (IBASIDn register implemented)</td></tr>
</table></td><td>R</td><td>1</td></tr>
<tr><td>Res</td><td>29:28</td><td>Must be written as zero; returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>BCN</td><td>27:24</td><td>Number of instruction breakpoints implemented.</td><td>R</td><td>2 or 4</td></tr>
<tr><td>Res</td><td>23:4</td><td>Must be written as zero; returns zero on read.</td><td>R</td><td>0</td></tr>
</table>

Table 14.14 IBS Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
BS	3:0	Break status for breakpoint n is at BS[n], with n from 0 to 3. The bit is set to 1 when the corresponding breakpoint is enabled and the condition has matched. If only two instruction breakpoints are implemented, bits 2 and 3 must be written as zero and will return zero on read.	R/W	Undefined

14.14.2.2 Instruction Breakpoint Address n (IBAn) Register

Compliance Level: Implemented only for implemented instruction breakpoints.

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

Figure 14.10 IBAn Register Format

31 … 1	0
IBA	ISA

Table 14.15 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

14.14.2.3 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 14.11 IBMn Register Format

31 … 1	0
IBM	ISAM

Table 14.16 IBMn Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>IBM</td><td>31:1</td><td>Instruction breakpoint address mask for condition:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Corresponding address bit not masked</td></tr>
<tr><td>1</td><td>Corresponding address bit masked</td></tr>
</table></td><td>R/W</td><td>Undefined</td></tr>
<tr><td>ISAM</td><td>0</td><td>Instruction breakpoint ISA mode mask for condition:
condition:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Corresponding address bit not masked</td></tr>
<tr><td>1</td><td>Corresponding address bit masked</td></tr>
</table>
0: ISA mode considered for match condition
1: ISA mode masked</td><td>R/W</td><td>Undefined</td></tr>
</table>

14.14.2.4 Instruction Breakpoint ASID n (IBASIDn) Register

Compliance Level: Implemented only for implemented instruction breakpoints.

This register is used to define an ASID value to be used in the match expression.

Figure 14.12 IBASIDn Register Format

31 8	7 0
Res	ASID

Table 14.17 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

14.14.2.5 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 14.13 IBCn Register Format

31 24	23	22 3	2	1	0
R	ASIDuse	R	TE	R	BE

Table 14.18 IBCn Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>R</td><td>31:24</td><td>Must be written as zero; returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>ASIDuse</td><td>23</td><td>Use ASID value in compare for instruction breakpoint n:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Don’t use ASID value in compare</td></tr>
<tr><td>1</td><td>Use ASID value in compare</td></tr>
</table></td><td>R/W</td><td>Undefined</td></tr>
<tr><td>R</td><td>22:3</td><td>Must be written as zero; returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>TE</td><td>2</td><td>Trigger-only Enable. This field is ignored when BE is set. When BE is cleared and TE is set, instruction breakpoint n is enabled, but will not signal a debug exception.</td><td>R/W</td><td>0</td></tr>
<tr><td>R</td><td>1</td><td>Must be written as zero; returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>BE</td><td>0</td><td>Breakpoint Enable. When set, instruction breakpoint n is enabled and will signal a debug exception when its condition matches.</td><td>R/W</td><td>0</td></tr>
</table>

14.14.3 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 14.19.

Table 14.19 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.

14.14.3.1 Data Breakpoint Status (DBS) Register

Compliance Level: Implemented if data breakpoints are implemented.

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

The ASIDsup field indicates whether ASID compares are supported.

Figure 14.14 DBS Register Format

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

Table 14.20 DBS Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>Res</td><td>31</td><td>Must be written as zero; returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>ASID</td><td>30</td><td>Indicates that ASID compares are supported in data breakpoints.
n:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Don’t use ASID value in compare</td></tr>
<tr><td>1</td><td>Use ASID value in compare</td></tr>
</table>
0: Not supported
1: Supported</td><td>R</td><td>TLB MMU - 1</td></tr>
<tr><td>Res</td><td>29:28</td><td>Must be written as zero; returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>BCN</td><td>27:24</td><td>Number of data breakpoints implemented.</td><td>R</td><td>1 or 2</td></tr>
<tr><td>Res</td><td>23:2</td><td>Must be written as zero; returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>BS</td><td>1:0</td><td>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 and the condition has matched. If only one data breakpoint is implemented, bit 1 must be written as 0 and will return 0 on reads.</td><td>R/W0</td><td>Undefined</td></tr>
</table>

14.14.3.2 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 14.15 DBAn Register Format

31 0
DBA

Table 14.21 DBAn Register Field Descriptions

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

14.14.3.3 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 14.16 DBMn Register Format

31 … 0
DBM

Table 14.22 DBMn Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>DBM</td><td>31:0</td><td>Data breakpoint address mask for condition:
n:<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Don’t use ASID value in compare</td></tr><tr><td>1</td><td>Use ASID value in compare</td></tr></table>0: Corresponding address bit not masked
1: Corresponding address bit masked</td><td>R/W</td><td>Undefined</td></tr>
</table>

14.14.3.4 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 14.17 DBASIDn Register Format

31 … 8	7 … 0
Res	ASID

Table 14.23 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

14.14.3.5 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 14.18 DBCn Register Format

31 24	23	22	21 14	13	12	11 4	3	2	1	0
R	ASIDuse	R	BAI	NoSB	NoLB	BLM	R	TE	R	BE

Table 14.24 DBCn Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>R</td><td>31:24</td><td>Must be written as zero; returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>ASIDuse</td><td>23</td><td>Use ASID value in compare for data breakpoint n:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Don’t use ASID value in compare</td></tr>
<tr><td>1</td><td>Use ASID value in compare</td></tr>
</table></td><td>R/W</td><td>Undefined</td></tr>
<tr><td>R</td><td>22</td><td>Must be written as zero; returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>BAI</td><td>21:14</td><td>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.:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Condition depends on access to corresponding byte</td></tr>
<tr><td>1</td><td>Access for corresponding byte is ignored</td></tr>
</table></td><td>R/W</td><td>Undefined</td></tr>
<tr><td>NoSB</td><td>13</td><td>Controls if condition for data breakpoint is fulfilled on a store transaction:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Condition may be fulfilled on store transaction</td></tr>
<tr><td>1</td><td>Condition is never fulfilled on store transaction</td></tr>
</table></td><td>R/W</td><td>Undefined</td></tr>
<tr><td>NoLB</td><td>12</td><td>Controls if condition for data breakpoint is fulfilled on a load transaction:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Condition may be fulfilled on load transaction</td></tr>
<tr><td>1</td><td>Condition is never fulfilled on load transaction</td></tr>
</table></td><td>R/W</td><td>Undefined</td></tr>
</table>

Table 14.24 DBCn Register Field Descriptions*(continued)*

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>BLM</td><td>11:4</td><td>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.:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Compare corresponding byte lane</td></tr>
<tr><td>1</td><td>Mask corresponding byte lane</td></tr>
</table></td><td>R/W</td><td>Undefined</td></tr>
<tr><td>R</td><td>3</td><td>Must be written as zero; returns zero on reads.</td><td>R</td><td>0</td></tr>
<tr><td>TE</td><td>2</td><td>Trigger-only Enable. This field is ignored when *BE* is set. When *BE* is cleared and TE is set, data breakpoint n is enabled, but will not signal a debug exception.</td><td>R/W</td><td>0</td></tr>
<tr><td>R</td><td>1</td><td>Must be written as zero; returns zero on reads.</td><td>R</td><td>0</td></tr>
<tr><td>BE</td><td>0</td><td>Breakpoint Enable. When set, data breakpoint n is enabled and will signal a debug exception when its condition matches.</td><td>R/W</td><td>0</td></tr>
</table>

14.14.3.6 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 14.19 DBVn Register Format

31 … 0
DBV

Table 14.25 DBVn Register Field Descriptions

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

14.14.3.7 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 14.20 DBVHn Register Format

31 … 0
DBVH

Table 14.26 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

14.14.4 EJTAG TAP Registers

The EJTAG TAP Module has one Instruction register and a number of data registers, all accessible through the TAP:

14.14.4.1 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 14.8.

14.14.4.2 Data Registers Overview

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

This description applies in general to the following data registers:

- Bypass Register
- Device Identification Register
- Implementation Register
- EJTAG Control Register (ECR)
- Address Register
- Data Register
- FastData Register

14.14.4.3 Bypass Register

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

14.14.4.4 Device Identification (ID) Register

The *Device Identification* register is defined by IEEE 1149.1, to identify the device's manufacturer, part number, revision, and other device-specific information. Table 14.27 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. Note that this register contains only device manufacturer information and should not be used in an attempt to determine the EJTAG or PDTrace revisions of the device.

Figure 14.21 Device Identification Register Format

31 28	27 12	11 1	0
Version	PartNumber	ManufID	R

Table 14.27 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

14.14.4.5 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. The EJTAG probe uses this TAP register to determine the EJTAG version of the device. Software has no access to this register and must use the CP0 Debug register to determine the EJTAG version.

Figure 14.22 Implementation Register Format

31 29	28 25	24	23 21	20 17	16	15	14	13 11	10 1	0
EJTAGver	reserved	DINTsup	ASIDsize	reserved	MIPS16	0	NoDMA	TYPE	TYPEINFO	R

Table 14.28 Implementation Register Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
EJTAGver	31:29	Indicates EJTAG version 5.0.	R	5
reserved	28:25	Reserved. Must be written as zeros; returns zeros on reads.	R	0

Table 14.28 Implementation Register Descriptions*(continued)*

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>DINTsup</td><td>24</td><td>DINT Signal Supported from Probe
This bit indicates if the DINT signal from the probe is supported:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>DINT signal from the probe is not supported.</td></tr>
<tr><td>1</td><td>Probe can use DINT signal to make debug interrupt.</td></tr>
</table></td><td>R</td><td><i>EJ_DINTsup</i></td></tr>
<tr><td>ASIDsize</td><td>23:21</td><td>Size of ASID field in implementation:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No ASID in implementation</td></tr>
<tr><td>1</td><td>Reserved</td></tr>
<tr><td>2</td><td>8-bit ASID</td></tr>
<tr><td>3</td><td>Reserved</td></tr>
</table></td><td>R</td><td>2</td></tr>
<tr><td>R</td><td>20:17</td><td>Reserved</td><td>R</td><td>0</td></tr>
<tr><td>MIPS16</td><td>16</td><td>Indicates whether MIPS16 is implemented:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No MIPS16 support</td></tr>
<tr><td>1</td><td>MIPS16 implemented</td></tr>
</table></td><td>R</td><td>1</td></tr>
<tr><td>R</td><td>15</td><td>Reserved. Must be written as zeros; returns zeros on reads.</td><td>R</td><td>0</td></tr>
<tr><td>NoDMA</td><td></td><td></td><td>R</td><td>1</td></tr>
<tr><td>TYPE</td><td>13:11</td><td>Indicates what type of entity is associated with this TAP and whether the TypeInfo field exists.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>TYPEINFO field not implemented.
Legacy value - probably attached to a CPU.</td></tr>
<tr><td>1</td><td>This TAP is attached to a CPU and the TYPEINFO field reflects $EBase_{CPUNUM}$.</td></tr>
<tr><td>2</td><td>This TAP is attached to a Trace-Master and the TypeInfo field is not used.</td></tr>
<tr><td>3</td><td>Reserved</td></tr>
</table></td><td>R</td><td>1</td></tr>
</table>

Table 14.28 Implementation Register Descriptions*(continued)*

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>TYPEINFO</td><td>10:1</td><td>Identifier information specific to the type of entity associated with this TAP. The attached entity is specified by the TYPE field.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>CPU</td><td>Reflects <i>EBase</i>$_{CPUNUM}$ of the associated CPU.</td></tr>
<tr><td>Others</td><td>Reserved.</td></tr>
</table>
</td><td>R</td><td>1</td></tr>
<tr><td>R</td><td>0</td><td>Reserved. Must be written as zeros; returns zeros on reads.</td><td>R</td><td>0</td></tr>
</table>

14.14.4.6 EJTAG Control Register

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

The EJTAG Control register is not updated in the *Update-DR* state unless the Reset occurred (Rocc) bit 31, is either 0 or written to 0. This is in order to ensure prober handling of processor accesses.

The value used for reset indicated in the table below takes effect on CPU resets, but not on TAP controller resets (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* is supplied. The first 5 *TCK* clocks after CPU reset may result in reset of the bits, due to synchronization between clock domains.

Figure 14.23 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	R	Doze	Halt	PerRst	PRnW	PrAcc	Res	PrRst	ProbEn	ProbTrap	Res	EjtagBrk	Res	DM	Res

Table 14.29 EJTAG Control Register Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>Rocc</td><td>31</td><td>Reset Occurred
The bit indicates if a CPU reset has occurred:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No reset occurred since bit last cleared</td></tr>
<tr><td>1</td><td>Reset occurred since bit last cleared</td></tr>
</table>
The Rocc bit will remain set to 1 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 <i>Update-DR</i> state unless Rocc is 0 or written to 0, in order to ensure proper handling of processor access following reset.</td><td>R/W</td><td>1</td></tr>
<tr><td>Psz[1:0]</td><td>30:29</td><td>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.
<table>
<tr><th>PAA[1:0]</th><th>Psz[1:0]</th><th>Transfer Size</th></tr>
<tr><td>00</td><td>00</td><td>Byte (LE, byte 0; BE, byte 3)</td></tr>
<tr><td>01</td><td>00</td><td>Byte (LE, byte 1; BE, byte 2)</td></tr>
<tr><td>10</td><td>00</td><td>Byte (LE, byte 2; BE, byte 1)</td></tr>
<tr><td>11</td><td>00</td><td>Byte (LE, byte 3; BE, byte 0)</td></tr>
<tr><td>00</td><td>01</td><td>Halfword (LE, bytes 1:0; BE, bytes 3:2)</td></tr>
<tr><td>10</td><td>01</td><td>Halfword (LE, bytes 3:2; BE, bytes 1:0)</td></tr>
<tr><td>00</td><td>10</td><td>Word (LE, BE; bytes 3, 2, 1, 0)</td></tr>
<tr><td>00</td><td>11</td><td>Triple (LE, bytes 2, 1, 0; BE, bytes 3, 2,1)</td></tr>
<tr><td>01</td><td>11</td><td>Triple (LE, bytes 3, 2, 1; BE, bytes 2, 1, 0)</td></tr>
<tr><td colspan="2">All others</td><td>Reserved</td></tr>
</table>
Note: LE=little endian, BE=big endian, the byte# refers to the byte number in a 32-bit register, where byte 3 = bits 31:24; byte 2 = bits 23:16; byte 1 = bits 15:8; byte 0=bits 7:0, independently of the endianess.</td><td>R</td><td>Undefined</td></tr>
<tr><td>Res</td><td>28:24</td><td>Reserved.</td><td>R</td><td>0</td></tr>
<tr><td>VPED</td><td>23</td><td>This bit is implemented and must be tied to zero.</td><td>R</td><td>1</td></tr>
</table>

Table 14.29 EJTAG Control Register Descriptions *(continued)*

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>Doze</td><td>22</td><td>Doze state
The Doze bit indicates any type of low-power mode. The value is sampled in the Capture-DR state of the TAP controller:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>CPU not in low power mode</td></tr>
<tr><td>1</td><td>CPU is in low power mode</td></tr>
</table>
Doze includes the Reduced Power (RP) and WAIT power-reduction modes.</td><td>R</td><td>0</td></tr>
<tr><td>Halt</td><td>21</td><td>Halt state
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:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Internal system clock is running</td></tr>
<tr><td>1</td><td>Internal system clock is stopped</td></tr>
</table></td><td>R</td><td>0</td></tr>
<tr><td>PerRst</td><td>20</td><td>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 takes effect in the CPU clock domain and in peripherals.
When the bit is written to 0, it must also be read as 0 before it is guaranteed that the indication is also cleared in the CPU clock domain.
This bit controls the *EJ_PerRst* signal on the core.</td><td>R/W</td><td>0</td></tr>
<tr><td>PRnW</td><td>19</td><td>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:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Read transaction</td></tr>
<tr><td>1</td><td>Write transaction</td></tr>
</table></td><td>R</td><td>Undefined</td></tr>
</table>

Table 14.29 EJTAG Control Register Descriptions *(continued)*

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>PrAcc</td><td>18</td><td>Processor Access (PA)
Read value of this bit indicates if a Processor Access (PA) to the EJTAG memory is pending:<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No pending processor access</td></tr><tr><td>1</td><td>Pending processor access</td></tr></table>The probe’s software must clear this bit to 0 to indicate the end of the PA. A 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 a Processor Access occurring after the reset is finished because of an indication of a Processor Access that occurred before the reset.
The FASTDATA access can clear this bit.</td><td>R/W0</td><td>0</td></tr>
<tr><td>Res</td><td>17</td><td>Reserved</td><td>R</td><td>0</td></tr>
<tr><td>PrRst</td><td>16</td><td>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.</td><td>R/W</td><td>0</td></tr>
</table>

Table 14.29 EJTAG Control Register Descriptions *(continued)*

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>ProbEn</td><td>15</td><td>Probe Enable
This bit indicates to the CPU if the EJTAG memory is handled by the probe so processor accesses are answered:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Probe does not handle EJTAG memory transactions</td></tr>
<tr><td>1</td><td>Probe does handle EJTAG memory transactions</td></tr>
</table>
It is an error by the software controlling the probe if it sets the ProbTrap bit to 1, but resets the *ProbEn* to 0. The operation of the processor is UNDEFINED in this case.
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</td><td>R/W</td><td>0 or 1
from
EJTAGBOOT</td></tr>
<tr><td>ProbTrap</td><td>14</td><td>Probe Trap
This bit controls the location of the debug exception vector:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>In normal memory. Vector is located as described in Section 14.14.1.2 “DebugVectorAddr Register”</td></tr>
<tr><td>1</td><td>In EJTAG memory at 0xFF20.0200 in dmseg</td></tr>
</table>
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 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:</td><td>R/W</td><td>0 or 1
from
EJTAGBOOT</td></tr>
<tr><td>Res</td><td>13</td><td>reserved</td><td>R</td><td>0</td></tr>
</table>

Table 14.29 EJTAG Control Register Descriptions *(continued)*

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bit(s)</th></tr>
<tr><td>EjtagBrk</td><td>12</td><td>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:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>No EJTAGBOOT indication given</td></tr>
<tr><td>1</td><td>EJTAGBOOT indication given</td></tr>
</table></td><td>R/W</td><td>0 or 1
from
EJTAGBOOT</td></tr>
<tr><td>Res</td><td>11:4</td><td>Reserved</td><td>R</td><td>0</td></tr>
<tr><td>DM</td><td>3</td><td>Debug Mode
This bit indicates the debug or non-debug mode:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Processor is in non-debug mode</td></tr>
<tr><td>1</td><td>Processor is in debug mode</td></tr>
</table>
The bit is sampled in the <i>Capture-DR</i> state of the TAP controller.</td><td>R</td><td>0</td></tr>
<tr><td>Res</td><td>2:0</td><td>Reserved</td><td>R</td><td>0</td></tr>
</table>

14.14.5 Processor Access Registers

14.14.5.1 Processor Access Address Register

The Address 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.

14.14.5.2 Processor Access Data Register

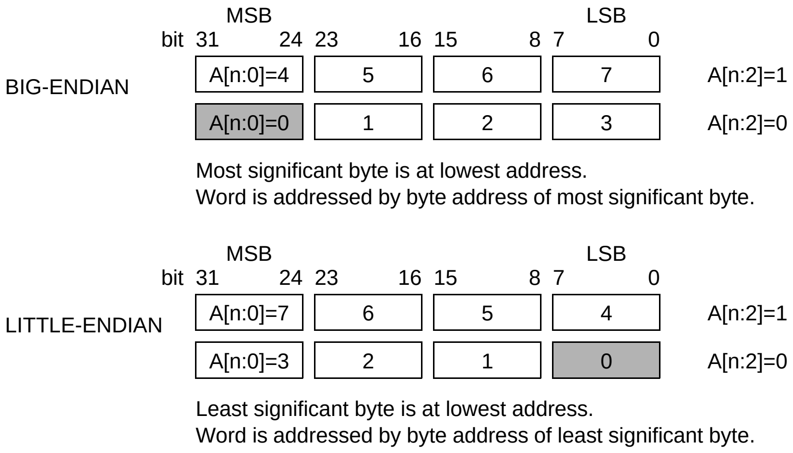
The Data register is used to provide data value to and from a processor access. The length of the Data register is 32 bits, and this register is selected by shifting in the DATA instruction.

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. The register will be updated with a new value when a processor access write is pending.

The Data register is 32 bits wide. Data alignment is not used for this register, so the value in the Data register matches data on the internal bus. The unused bytes for a processor access write are undefined, and for a Data register read, 0 (zero) must be shifted in for the unused bytes.

The organization of bytes in the Data register depends on the endianess of the core, as shown in Figure 14.24. The endian mode for debug/kernel mode is determined by the state of the *SI_Endian* input at power-up.

Figure 14.24 Endian Formats for the Data 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 Data register is determined by the Psz field in the *ECR*.

14.14.6 Fastdata Registers

14.14.6.1 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 14.25 Fastdata Register Format

31 … 1	0
R	SPrAcc

Table 14.30 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

14.14.7 FDC TAP Register

The FDC TAP instruction performs a 38 bit bidirectional transfer of the FDC TAP register. The register format is shown in Figure 14.26 and the fields are described in Figure 14.31

Figure 14.26 FDC TAP Register Format

	37	36	35 32	31 0
In	Probe Data Accept	Data In Valid	ChannelID	Data
Out	Receive Buffer Full	Data Out Valid		

Table 14.31 FDC TAP Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
Probe Data Accept	37	Indicates to core that the probe is accepting the data that was scanned out.	W	Undefined
Data In Valid	36	Indicates to core that the probe is sending new data to the receive FIFO.	W	Undefined
Receive Buffer Full	37	Indicates to probe that the receive buffer is full and the core will not accept the data being scanned in. Analagous to ProbeDataAccept, but opposite polarity	R	0
Data Out Valid	36	Indicates to probe that the core is sending new data from the transmit FIFO	R	0
ChannelID	35:32	Channel number associated with the data being scanned in or out. This field can be used to indicate the type of data that is being sent and allow independent communication channels Scanning in a value with ChannelID=0xd and Data In Valid = 0 will generate a receive interrupt. This can be used when the probe has completed sending data to the core.	R/W	Undefined
Data	31:0	Data value being scanned in or out	R/W	Undefined

14.14.8 Fast Debug Channel Registers

This section describes the Fast Debug Channel registers. CPU access to FDC is via loads and stores to the FDC device in the Common Device Memory Map (CDMM) region. These registers provide access control, configuration and status information, as well as access to the transmit and receive FIFOs. The registers and their respective offsets are shown in Table 14.32

Table 14.32 FDC Register Mapping

Offset in CDMM device block	Register Mnemonic	Register Name and Description
0x0	FDACSR	FDC Access Control and Status Register
0x8	FDCFG	FDC Configuration Register
0x10	FDSTAT	FDC Status Register
0x18	FDRX	FDC Receive Register
0x20 + 0x8* n	FDTXn	FDC Transmit Register n (0 ≤ n ≤ 15)

14.14.8.1 FDC Access Control and Status (FDACSR) Register (Offset 0x0)

This is the general CDMM Access Control and Status register which defines the device type and size and controls user and supervisor access to the remaining FDC registers. The Access Control and Status register itself is only accessible in kernel mode. Figure 14.27 has the format of an Access Control and Status register (shown as a 64-bit register), and Table 14.33 describes the register fields.

Figure 14.27 FDC Access Control and Status Register

63 32	31 24	23 22	21 16	15 12	11 4	3	2	1	0
0	DevID	0	DevSize	DevRev	0	Uw	Ur	Sw	Sr

Table 14.33 FDC Access Control and Status Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
DevType	31:24	This field specifies the type of device.	R	0xfd
DevSize	21:16	This field specifies the number of extra 64-byte blocks allocated to this device. The value 0x2 indicates that this device uses 2 extra, or 3 total blocks.	R	0x2
DevRev	15:12	This field specifies the revision number of the device. The value 0x0 indicates that this is the initial version of FDC	R	0x0
Uw	3	This bit indicates if user-mode write access to this device is enabled. A value of 1 indicates that access is enabled. A value of 0 indicates that access is disabled. An attempt to write to the device while in user mode with access disabled is ignored.	R/W	0
Ur	2	This bit indicates if user-mode read access to this device is enabled. A value of 1 indicates that access is enabled. A value of 0 indicates that access is disabled. An attempt to read from the device while in user mode with access disabled will return 0 and not change any state.	R/W	0

Table 14.33 FDC Access Control and Status Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
Sw	1	This bit indicates if supervisor-mode write access to this device is enabled. A value of 1 indicates that access is enabled. A value of 0 indicates that access is disabled. An attempt to write to the device while in supervisor mode with access disabled is ignored.	R/W	0
Sr	0	This bit indicates if supervisor-mode read access to this device is enabled. A value of 1 indicates that access is enabled. A value of 0 indicates that access is disabled. An attempt to read from the device while in supervisor mode with access disabled will return 0 and not change any state..	R/W	0
0	11:4	Reserved for future use. Ignored on write; returns zero on read.	R	0

14.14.8.2 FDC Configuration (FDCFG) Register (Offset 0x8)

The FDC configuration register holds information about the current configuration of the Fast Debug Channel mechanism. Figure 14.28 has the format of the FDC Configuration register, and Table 14.34 describes the register fields.

Figure 14.28 FDC Configuration Register

31 20	19 18	17 16	15 8	7 0
0	Tx_IntThresh	Rx_IntThresh	TxFIFOSize	RxFIFOSize

Table 14.34 FDC Configuration Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>0</td><td>31:20</td><td>Reserved for future use. Read as zeros, must be written as zeros.</td><td>R</td><td>0</td></tr>
<tr><td>TxIntThresh</td><td>19:18</td><td>Controls whether transmit interrupts are enabled and the state of the TxFIFO needed to generate an interrupt.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Transmit Interrupt Disabled</td></tr>
<tr><td>1</td><td>Empty</td></tr>
<tr><td>2</td><td>Not Full</td></tr>
<tr><td>3</td><td>Almost Empty - zero or one entry in use*(see 14.15.2 for specifics)</td></tr>
</table></td><td>R/W</td><td>0</td></tr>
</table>

Table 14.34 FDC Configuration Register Field Descriptions*(continued)*

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>RxIntThresh</td><td>17:16</td><td>Controls whether receive interrupts are enabled and the state of the RxFIFO needed to generate an interrupt.
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>0</td><td>Receive Interrupt Disabled</td></tr>
<tr><td>1</td><td>Full</td></tr>
<tr><td>2</td><td>Not empty</td></tr>
<tr><td>3</td><td>Almost Full - zero or one entry free</td></tr>
</table></td><td>R/W</td><td>0</td></tr>
<tr><td>TxFIFOSize</td><td>15:8</td><td>This field holds the total number of entries in the transmit FIFO.</td><td>R</td><td>Preset</td></tr>
<tr><td>RxFIFOSize</td><td>7:0</td><td>This field holds the total number of entries in the receive FIFO.</td><td>R</td><td>Preset</td></tr>
</table>

14.14.8.3 FDC Status (FDSTAT) Register (Offset 0x10)

The FDC Status register holds up to date state information for the FDC mechanism. Figure 14.29 has the format of the FDC Status register, and Table 14.35 describes the register fields.

Figure 14.29 FDC Status Register

31 24	23 16	15 8	7 4	3	2	1	0
Tx_Count	Rx_Count	0	RxChan	RxE	RxF	TxE	TxF

Table 14.35 FDC Status Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
Tx_Count	31:24	This optional field is not implemented and will read as 0	R	0
Rx_Count	23:16	This optional field is not implemented and will read as 0	R	0
0	15:8	Reserved for future use. Must be written as zeros and read as zeros.	R	0
RxChan	7:4	This field indicates the channel number used by the top item in the receive FIFO. This field is only valid if RxE=0.	R	Undefined
RxE	3	If RxE is set, the receive FIFO is empty. If RxE is not set, the FIFO is not empty.	R	1
RxF	2	If RxF is set, the receive FIFO is full. If RxF is not set, the FIFO is not full.	R	0
TxE	1	If TxE is set, the transmit FIFO is empty. If TxE is not set, the FIFO is not empty.	R	1
TxF	0	If TxF is set, the transmit FIFO is full. If TxF is not set, the FIFO is not full.	R	0

14.14.8.4 FDC Receive (FDRX) Register (Offset 0x18)

This register exposes the top entry in the receive FIFO. A read from this register returns the top item in the FIFO and removes it from the FIFO itself. The result of a write to this register is **UNDEFINED**. The result of a read when the FIFO is empty is also **UNDEFINED** so software must check the *$FDSTAT_{RxE}$* flag prior to reading. Figure 14.30 has the format of the FDC Receive register, and Table 14.36 describes the register fields.

Figure 14.30 FDC Receive Register

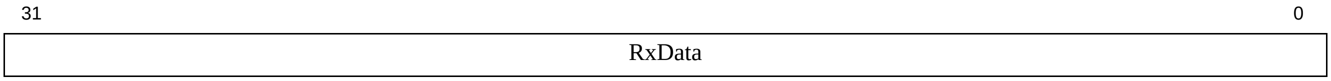


Table 14.36 FDC Receive Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
RxData	31:0	This register holds the top entry in the receive FIFO	R	Undefined

14.14.8.5 FDC Transmit n (FDTXn) Registers (Offset 0x20 + 0x8*n)

These sixteen registers all access the bottom entry in the transmit FIFO. The different addresses are used to generate a 4b channel identifier that is attached to the data value. This allows software to track different event types without needing to reserve a portion of the 32b data as a tag. A write to one of these registers results in a write to the transmit FIFO of the data value and channel ID corresponding to the register being written. Reads from these registers are **UNDEFINED**. Attempting to write to the transmit FIFO if it is full has **UNDEFINED** results. Hence, the software running on the core must check the *$FDSTAT_{TxF}$* flag to ensure that there is space for the write. Figure 14.31 has the format of the FDC Transmit register, and Table 14.37 describes the register fields.

Figure 14.31 FDC Transmit Register

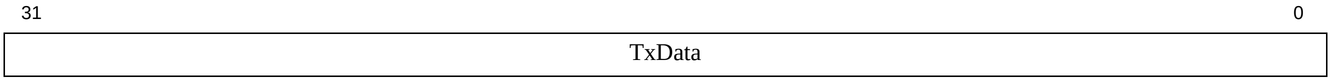


Table 14.37 FDC Transmit Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
TxData	31:0	This register holds the bottom entry in the transmit FIFO	W, Undefined value on read	Undefined

Table 14.38 FDTXn Address Decode

Addr	Chan		Addr	Chan		Addr	Chan		Addr	Chan
0x20	0x0		0x40	0x4		0x60	0x8		0x80	0xc
0x28	0x1		0x48	0x5		0x68	0x9		0x88	0xd
0x30	0x2		0x50	0x6		0x70	0xa		0x90	0xe

Table 14.38 FDTXn Address Decode

Addr	Chan		Addr	Chan		Addr	Chan		Addr	Chan
0x38	0x3		0x58	0x7		0x78	0xb		0x98	0xf

14.14.9 PDtrace™ Registers (Software Control)

The CP0 registers associated with PDtrace are listed in Table 14.39 and described in Chapter 2, “CP0 Registers” on page 73.

Table 14.39 A List of Coprocessor 0 Trace Registers

Register Number	Sel	Register Name
23	1	*TraceControl*
23	2	*TraceControl2*
24	2	*TraceControl3*
23	3	*UserTraceData1*
24	3	*UserTraceData2*

14.14.10 Trace Control Block (TCB) Registers (Hardware Control)

The TCB registers used to control its operation are listed in Table 14.40 and Table 14.41. These registers are accessed via the EJTAG TAP interface, or by software through mapping to dsseg memory space.

Table 14.40 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 14.14.10.1 “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 14.41. See Section 14.14.10.2 “TCBCONTROLB Register”.	Yes
0x12	TCBDATA	This is used to access registers specified by the *REG* field in the *TCBCONTROLB* register. See Section 14.14.10.3 “TCBDATA Register”.	Yes
0x13	TCBCONTROLC	Control Register in the TCB used to control and hold tracing information. See Section 14.14.10.4 “TCBCONTROLC Register”.	Yes
0x16	TCBCONTROLE	Control Register in the TCB used to control tracing for the performance counter tracing feature. See Section 14.14.11.1 “TCBCONTROLE Register”.	Yes

Table 14.41 Registers Selected by TCBCONTROLB$_{REG}$

TCBCONTROLB$_{REG}$ field	Name	Reference	Implemented
0	TCBCONFIG	Section 14.14.11.2 “TCBCONFIG Register (Reg 0)”	Yes
4 - 7		Values are undefined.	No
16-23	TCBTRIGx	Section 14.14.11.3 “TCBTRIGx Register (Reg 16-23)”	Only the number indicated by *TCBCONFIG*$_{TRIG}$ are implemented.

14.14.10.1 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). This register is also mapped to offset 0x3000 in drseg. .

The format of the *TCBCONTROLA* register is shown below, and the fields are described in Table 14.42.

Figure 14.32 TCBCONTROLA Register Format

31 30	29 26	25 24	23	22 20	19	18	17	16	15	14	13	12 5	4	3	2	1	0
SyPExt	0	VModes	ADW	SyP	TB	IO	D	E	S	K	U	ASID	G	TFCR	TLSM	TIM	On

Table 14.42 TCBCONTROLA Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
SyPExt	31:30	These two bits used to be Implementation specific until PDtrace spec revision 06.00 when it reverts to architecturally defined bits to extend the SyP (sync period) field for implementations that need higher numbers 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 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. Since the value of 11010 represents the value of 31 (with +5), all values greater than 11010 are UNPREDICTABLE. Note that with these new bits, a sync period range of 2^{5} to 2^{31} cycles can now be obtained.	R/W	0
0	29:26	Reserved. Must be written as zero; returns zero on read.	R	0

Table 14.42 TCBCONTROLA Register Field Descriptions*(continued)*

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>VModes</td><td>25:24</td><td>This field specifies the type of tracing that is supported by the processor, as follows:
<table>
<tr><th>Encoding</th><th>Meaning</th></tr>
<tr><td>00</td><td>PC tracing only</td></tr>
<tr><td>01</td><td>PC and Load and store address tracing only</td></tr>
<tr><td>10</td><td>PC, load and store address, and load and store data.</td></tr>
<tr><td>11</td><td>Reserved</td></tr>
</table>
This field is preset to the value of *PDO_ValidModes*.</td><td>R</td><td>10</td></tr>
<tr><td>ADW</td><td>23</td><td>*PDO_AD* bus width.
0: The *PDO_AD* bus is 16 bits wide.
1: The *PDO_AD* bus is 32 bits wide.</td><td>R</td><td>1</td></tr>
<tr><td>SyP</td><td>22:20</td><td>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.
<table>
<tr><th>SyP</th><th>Sync Period</th></tr>
<tr><td>000</td><td>2^{5}</td></tr>
<tr><td>001</td><td>2^{6}</td></tr>
<tr><td>010</td><td>2^{7}</td></tr>
<tr><td>011</td><td>2^{8}</td></tr>
<tr><td>100</td><td>2^{9}</td></tr>
<tr><td>101</td><td>2^{10}</td></tr>
<tr><td>110</td><td>2^{11}</td></tr>
<tr><td>111</td><td>2^{12}</td></tr>
</table>
This field defines the value on the *PDI_SyncPeriod* signal.</td><td>R/W</td><td>000</td></tr>
<tr><td>TB</td><td>19</td><td>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.
This field defines the value on the *PDI_TraceAllBranch* signal.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>IO</td><td>18</td><td>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.
This field defines the value on the *PDI_InhibitOverflow* signal.</td><td>R/W</td><td>Undefined</td></tr>
<tr><td>D</td><td>17</td><td>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.
This field defines the value on the *PDI_DM* signal.</td><td>R/W</td><td>Undefined</td></tr>
</table>

Table 14.42 TCBCONTROLA Register Field Descriptions*(continued)*

Fields		Description	Read / Write	Reset State
Name	Bits			
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. This field defines the value on the *PDI_E* signal.	R/W	Undefined
S	15	When set, this enables tracing when the core is in Supervisor mode as defined in the MIPS32 or MIPS64 architecture specification. This is provided the On bit (bit 0) is also set, and either the G bit is set, or the current process ASID matches the *ASID* field in this register. This field defines the value on the *PDI_S* signal.	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. This field defines the value on the *PDI_K* signal.	R/W	Undefined
U	13	When set, this enables tracing when the core is in User mode as defined in the MIPS32 or MIPS64 architecture specification. This is provided the *On* bit (bit 0) is also set, and either the *G* bit is set, or the current process ASID matches the *ASID* field in this register. This field defines the value on the *PDI_U* signal.	R/W	Undefined
ASID	12:5	The ASID field to match when the *G* bit is zero. When the *G* bit is one, this field is ignored. This field defines the value on the *PDI_ASID* signal.	R/W	Undefined
G	4	When set, this implies that tracing is to be enabled for all processes, provided that other enabling functions (like U, S, etc.,) are also true. This field defines the value on the *PDI_G* signal.	R/W	Undefined
TFCR	3	When set, this indicates to the PDtrace interface that complete information about instruction if it can be a function call or return should be traced, that is signal *PDI_TraceFuncCR* is asserted as long as this value is set to 1. It also indicates to the TCB that the optional Fcr bit must be traced in the appropriate trace formats	R/W	Undefined
TLSM	2	When set, this indicates to the PDtrace interface that complete information about Load and Store data cache miss should be traced, that is signal *PDI_TraceLSMiss* is asserted as long as this value is set to 1. It also indicates to the TCB that the optional LSm bit must be traced in the appropriate trace formats.	R/W	Undefined
TIM	1	When set, this indicates to the PDtrace interface that complete information about instruction cache miss should be traced, that is signal *PDI_TraceIMiss* is asserted as long as this value is set to 1. It also indicates to the TCB that the optional Im bit must be traced in the appropriate trace formats.	R/W	Undefined

Table 14.42 TCBCONTROLA Register Field Descriptions*(continued)*

Fields		Description	Read / Write	Reset State
Name	Bits			
On	0	This is the global trace enable switch to the core. When zero, tracing from the core is always disabled, unless enabled by core internal software override of the *PDI_** input pins. When set to one, tracing is enabled whenever the other enabling functions are also true. This field defines the value on the *PDI_TraceOn* signal.	R/W	0

14.14.10.2 TCBCONTROLB Register

The TCB includes a second control register, *TCBCONTROLB* (0x11). This register generally controls what to do with the trace information received. This register is also mapped to offset 0x3008 in drseg.

The format of the *TCBCONTROLB* register is shown below, and the fields are described in Table 14.43.

Figure 14.33 TCBCONTROLB Register Format

31	30 28	27 26	25 21	20	19	18	17 12	11	10 7	6 3	2	1	0
WE	0	TWSrcWidth	REG	WR	0	TRPAD	0	TLSIF	0	0	CA	0	EN

Table 14.43 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. 00 - zero source field width 01 - two bit source field width 10 - four bit source field width 11 - reserved for future use This field can only be 10 for the proAptiv Multiprocessing System core.	R	10
REG	25:21	Register select: This field select the registers accessible through the *TCBDATA* register. Legal values are shown in Table 14.41.	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. 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
0	17:12	Reserved. Must be written as zero; returns zero on read.	R	0

Table 14.43 TCBCONTROLB Register Field Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Name	Bits			
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
0	10:7	Reserved. Must be written as zero; returns zero on read.	R	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	Preset
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	This bit is always set to 1, indicating that the trace is sent to off-chip memory using *TR_DATA* pins.	R	1
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 Words are generated and sent to the trace funnel. When set to 0, trace information is ignored. A potential TF6-stop (from a stop trigger) is generated as the last information, the TCB pipe-line is flushed, and trace output is stopped.	R/W	0

14.14.10.3 TCBDATA Register

The *TCBDATA* register (0x12) is used to access the registers defined by the *TCBCONTROLB*$_{REG}$ field; see Table 14.41. Regardless of which register or data entry is accessed through *TCBDATA*, the register is only written if the *TCBCONTROLB*$_{WR}$ bit is set. For read-only registers, *TCBCONTROLB*$_{WR}$ is a don't care.

The format of the *TCBDATA* register is shown below, and the field is described in Table 14.44. The width of *TCBDATA* is 64 bits when on-chip trace words (TWs) are accessed (*TCBTW* access).

Figure 14.34 TCBDATA Register Format

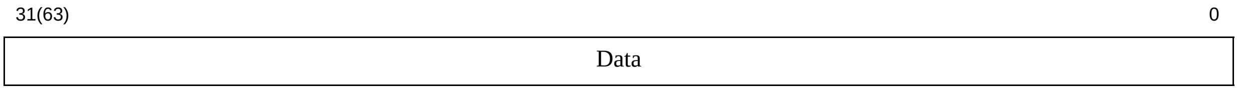


Table 14.44 TCBDATA Register Field Descriptions

Fields		Description	Read/Write	Reset State
Names	**Bits**			
Data	31:0 63:0	Register fields or data as defined by the $\mathit{TCBCONTROLB}_{REG}$ field	Only writable if $\mathit{TCBCONTROLB}_{WR}$ is set	0

14.14.10.4 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 to this register.

The *TCBCONTROLC* register is written by the EJTAG TAP controller instruction, TCBCONTROLC (0x13). This register is also mapped to offset 0x3010 in drseg.

The format of the *TCBCONTROLC* register is shown below, and the fields are described in Table 14.45.

Figure 14.35 TCBCONTROLC Register Format

31 30	29 28	27 23	22 0
Res	NumDO	Mode	R

31 30	29 28	27 23	22	21 15	14	13	12 9	8 5	4 2	1	0
Res	NumDO	Mode	CPUValid	Res	CPUId	TCValid	Res	TCNum	TCbits	MTraceType	MTraceTC

Table 14.45 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: 10 - Six bits	R	10

Table 14.45 TCBCONTROLC Register Field Descriptions*(continued)*

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>Mode</td><td>27:23</td><td>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.
<table><tr><th>Bit # Set</th><th>Trace The Following</th></tr><tr><td>0</td><td>PC</td></tr><tr><td>1</td><td>Load address</td></tr><tr><td>2</td><td>Store address</td></tr><tr><td>3</td><td>Load data</td></tr><tr><td>4</td><td>Store data</td></tr></table>
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.
On the proAptiv Multiprocessing System core PC tracing is always enabled, regardless of the value on bit 23.
This field defines the value on the *PDI_TraceMode* signal.</td><td>R/W</td><td>0</td></tr>
<tr><td>R</td><td>22:0</td><td>Reserved for future use. Must be written as zero; returns zero on read.</td><td>R/W</td><td>0</td></tr>
</table>

14.14.11 TCBCONTROLD Register

The TCB includes a control register, TCBCONTROLD, whose values are used to enable tracing of the Coherence Manager. External software (i.e., debugger) can therefore manipulate the trace output by writing to this register. Each of the cores in the proAptiv has this register, and the *Core_CM_En* field is considered from each of the cores.

The *TCBCONTROLD* register is written by an EJTAG TAP controller instruction, *TCBCONTROLD* (0x14). This register is also mapped to offset 0x3018 in drseg. The format of the *TCBCONTROLD* register is shown below, and the fields are described in Table 14.46.

Figure 14.36 TCBCONTROLD Register Format

31 2	1	0
0	Gore_CM_En	0

Table 14.46 TCBCONTROLD Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:2	Reserved. Must be written as zero; returns zero on read.	R	0
Core_CM_En	1	Core_CM_Enable: The CM looks at this bit coming from each of the cores. Allows cores other than the master to enable tracing if other conditions are met.	R/W	0
0	0	Reserved. Must be written as zero; returns zero on read.	R	0

14.14.11.1 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).This register is also mapped to offset 0x3020 in drseg.

The format of the *TCBCONTROLE* register is shown below, and the fields are described in Table 14.47.

Figure 14.37 TCBCONTROLE Register Format

31 9	8	7 6	5	4	3	2	1	0
0	TdIDLE	0	PecOvf	PeCFCR	PeCBP	PeCSync	PeCE	PeC

Table 14.47 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

Table 14.47 TCBCONTROLE Register Field Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Name	Bits			
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
PeCE	1	Performance counter tracing enable. If performance counter hardware is present, this field is read/write. If not present, this field is read-only. 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.	Config Option	0
PeC	0	Specifies whether or not Performance Control Tracing is implemented. This is an optional feature that may be omitted by implementation choice.	R	Preset

14.14.11.2 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 fields are described in Table 14.48.

Figure 14.38 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	0	OfT	REV

Table 14.48 Clock Ratio Encoding of the CR Field

CR/CRMin/CRMax	Clock Ratio
000	8:1 (Trace clock is eight times that of CPU clock)
001	4:1 (Trace clock is four times that of CPU clock)
010	2:1 (Trace clock is two times that of CPU clock)
011	1:1 (Trace clock is same as CPU clock)
100	1:2 (Trace clock is one half of CPU clock)
101	1:4 (Trace clock is one fourth of CPU clock)
110	1:6 (Trace clock is one sixth of CPU clock)
111	1:8 (Trace clock is one eighth of CPU clock)

14.14.11.3 TCBTRIGx Register (Reg 16-23)

Up to eight Trigger Control registers are possible. Each register is named *TCBTRIGx*, where *x* is a single digit number from 0 to 7 (*TCBTRIG0* is Reg 16). The actual number of trigger registers implemented is defined in the $TCBCONFIG_{TRIG}$ field. An unimplemented register will read all zeros and writes are ignored.

Each Trigger Control register controls when an associated trigger is fired, and the action to be taken when the trigger occurs. Please also read Section 14.16 “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 14.49.

Figure 14.39 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 14.49 TCBTRIGx Register Field Descriptions

Fields		Description	Read / Write	Reset State
Names	Bits			
TCBinfo	31:24	This field is 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

Table 14.49 TCBTRIGx Register Field Descriptions *(continued)*

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Names</th><th>Bits</th></tr>
<tr><td>PDTri</td><td>4</td><td>When set, this Trigger will fire when a rising edge on <i>TC_ProbeTrigIn</i> 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 <i>TCBTRIGx</i> register is written.</td><td>R/W</td><td>0</td></tr>
<tr><td>Type</td><td>3:2</td><td>Trigger Type: The Type indicates the action to take when this trigger fires. The table below show the Type values and the Trigger action.
<table>
<tr><th>Type</th><th>Trigger action</th></tr>
<tr><td>00</td><td><b>Trigger Start:</b> Trigger start-point of trace.</td></tr>
<tr><td>01</td><td><b>Trigger End:</b> Trigger end-point of trace.</td></tr>
<tr><td>10</td><td><b>Trigger About:</b> Trigger center-point of trace.</td></tr>
<tr><td>11</td><td><b>Trigger Info:</b> No action trigger, only for trace info.</td></tr>
</table>
The actual action is to set or clear the $\mathit{TCBCONTROLB}_{EN}$ bit. A Start trigger will set $\mathit{TCBCONTROLB}_{EN}$, a End trigger will clear $\mathit{TCBCONTROLB}_{EN}$. The About trigger will clear $\mathit{TCBCONTROLB}_{EN}$ half way through the trace memory, from the trigger. The size determined by the $\mathit{TCBCONFIG}_{SZ}$ field for on-chip memory. Or from the $\mathit{TCBCONTROLA}_{SyP}$ 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 $\mathit{TCBCONTROLB}_{TM}$ field is implemented it must be set to Trace-To mode (00), for the <i>Type</i> 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 <i>TCBTRIGx</i> register is written.</td><td>R/W</td><td>0</td></tr>
<tr><td>FO</td><td>1</td><td>Fire Once. When set, this trigger will not re-fire until the <i>TR</i> bit is de-asserted. When de-asserted this trigger will fire each time one of the trigger sources indicates trigger.</td><td>R/W</td><td>0</td></tr>
</table>

Table 14.49 TCBTRIGx Register Field Descriptions *(continued)*

Fields		Description	Read / Write	Reset State
Names	Bits			
TR	0	Trigger happened. When set, this trigger fired since the *TR* bit was last written 0. 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

14.14.12 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.

14.15 Fast Debug Channel

The Fast Debug Channel (FDC) mechanism provides an efficient means to transfer data between the core and an external device using the EJTAG TAP pins. The FDC was created to allow for faster communication between the core and the probe. In previous generation MIPS processors, whenever the core wanted to communicate with the probe, the core would be halted and data send to the probe because the probe had no way to read the core. The FDC provides a mechanism using FIFO’s, whereby the probe can read the core without requiring that the core be halted. These FIFO’s provide a cross boundary between the core and the EJTAG regions of the proAptiv Multiprocessing System.

In the FDC, when the probe wishes to read and FDC register, the core gets an interrupt from the probe requesting this information. The core then places the requested information into the FIFO and continues operation. The core places information in the top of the FIFO, and the probe reads information from the bottom of the FIFO. The data contains information such as transmit versus receive, status of the operation, etc.

The external device would typically be an EJTAG probe and that is the term used here, but it could be something else. FDC utilizes two First In First Out (FIFO) structures to buffer data between thecore and probe. The probe uses the FDC TAP instruction to access these FIFOs, while the core itself accesses them using memory accesses. To transfer data out of the core, the core writes one or more pieces of data to the transmit FIFO. At this time, the core can resume doing other work. An external probe would examine the status of the transmit FIFO periodically. If there is data to be read, the probe starts to receive data from the FIFO, one entry at a time. When all data from the FIFO has been drained, the probe goes back to waiting for more data. The core can either choose to be informed of the empty transmit FIFO via an interrupt, or it can choose to periodically check the status. Receiving data works in a similar manner - the probe writes to the receive FIFO. At that time, the core is either interrupted, or finds out via polling a status bit. The core can then do load accesses to the receive FIFO and receive data being sent to it by the probe. The TAP transfer is bidirectional - a single shift can be pulling transmit data and putting receive data at the same time.

The primary advantage of FDC over normal processor accesses or fastdata accesses is that it does not require the core to be blocked when the probe is reading or writing to the data transfer FIFOs. This significantly reduces the core overhead and makes the data transfer far less intrusive to the code executing on the core.

The FDC memory mapped registers are located in the common device memory map (CDMM) region. FDC has a device ID of 0xFD.

14.15.1 Common Device Memory Map

Software on the core accesses FDC through memory mapped registers. These memory mapped registers are located within the Common Device Memory Map (CDMM). The CDMM is a region of physical address space that is reserved for mapping IO device configuration registers within a MIPS processor. The base address and enabling of this region is controlled by the CDMMBase CP0 register.

14.15.2 Fast Debug Channel Interrupt

The FDC block can generate an interrupt to inform software of incoming data being available or space being available in the outgoing FIFO. This interrupt is handled similarly to the timer or performance counter interrupts. The $Cause_{FDCI}$ bit indicates that the interrupt is pending. The interrupt is also sent to the core outputs *SI_FDCI[_1]* where it is combined with one of the *SI_Int* pins. For non-EIC mode, the *SI_IPFDCI* input indicates which interrupt pin is has been combined with and this information is reflected in the $IntCtl_{IPFDCI}$ field. Note that this interrupt is a regular interrupt and not a debug interrupt.

The FDC Configuration Register (see Section 14.14.8.2 "FDC Configuration (FDCFG) Register (Offset 0x8)") includes fields for enabling and setting the threshold for generating each interrupt. Receive and transmit interrupt thresholds are specified independently, but transmit/receive interrrupts are ORed together to form a single interrupt per core.

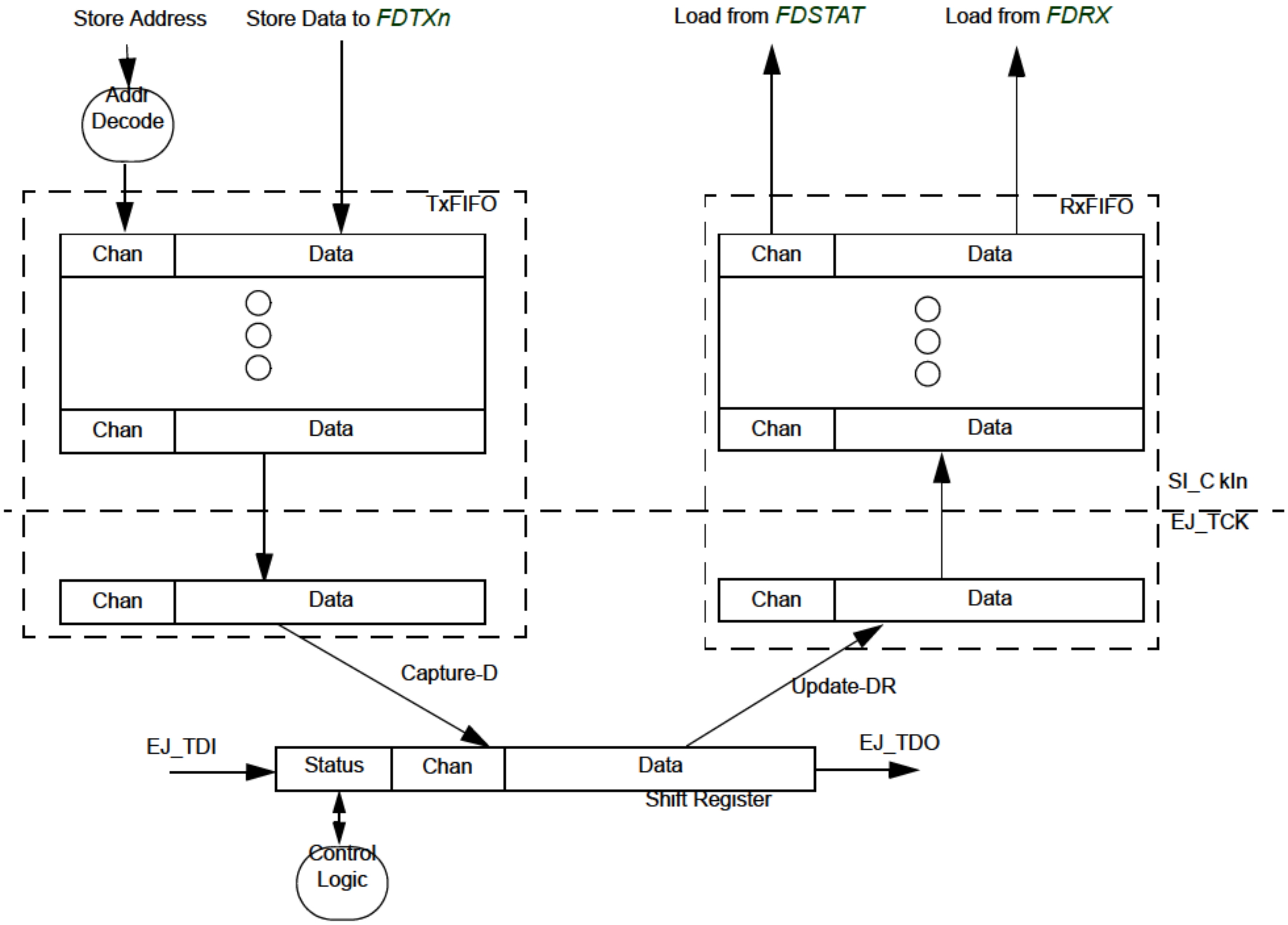
The following interrupt thresholds are supported:

- Interrupts Disabled: No interrupt will be generated and software must poll the status registers to determine if incoming data is available or if there is space for outgoing data.
- Minimum core Overhead: This setting minimizes the core overhead by not generating an interrupt until the receive FIFO (RxFIFO) is completely full or the transmit FIFO (TxFIFO) is completely empty.
- Minimum latency: To have the core take data as soon as it is available, the receive interrupt can be fired whenever the RxFIFO is not empty. There is a complimentary TxFIFO not full setting although that may not be quite as useful.
- Maximum bandwidth: When configured for minimum core overhead, bandwidth between the probe and core can be wasted if the core does not service the interrupt before the next transfer occurs. To reduce the chances of this happening, the interrupt threshold can be set to almost full or almost empty to generate an interrupt earlier. This setting causes receive interrupts to be generated when there are 0 or 1 unused RxFIFO entries. Transmit interrupts are generated when there are 0 or 1 used TxFIFO entries (see note in following section about this condition)

14.15.3 Core FDC Buffers

Figure 14.40 shows the general organization of the transmit and receive buffers on the proAptiv Multiprocessing System core.

Figure 14.40 Fast Debug Channel Buffer Organization



One particular thing to note is the asynchronous crossings between the EJ_TCK and SI_ClkIn clock domains. This crossing is handled with a handshaked interface that safely transfers data between the domains. Two data registers are included in this interface, one in the source domain and one in the destination domain. The control logic actively manages these registers so that they can be used as FIFO entries. The fact that one FIFO entry is in the EJ_TCK clock domain is normally transparent, but it can create some unexpected behavior:

- TxFIFO availability: Data is first written into the *SI_Clk* FIFO entries, then it will move into the *EJ_TCK* FIFO entry. But, it takes several *EJ_TCK* cycles to complete the handshake and move the data. *EJ_TCK* is generally much slower than *SI_ClkIn* and may even be stopped (although that would be uncommon when this feature is in use). This can result in there not being space for new data, even though there are only N-1 data values queued up. To prevent the loss of data, the $FDSTAT_{TxF}$ bit is set when all of the *SI_ClkIn* FIFO entries are full. Software writing to the FIFO should always check the $FDSTAT_{TxF}$ bit prior to attempting a write and should not make any assumptions about being able to arbitrarily use all entries. ie. software seeing the $FDSTAT_{FxE}$ bit set should not assume that it can write $FDCFG_{TxCnt}$ data words without checking for full.
- TxFIFO Almost Empty Interrupt: As transmit data moves from *SI_ClkIn* to *EJ_TCK*, both of the flops will temporarily look full. This makes it difficult to determine when just 1 FIFO entry is in use. To enable a simpler condition, the almost empty TxInterrupt condition is set when all of the *SI_ClkIn* FIFO entries are empty. When this

condition is met, there will be 0 or 1 valid entries. However, the interrupt will not be asserted when there is only one valid entry if it is an *SI_ClkIn* entry

- The RxFIFO has similar characteristics but these are even less visible to software since *SI_ClkIn* must be running to access the FDC registers.

14.15.4 Sleep mode

FDC data transfers do not prevent the core from entering sleep mode and will proceed normally in sleep mode. The FDC block monitors the TAP interface signals with a free-running clock. When new receive data is available or transmit data can be sent, the gated clock will be enabled for a few cycles to transfer the data and then allowed to stop again. If FDC interrupts are enabled, transferring data may cause an interrupt to be generated which can wake the core up.

14.16 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.

14.16.1 TCB Trace Enabling

The TCB must be enabled in order to produce a trace to the trace funnel, 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 the trace funnel.

The TCB can optionally include trigger logic, which can control the $TCBCONTROLB_{EN}$ bit. Please see Section 14.16 “TCB Trigger Logic” for details.

14.16.2 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 core.

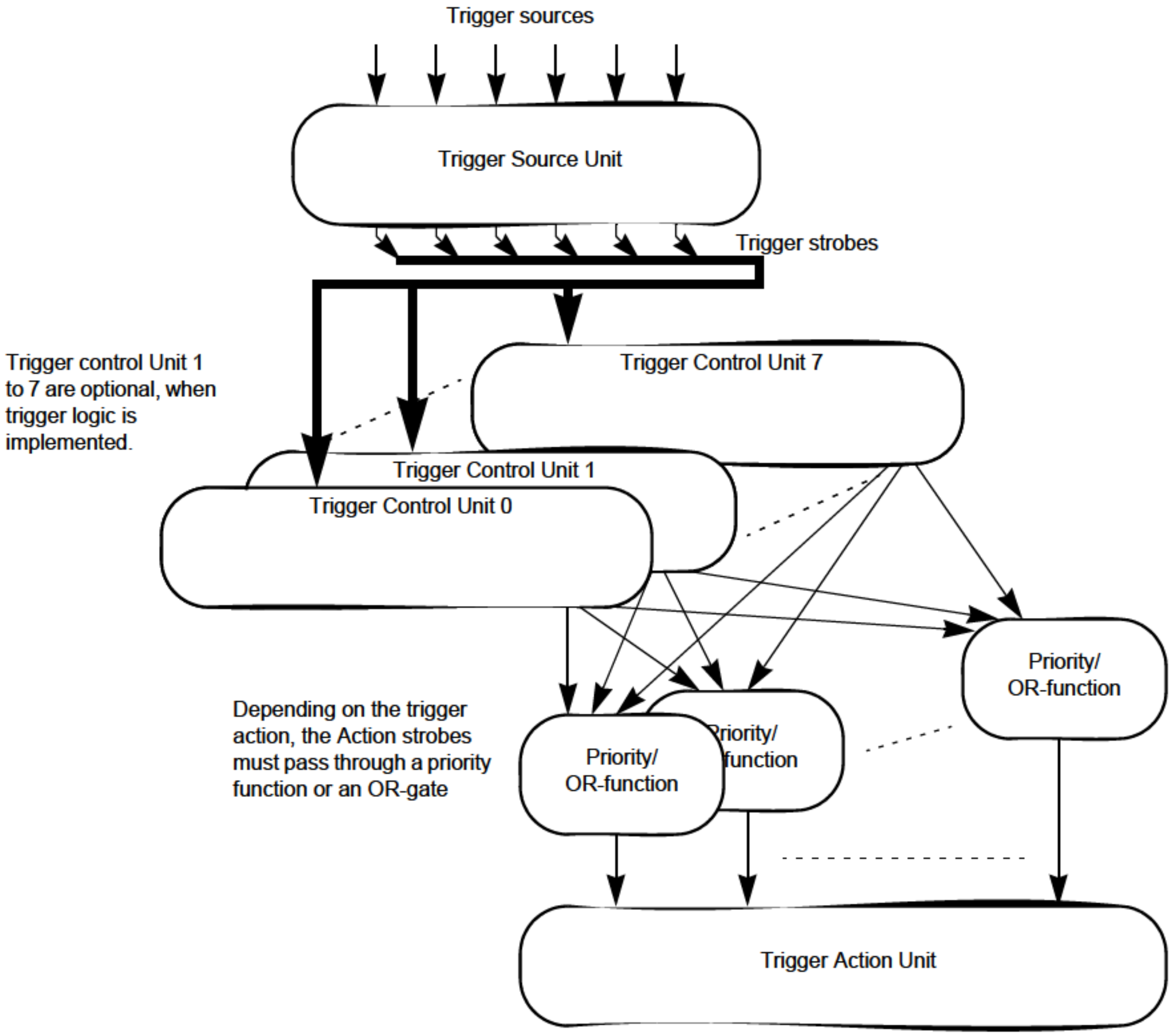
14.16.3 Trigger Units Overview

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 14.41 show the functional overview of the trigger flow in the TCB.

Figure 14.41 TCB Trigger Processing Overview



14.16.4 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 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.

14.16.5 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 14.14.11.3 “TCBTRIGx Register (Reg 16-23)”).

14.16.6 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 14.14.11.3 “TCBTRIGx Register (Reg 16-23)”. Please also read the next Section 14.16.7 “Simultaneous Triggers”.

14.16.7 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.

14.16.7.1 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.

14.16.7.2 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.

Chapter 15

Multi-CPU Debug

This section describes the debug features of the proAptiv Coherent Processing System. The following sections are included in this chapter:

- Section 15.1 “CM Performance Counters”
- Section 15.2 “Debug Mode Triggering”
- Section 15.3 “PDTrace Software Architecture”

15.1 CM Performance Counters

15.1.1 CM Performance Counter Functionality

Performance characteristics of the CM can be measured via the CM performance counters. Two sets of identical programmable 32-bit performance counters in addition to a 32-bit cycle counter are implemented. The counters are controlled and accessed via GCR registers described in Chapter 8, “CM2 Global Control Registers” on page 395. This section describes the operation of those registers.

The counters are started by writing a 1 to the *P0_CountOn*, *P1_CountOn* and *Cycl_Cnt_CountOn* bits in the *CM Performance Counter Control Register* (see Table 8.49 for a description of this register). Each counter can be reset to 0, and the corresponding overflow bit (*P0_Overflow*, *P1_Overflow*, *Cyc_Cnt_Overflow*) is reset to 0 prior to the start of counting by writing a 1 to the *P0_Reset*, *P1_Reset* and *Cycl_Cnt_Reset* bits in the same access that sets the corresponding start bits. This functionality allows all three counters to be reset and started with a single GCR write.

The *CM Performance Counter Control Register* also controls how a counter overflow is handled. If the *Perf_Ovf_Stop* bit is set to 1, then all CM Performance counters will stop when one of the counters (including the Cycle Counter) reaches its maximum value of 0xFFFFFFFF. If instead the *Perf_Ovf_Stop* bit is set to 0, when a counter overflows, it rolls over and continues counting from 0.

If the *Perf_Int_En* bit is set to 1, an interrupt is generated when one of the counters (including the cycle counter) reaches its maximum value of 0xFFFFFFFF. The CM asserts the *CM_PCInt* signal which generates an interrupt only if the System Integrator has connected *CM_PCInt* to one bit of *SI_CMInt*.

When a performance counter overflows, the corresponding bit is automatically set in the *CM Performance Counter Overflow Status Register.* A status bit is cleared by writing a 1 to it.

The event to be counted by each performance counter is designated by the event number set in the *Event_Sel_0* and *Event_Sel_1* fields of the *CM Performance Counter Event Selection Register*. The events corresponding to the event numbers are listed and described in Table 15.1. Each event is further specified by the *CM Performance Counter Qualifier Register*. The meaning of the *CM Performance Counter Qualifier Register* is different for each event. The column labeled “Qualifier” in Table 15.1 shows the qualifiers that can be specified for each event. For example, the qualifiers for the Request_Count event (Event 0) are the request port, CCA, Burst Length, Command, and Target. The details of the qualifiers for the Request_Count event are defined in Table 15.2.

The qualifiers for some events are composed of several groups. A performance counter will increment if the specified event occurs and the qualifier criteria is matched in all groups. For example, assume the *Event_Sel_0* field in the *CM Performance Counter Event Selection Register* is set to 0 (Request_Count). This event occurs when the CM serializes a request. However, the performance counter for this event will only count if the request meets the criteria programmed in all 5 groups in the Request Qualifier (see Table 15.2):

```
    The port that issued the request has the corresponding Request Port qualifier bit
    set to 1
AND
    The Cacheability attribute (CCA) for the request has the corresponding CCA
    qualifier bit set to 1
AND
    The Burst Length of the request (in dwords) has the corresponding qualifier bit set
    to 1
AND
    The OCP MCmd Type for the request has the corresponding Request Command qualifier
    bit set to 1
AND
    The target of the request has the corresponding Target qualifier bit set to 1
```

Multiple bits within a qualification group may be set. In this case, the OR of all bits set within the group. For example, by setting the request port qualifier for Port 0 and Port 1, then a request will be counted if it originated from Port 0 or Port 1.

A qualifier group can be set to “don’t care” by setting all bits within the group to 1. For example, to have performance counter 0 count all requests from port 1, program the *CM Performance Counter Event Selection Register* and *CM Performance Counter Qualifier 0 Register* as follows:

```
    Set Event_Sel_0 to 0 (Request_Count)
    Set Request Port Qualifer bit to 1 for Port 1
    Set Requeset Port Qualifier bits to 0 for all other Ports
    Set all other qualifer bits to 1 (causing the CCA, Burst Length, Command and Target
    to be ignored)
```

The two counters can be programmed to count a different event or the same event with different qualifiers. For example, to measure the ratio of requests from Port 1 vs. all Ports, set program Counter 0 to count requests from Port 1 (see previous example) and program Counter 1 to count all request from all Ports by setting *Event_Sel_1* to 0 (Request_Count) and set *all* bits in the *CM Performance Counter Qualifier 1* Register to 1.

The cycle counter can be used to calculate the average rates of specified events. Continuing the above example, assuming the cycle counter is reset, started, and stopped simultaneously with the two performance counters, then the rate of requests from port 1 and all ports can be easily computed (value of each performance counter / value in cycle counter).

15.1.2 Performance Counter Usage Models

There are several model for using the CM performance counters. This sections discusses 3 possible models:

- Periodic Sampling - take many measurement samples of specific duration
- Stop and Interrupt when counter overflows - counters run until one overflows, then interrupt CPU
- Large count capability - enables unrestricted sample periods

One model for making performance measurements is for the software to set up and gather samples for a set period of time. The code sequence could follow the following steps:

```
start:
Write CM Event and Qualifier Registers for particular event of interest
Write CM Performance Counter Control Register to reset and start counters
   Perf_Int_En = 0 (no interrupt on overflow)
   Perf_Ovf_Stop = 0(no stop on overflow).
   P1_Reset = 1, P1_CountOn = 1
   P0_Reset = 1, P0_CountOn = 1
   Cycl_Cnt_Reset = 1, Cycl_Cnt_CountOn = 1
Wait for some relatively small period of time (i.e., 2 seconds)
Write CM Performance Counter Control Register to stop counters
   P1_Counton = 0, P0_CountOn=0, Cycl_Cnt_CountOn = 0
Read CM Performance Counter 0, Counter 1, and Cycle Counter Registers
If more events, go to start (or if measuring same counter go to step 2 instead)
```

A second CM performance counter usage model involves setting up the counters to stop and interrupt on overflow. This runs the counters until one of the counters (usually the cycle counter) reaches the 32-bit limit. An example of such a code sequence is:

```
start:
Write CM Event and Qualifier Registers for particular event of interest
Write CM Performance Counter Control Register to reset and start counters
   Perf_Int_En = 1 (interrupt on overflow)
   Perf_Ovf_Stop = 1(stop on overflow).
   P1_Reset = 1, P1_CountOn = 1
   P0_Reset = 1, P0_CountOn = 1
   Cycl_Cnt_Reset = 1, Cycl_Cnt_CountOn = 1
When interrupt occurs:
Read CM Performance Counter Status Register
Read CM Performance Counter 0, Counter 1, and Cycle Counter Registers
Write CM Performance Counter Control Register to reset counters
   (clears status register and interrupt)
   P0_Reset = 1, P1_Reset = 1, Cycl_Cnt_Reset = 1
If more events, go to start (or if measuring same counter go to step 2 instead)
```

If larger counts than can fit into the 32-bit counters are required, the counters can be set up to interrupt, but not stop, on overflow. Memory variables can then count the number of overflows, as shown below:

```
start:
Write CM Event and Qualifier Registers for particular event of interest
Write CM Performance Counter Control Register to reset and start counters
   Perf_Int_En = 1 (interrupt on overflow)
   Perf_Ovf_Stop = 0 (do not stop on overflow).
   P1_Reset = 1, P1_CountOn = 1
   P0_Reset = 1, P0_CountOn = 1
   Cycl_Cnt_Reset = 1, Cycl_Cnt_CountOn = 1
When interrupt occurs:
<status>=Read CM Performance Counter Status Register
Increment <overflow_count>[counter] for each counter with <status> = 1
Write <status> to CM Performance Counter Status Register to clear interrupt
```

```
When run limit is reached then :
Write CM Performance Counter Control Register to stop counters
   P1_Counton = 0, P0_CountOn=0, Cycl_Cnt_CountOn = 0
Read CM Performance Counter 0, Counter 1, and Cycle Counter Registers
Write CM Performance Counter Control Register to reset counters
   (clears status register and interrupt)
   P0_Reset = 1, P1_Reset = 1, Cycl_Cnt_Reset = 1
If more events, go to start (or if measuring same counter go to step 2 instead)
```

In the above model, the final counts are calculated for each counter by multiplying `<overflow_count>`[`counter`] by 4G and adding the final values in the performance counter register.

15.1.3 CM Performance Counter Event Types and Qualifiers

This section describes the Performance Counter Event Types and associated qualifiers.

Table 15.1 CM Performance Counter Event Types

Event #	Related Events	Use	Qualifiers	Description/Comments
0	Request_Count	Measuring Load	Request Port Request CCA Request Cmd Request Length Request Target See Table 15.2	Can be used in conjunction with a cycle count to determine number of requests received in a given period of time.
1	Coh_Req_Resp	Track coherent requests or responses, and measure sharing	Intervention State Speculation Intervention Cmd Store Conditional See Table 15.3	Gives a count of the specified coherent request and response types.
2	L2_WR_Data_Util	L2 Write Data Bus Usage	Accept State See Table 15.4	Counts number of cycles the L2/Memory write data bus is occupied. The qualifier determines if stall cycles are counted or not.
3	L2_Cmd_Util	L2 Command Bus Usage	Accept State See Table 15.4	Counts number of cycles the L2/Memory command data bus is occupied. The qualifier determines if stall cycles are counted or not.
4	L2_RD_Data_Util	L2 Read Data Bus Usage	L2 Data Width See Table 15.5	Counts number of cycles the L2/Memory read data bus is occupied. Qualifier determines if 64-bit cycles, 256-bit cycles, or both are counted.
5	Sharing_Miss	Sharing Frequency	Request Source Port Data Source Port See Table 15.6	Counts source of data for coherent read requests only (i.e., CohReadShare, CohReadDiscard, CohReadOwn, and CohReadAlways). Useful to determine how many cache misses were satisfied by other processors.
6	RSU_Util	RSU Usage	Port to measure Response Type See Table 15.7	Counts number of d-words on the processor/iocu read data bus. A counter can only measure one port at a time. The port number is specified as the qualifier.
8	L2_Util	L2 Pipeline Usage	L2 Pipeline starts See Table 15.8	Counts starts into the TA stage of the L2 pipeline.
9	L2_Hit	L2 Hit/Miss Usage	Hit/Miss Type Source Port See Table 15.9	Counts different types of L2 Cache Hits and Misses, crossed with Source Port ID.
16	IOCU_Request	IOCU Request	Transaction ID I/O Parking CM Transaction Cnt BurstLength L2 allocation Posted Cacheability Request Type See Table 15.10	Counts requests receive by the IOCU. The CM receives a sideband signal, SI_CMP_IOC_PerfInfo from the IOCU as described in Table 15.10.

Table 15.1 CM Performance Counter Event Types*(continued)*

Event #	Related Events	Use	Qualifiers	Description/Comments
17	IOCU1_Request	2nd IOCU Request	Transaction ID I/O Parking CM Transaction Cnt BurstLength L2 allocation Posted Cacheability Request Type See Table 15.10	Counts requests receive by the 2nd IOCU. The CM receives a sideband signal, SI_CMP_IOC1_PerfInfo from the 2nd IOCU as described in Table 15.10.

Table 15.2 CM Performance Counter Request Count Qualifier

Bit	Qualifier Group	Qualifier Value	Description/Comments
31	Request Port	Port 7	Request originated from port 7
30		Port 6	Request originated from port 6
29		Port 5	Request originated from port 5
28		Port 4	Request originated from port 4
27		Port 3	Request originated from port 3
26		Port 2	Request originated from port 2
25		Port 1	Request originated from port 1
24		Port 0	Request originated from port 0
23	Request CCA[1]	WT	Request had Write Through Cacheability Attribute
22		UC/UCA	Request had Uncached Cacheability Attribute
21		WB	Request had Cached (non-coherent) Attribute
20		CWBE	Request had Coherent (Exclusive) Attribute
19		CWB	Request had Coherent (Shared) Attribute
18	Burst Length[2] (# of dwords)	1 dword	Request was for 1 dword of data Note: This counts the burst length as seen by the Coherent Manager. Requests from the I/O Subsystem may be longer, but the IOCU may break these into multiple smaller requests.
17		2 dwords	Request was for 2 dwords of data See Note for 1 dword.
16		4 dwords	Request was for 4 dwords of data See Note for 1 dword

Table 15.2 CM Performance Counter Request Count Qualifier*(continued)*

Bit	Qualifier Group	Qualifier Value	Description/Comments
15	Request Command	Legacy WR	Request is a legacy Write command. This is used for all non-coherent writes. Note: When a processor is in coherent mode, L1 cache writebacks are always considered coherent, so they result in a cohWriteBack command, not a WR command.
14		Legacy RD	Request is a legacy Read command. This is used for all non-coherent reads, including code fetches.
13		CohReadShare CohReadShareAlways	Request is a coherent read share generated by the processor on a load that misses its L1 cache. Currently CohReadShareAlways is unused.
12		CohReadOwn	Request is a coherent read own generated by the processor on a store that misses its L1 cache.
11		CohReadDiscard	Request is a coherent read discard generated by the IOCU for coherent requests.
10		CohUpgrade	Request is a coherent upgrade request generated by the the processor on a store that hits a shared line in its L1 cache.
9		CohWriiteBack	Request is coherent writeback generated by the processor when evicting a line from the L1 cache. The line may have been installed in the cache from a coherent or non-coherent transaction.
8		CohWriteInval (Partial Line)	Request is a coherent write invalidate (not a full line of data) generated by the IOCU.
7		CohWriteInval (Full Line)	Request is a coherent write invalidate (full line of data) generated by the IOCU.
6		CohInvalidate	Request is an invalidate request from a processor executing a PREF Prepare for Store or a CACHE Hit Invalidate.
5		CohCopyBack	Request from a processor executing a CACHE hit writeback
4		CohCopyBackInv	Request from a processor executing a CACHE hit CACHE WriteBackInvalidate
3		CohCompletionSync	Request is from a processor executing a SYNC instruction
2	Target	Memory	Request targets memory (coherent or non-coherent)
1		GCR/GIC/CPC	Request targets the Interrupt controller or Global Control Registers
0		MMIO	Request targets Memory Mapped I/O space

1. CCA qualifier group is ignored on non-coherent cache-ops
2. Burst Length only used when Request Command is Legacy Read, Legacy Write, CohReadDiscard or CohWriteInval.

Table 15.3 CM Performance Counter Coherent Request/Response Qualifier

Bit	Qualifier Group	Qualifier Value	Description/Comments
31:25	Reserved		

Table 15.3 CM Performance Counter Coherent Request/Response Qualifier*(continued)*

Bit	Qualifier Group	Qualifier Value	Description/Comments
24	Intervention State	Exclusive with data	A processor has an exclusive copy in its L1 cache and returned data (all commands except CohInvalidate)
23		Exclusive with no data	A processor has an exclusive copy in its L1 cache but no data was returned (occurs on a CohInvalidate)
22		Modified with data	A processor has a modified copy in its L1 cache and returned data (all commands except CohInvalidate)
21		Modified with no data	A processor has a modified copy in its L1 cache but no data was returned (occurs on a CohInvalidate)
20		Shared	One or more processors have a shared copy in its L1 cache
19		Invalid	No processor has a copy of the data in its L1 cache
18	Speculation	Speculate	Request was a CohReadShare, CohReadOwn, CohReadDiscard or CohReadAlways and the CM issued a speculative read request to L2/Memory. This qualifier group is ignored when the request is not one of the commands listed above.
17		No Speculate	Request was a CohReadShare, CohReadOwn, CohReadDiscard or CohReadAlways and the CM did not issue a speculative read request to L2/Memory. This qualifier group is ignored when the request is not one of the commands listed above.
16	Intervention Cmd	Reserved	Currently a don’t care.
15		Reserved	Currently a don’t care.
14		CohReadShare	Request is a coherent read share generated by the processor on a load that misses its L1 cache.
13		CohReadShareAlways	Currently CohReadShareAlways is unused.
12		CohReadOwn	Request is a coherent read own generated by the processor on a store that misses its L1 cache.

Table 15.3 CM Performance Counter Coherent Request/Response Qualifier*(continued)*

Bit	Qualifier Group	Qualifier Value	Description/Comments
11	Intervention Cmd (cont.)	CohReadDiscard	Request is a coherent read discard generated by the IOCU for coherent requests.
10		CohUpgrade (OK Response)	Request is a coherent upgrade request generated by the processor on a store that hits a shared line in its L1 cache. There is no intervening request to the same line so an OK response is given.
9		CohUpgrade (Data Response)	Request is a coherent upgrade request generated by the processor on a store that hits a shared line in its L1 cache. There is an intervening request to the same line so a data response is given.
8		CohWriteBack	Request is coherent writeback generated by the processor when evicting a line from the L1 cache. The line may have been installed in the cache from a coherent or non-coherent transaction.
7		CohWriteInval (Partial Line)	Request is a coherent write invalidate (not a full line of data) generated by the IOCU.
6		CohWriteInval (Full Line)	Request is a coherent write invalidate (full line of data) generated by the IOCU.
5		CohInvalidate	Request is an invalidate request from a processor executing a PREF Prepare for Store or a CACHE Hit Invalidate.
4		CohCopyBack	Request from a processor executing a CACHE hit writeback
3		CohCopyBackInv	Request from a processor executing a CACHE hit CACHE WriteBackInvalidate
2	Store Conditional (only used when cmd is CohUpgrade or CohReadOwn)	Not due to a Store Conditional	CohUpgrade or CohReadOwn is not due to a store conditional instruction. This qualifier group is ignored when the command is not a CohUpgrade or CohReadOwn.
1		Store Conditional that was not Cancelled	CohUpgrade or CohReadOwn is due a store conditional instruction and the intervention was not cancelled. This qualifier group is ignored when the command is not a CohUpgrade or CohReadOwn.
0		Store Conditional that was Cancelled	CohUpgrade or CohReadOwn is due a store conditional instruction and the intervention was cancelled due to livelock avoidance scheme. This qualifier group is ignored when the command is not a CohUpgrade or CohReadOwn.

Table 15.4 CM Performance Counter Accept State Qualifier

Bit	Qualifier Group	Qualifier Value	Description/Comments
31:1	Reserved		
0	Accept State	Count_Stalls	Setting this value to 0 for the L2_WR_Data_Util or L2_Cmd_Util events cause a count of cycles when a data word or command is accepted by the L2/Memory. Setting this value to 1 for L2_WR_Data_Util or L2_Cmd_Util cause a count of cycles when a data word or command is valid on the bus, i.e., the count includes cycles where the command or data bus is stalled.

Table 15.5 CM Performance Counter L2 Data Width Qualifier

Bit	Qualifier Group	Qualifier Value	Description/Comments
31:2	Reserved		
1	L2 Data Width	256	Counts cycles where the L2/Memory returns data in 256-bit mode
0		64	Counts cycles where the L2/Memory returns data in 64-bit mode

Table 15.6 CM Performance Counter CM Data Source Qualifier

31:15	Reserved		
14	Request Port	7	Request originated from port 7
13		6	Request originated from port 6
12		5	Request originated from port 5
11		4	Request originated from port 4
10		3	Request originated from port 3
9		2	Request originated from port 2
8		1	Request originated from port 1
7		0	Request originated from port 0
6	Response Port	5	Data returned by processor connected to port 5
5		4	Data returned by processor connected to port 4
4		3	Data returned by processor connected to port 3
3		2	Data returned by processor connected to port 2
2		1	Data returned by processor connected to port 1
1		0	Data returned by processor connected to port 0
0		L2/Mem	Data returned by L2/Memory

Table 15.7 CM Performance Counter CM Port Response Qualifier

Bit	Qualifier Group	Qualifier Value	Description/Comments
31:6	Reserved		
5	Response Type	Read Data Response	Response was a dword of data.
4		Write Acknowledge Response	Response was a write acknowledge (DVA response for a write).
3		OK Response	Response was an OK response (due to a CohUpgrade).
2:0	Port Number	Port to measure	Encoded value of port number to measure. For example, a value of 2 will only count responses on response port 2.

Table 15.8 L2 Utilization Qualifier

Bit	Qualifier Group	Qualifier Value	Description/Comments
31:6	Reserved		
5	Pipeline Start Type	L2 Pipeline start was stalled	Any type of pipeline request start (new, replay,refill) was refused due to a stall (ram or global stall)
4		L2 Pipeline start is taken	Use to calculate L2 utilization Any type of pipeline request start (new, replay,refill)
3		New request waiting for Sync to clear	A new request is waiting to be dispatched to the L2 until a preceeding Sync has guaranteed ordering
2		New L2 request stalled	New request to the L2 was not accepted due to a stall (ram or global stall)
1		New L2 request denied	New request to the L2 was not accepted due to replay, refill, or a stall.
0		New L2 request started	Use to calculate L2 bandwidth

Table 15.9 L2 Hit Qualifier

Bit	Qualifier Group	Qualifier Value	Description/Comments
31:20	Reserved		
19	Allocation (for Write or Read misses only)	Line allocated	A miss caused an allocation by the L2. This occurs either for a full line write miss or a read miss, depending on the L2 allocation policy.
18		Line not allocated	A miss did not cause an allocation by the L2.
17	Hit/Miss Type (these are mutially exclusive)	Other	Index L2 cacheop or Fetch&Lock.
16		Non-index cache-op hit	Non-index L2 cacheop hit the L2 cache.
15		Non-index cache-op miss	Non-index L2 cacheop missed the L2 cache.
14		Full line write hit	Full line write hit the L2 cache.
13		Partial line write hit	Partial line write hit the L2 cache. The line will be read, merged with the original write data, and replayed to complete the write.
12		Full line write miss	Full line write missed the L2 cache. Either allocates or writes through to memory depending on the L2 allocation policy.
11		Partial line write miss	Partial line write missed the L2 cache. Writes through to memory regardless of the L2 allocation policy.
10		Read into CRQ	Read matched a pending L2 miss. Data is returned when the pending line is refilled. It is not a Read hit or a Read miss.
9		Read hit	Read hit the L2 cache.
8		Read miss	Read missed the L2 cache. Either allocates or reads through to memory, depending on the L2 allocation policy.

Table 15.9 L2 Hit Qualifier*(continued)*

Bit	Qualifier Group	Qualifier Value	Description/Comments
7	Source Port	7	Request originated from port 7
6		6	Request originated from port 6
5		5	Request originated from port 5
4		4	Request originated from port 4
3		3	Request originated from port 3
2		2	Request originated from port 2
1		1	Request originated from port 1
0		0	Request originated from port 0

Table 15.10 IOCU Performance Counter Request Count Qualifier

Bit	Qualifier Group	Qualifier Value	Description/Comments
31	Reserved		
30:27	Transaction ID	TID	Value of IC_MTagID to match when the All_TID qualifier bit is set to 0. This field is unused when All_TID is 1.
26		All_TID	If 1 then the all values of IC_MTagID will match. If 0 then only transactions with IC_MTagID equal to the TID specified above will match.
25	I/O Parking	Start and Stop Parking	Request will start and stop I/O Parking.
24		Stop Parking	Request will stop I/O parking (but not start it).
23		Start Parking	Request will start I/O Parking (but not stop it).
22		No parking	Request will not start or stop I/O parking.
21	CM Transaction Count	5 CM Transactions	Request resulted in 5 CM transactions.
20		4 CM Transactions	Request resulted in 4 CM transactions.
19		3 CM Transactions	Request resulted in 3 CM transactions.
18		2 CM Transactions	Request resulted in 2 CM transactions.
17		1 CM Transaction	Request resulted in 1 CM transaction.
16	BurstLength	13-16	IC_MBurstLength is 13, 14, 15, or 16 dwords.
15		9-12	IC_MBurstLength is 9, 10, 11, or 12 dwords.
14		5-8	IC_MBurstLength is 5, 6, 7, or 8 dwords.
13		4	IC_MBurstLength is 4 dwords.
12		3	IC_MBurstLength is 3 dwords.
11		2	IC_MBurstLength is 2 dwords.
10		1	IC_MBurstLength is 1 dword.

Table 15.10 IOCU Performance Counter Request Count Qualifier*(continued)*

Bit	Qualifier Group	Qualifier Value	Description/Comments
9	L2 Allocation	L2 Allocation with Prepare for Store	Request will cause an L2 allocation and the request is a write with L2 Prepare For Store. This bit will never cause a match for read requests.
8		L2 Allocation without Prepare for Store	Request will cause an L2 allocation and the request is either a read or a write with L2 Prepare For Store not asserted.
7		No L2 Allocation	Request will not cause an L2 allocation.
6	Posted	Non-posted Write	Write is non-posted. Not used on reads.
5		Posted Write	Write is posted. Not used on reads.
4	Cacheability	Uncached	Request is uncached.
3		Cached	Request is Cached, non-coherent.
2		Coherent	Request is Coherent.
1	Request Type	Read	Request is a read.
0		Write	Request is a write.

15.2 Debug Mode Triggering

This section describes the how to control the cores when entering debug mode.

15.2.1 Selecting CPUs to Enter Debug Mode

The proAptiv Multiprocessing System contains a set of registers and logic that controls when the proAptiv Multiprocessing System cores enter Debug mode. The logic allows software to:

- Specify which proAptiv Multiprocessing System the CPU enters debug mode on assertion of the *EJ_DINT_IN* signal (generally asserted by a debug probe).

- Force one or more proAptiv Multiprocessing System CPUs to enter debug mode by writing to the *DINT Send to Group Register*.

15.2.2 Debug Mode Groups and Cross Triggering

The proAptiv Multiprocessing System (MPS) allows software to define debug mode groups so that when one proAptiv Multiprocessing System core enters debug mode, all other cores within the group also enter debug mode.

Software creates debug mode groups by writing to each CPU’s CPU-*Local DebugBreak Group Register*. Each bit in the *Join_DebugM* field of the CPU-*Local DebugBreak Group Register* represents a CPU in the system. If the bit is set, the corresponding CPU will enter debug mode. If the bit is clear, the corresponding CPU is not affected by Debug Mode.

Only the positive edge of a CPU’s *EJ<cpu>_DebugM* signal can cause the other CPUs to also enter the Debug Mode as a group. When there is no positive edge on the *DebugM* signals, the *Join_DebugM* fields in the *DebugBrk_Group* registers can be written without causing spurious glitches on the *EJ<cpu>_DINT* signals.

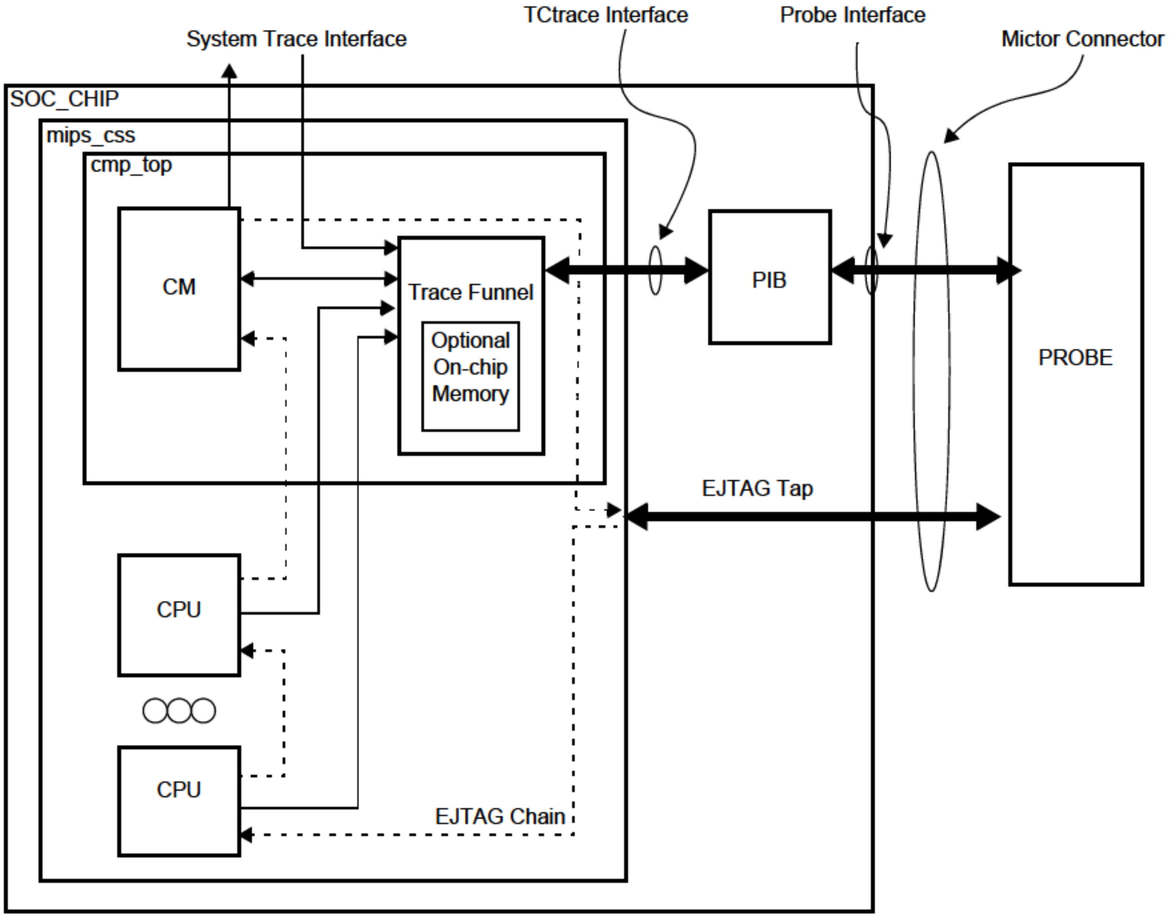
15.2.3 Debug Cross Trigger Facility and Power Management

Due to power management of proAptiv Multiprocessing System components, CPUs might not be powered or clocked when receiving a DINT via the debug cross trigger facility. However, the power controller observes all DINT events and will start up domains as requested. Depending on the programming of the power controller and time constants of the physical design, a delay between DINT event and a target CPU participating in the debug session might occur. To inquire about the current power status of a CPU, the debug handler can poll the power controller status registers. Generally, an EJTAG debug probe attached and recognized by the system will shorten the wake-up delay, while debug events without debug probe attachment might show more wake-up latency.

15.3 PDTrace Software Architecture

The proAptiv MPS enables debug trace information from the proAptiv Multiprocessing System CPUs, the Coherence Manager, and a System Trace Interface to be streamed off chip or stored in on-chip RAM. As shown in Figure 15.1, each proAptiv Multiprocessing System CPU produces a 64-bit debug trace stream describing its program and data flow. The CM produces a 64-bit stream describing the flow of transactions within the CM. If a System Trace Interface is part of the build, it captures a 128-bit stream describing activity supplied externally by the System. The Trace Funnel muxes the CPU, CM, and System Trace streams into a single debug trace stream which is either stored in an on-chip buffer or passed onto a Probe Interface Block (PIB). A PIB is the on-chip link between the Trace Funnel and debug probe interface, and may include functionality such as time multiplexing the 64-bit TCtrace data onto a narrower, slower probe interface.

Figure 15.1 PD Trace Architecture



The TCtrace stream consists of 64-bit trace words (TW). Each trace word trace is packed with one or more Trace Formats (TF). There are many trace format types produced by CPUs and the CM. The CPU TFs allow tracing of information such as the program counter, load/store addresses, and load/address data values. The CM TFs produce information such as the serialization order of requests and the results of L1 cache interventions. See *The PDtrace® Interface and Trace Control Block Specification* for a detailed description of the TW and TF formats.

The trace output of each CPU can be controlled by a set of EJTAG accessible registers located in the Trace Control Block (TCB) associated with that CPU. Refer to *The MIPS32® proAptiv Multiprocessing System™ CPU Family Software User's Manual* for a detailed description of these registers and the CPU PDTrace functionality.

15.3.1 CM Trace Functionality

This section describes the configuration and functionality of the CM debug trace.

15.3.1.1 CM Trace Configuration and Control

The CM Trace is controlled by the *CM TCBCONTROLD Register* as defined in Section "TCBCONTROLD Register". The enabling of the CM's Trace is determined by two fields in this register along with a field in each of the core's TCBCONTROLD register. Figure 15.11 shows that there are two ways to enable CM Trace. First, CM Trace can be enabled independent of the Cores' state by setting both *CM_EN* and *Global_CM_En* in the CM's *TCBCONTROLD*

Register. Alternatively, by setting *CM_EN* and clearing *Global_CM_En*, the CM will only trace if at least one other core is tracing, i.e., *Core_CM_En* in at least one core's TCBCONTROLD register is set to 1. A core's *Core_CM_En* bit may be asserted/deasserted based on debug triggers as defined in *The MIPS32® proAptiv Multiprocessing System™ Processor Core Family Software User's Manual*. The value of each core's *Core_CM_En* bit is communicated to the CM on the *TC<core>_Trace_CM_En* signal.

Table 15.11 CM Trace Enable

CM TCBCONTROLD Reg		**Cores' TCBCONTROLD Reg**	**CM PDTrace Enabled/Disabled**
CM_EN	**Global_CM_En**	**Core_CM_En**	
0	x	x	Disabled
1	1	x	Enabled
1	0	All 0	Disabled
1	0	not All 0	Enabled

15.3.1.2 System Trace Interface Configuration and Control

The System Trace Interface stream is generated and controlled by external logic. The CM has control output pins to support design of this logic. There are 2 specific control outputs and one 32-bit user-defined output. These outputs and the trace data/contol pins associated with the trace stream are shown in Table 15.12. All the signals are timed relative to the *SI_CMClk*.

Table 15.12 System Trace Interface Stream and Control Pins

Signal	**Direction/Type**	**Usage**
SI_TC_Sys_Data[127:0]	CM stream input	System Trace stream data for 128-bit stream SI_TC_Sys_Data[71:68] must contain a Source Port ID and SI_TC_Sys_Data[7:4] must contain a Source Port ID. Legal values of either Source Port ID are: 4'hc or 4'hd. All other bits are completely user defined
SI_TC_Sys_Valid[1:0]	CM stream input	System Trace stream valid bits for upper and lower streams Bit 1 qualifies SI_TC_Sys_Data[127:64] Bit 0 qualifies SI_TC_Sys_Data[63:0] A value of 2'b10 is illegal
SI_TC_Sys_Stall	CM stream output	System Trace stream flow control.
SI_TC_Sys_Enable	CM control output	System Trace control advisory, driven from the *CM TCBCONTROLD*$_{ST\_En}$. Its purpose is to advise the external logic of the state of this control bit. If desired, external logic can stop generation of the stream if this output is a zero, and allow generation of the stream if it is a 1. However, external logic may choose to continue sending stream data after de-assertion until it has flushed all its collected stream data.
SI_TC_Sys_AnyCore_Enabled	*CM control output*	*System Trace control advisory that at least one core is enabled to trace, derived from Cores' TCBCONTROLD*
SI_TC_Sys_CM_Enabled	*CM control output*	*System Trace control advisory that the CM2 is enabled to trace, derived from CM2's TCBCONTROLD*

Table 15.12 System Trace Interface Stream and Control Pins

Signal	Direction/Type	Usage
SI_TC_Sys_UserCtl[31:0]	CM control output	User defined control advisory bits, from TCBSYS. Bit 31 is a 1 when the Trace Funnel was configured with the System Trace present and is a 0 when the System Trace is not present. Bits [30:0] are completely user defined output values.

In addition to the System Trace Interface pins, there are internal control register bits that impact operation of the System Trace stream. Assertion of *CM TCBCONTROLB*$_{\mathrm{STCE}}$ allows the System Trace funnel port to capture stream data; de-assertion of this bit causes the Trace Funnel to stop capturing the System Trace stream from within the Trace Funnel in case the external logic is problematic. In addition, de-assertion of *CM TCBCONTROLB*$_{\mathrm{EN}}$ stops capture of all the streams (Cores, CM, System).

Thus the System Trace stream is enabled to capture the System Trace stream when these controls are asserted: *CM TCBCONTROLB*$_{\mathrm{STCE}}$ and *CM TCBCONTROLB*$_{\mathrm{EN}}$. The control outputs *SI_TC_Sys_Enable* and *SI_TC_Sys_UserCtl[31:0]* are available to the external logic to further control generation of the System Trace stream by allowing or disallowing assertion of the *SI_TC_Sys_Valid[1:0]* inputs. If any trace stream is being generated without enabling that stream to capture, then that stream is not captured and the data is dropped.

15.3.1.3 Trace Funnel Enable

When trace on the System, CM and/or Cores is enabled then trace information is continuously sent to the Trace Funnel. However, the trace funnel will only send the trace information to the trace probe or to the on-chip trace memory if it is enabled by setting the *CM TCBCONTROLB*$_{\mathrm{EN}}$ bit. The Trace Funnel can be subsequently disabled by clearing the *CM TCBCONTROLB*$_{\mathrm{EN}}$ bit. See “TCBCONTROLB Register Field Descriptions” on page 774 for more information.

15.3.1.4 CM Trace Formats

Trace information is captured at two points within the CM:

- Information about requests is captured by the Request Unit (RQU) after serialization, thus providing a view of the global order of requests.
- Information about L1 interventions is captured by the Intervention Unit (IVU) after all intervention responses have been received. This provides information about the state of the cache line in all L1 caches for coherent requests.

The type and amount of content in each Trace Format created by the CM depends on the source of the packet (RQU or IVU) and the configuration (TLev, AE, P<port>_Ctl control bits). Refer to *The PDtrace™ Interface and Trace Control Block Specification* for the detailed description of the CM Trace Formats.

15.3.1.5 CM / CPU Core Trace Correlation

In the proAptiv, trace information is provided from each of the CPUs as well as the Coherence Manager. In order to correlate transactions from the CM to the instruction stream, an identifier is used in both the CPU and CM traces.

The CM trace includes the core ID and CosID for each request. The CosID changes relatively slowly - it is generally incremented after PCSync in the CPU or if an overflow is detected in the CM. Typically several requests in a row will use the same CosID value, and the intermediate correlation is enabled by the requests appearing in the same order in the CM and CPU traces. Because of this, and the fact that the CosID is traced as a part of the instruction completion record, correlating instructions to CM transactions is possible only when PC tracing is enabled for all TCs executing on the CPU.

The PDtrace™ Interface and Trace Control Block Specification includes a more detailed description of the correlation process.

15.3.2 Controlling Trace in a Multi-CPU Multiprocessing System

The proAptiv MPS enables debug trace information from the proAptiv Multiprocessing System CPUs and the Coherence Manager to be streamed off chip or stored in on-chip RAM. As shown in Figure 15.1, each proAptiv Multiprocessing System CPU produces a 64-bit debug trace stream describing its program and data flow. The CM produces a stream describing the flow of transactions within the CM2. The Trace Funnel muxes the CPU and CM trace streams into a single debug trace stream which is either stored in an on-chip buffer or passed onto a Probe Interface Block (PIB). A PIB is the on-chip link between the Trace Funnel and debug probe interface, and may include functionality such as time multiplexing the 64-bit TCtrace data onto a narrower, slower probe interface.

Since the proAptiv Multiprocessing System core streams PDTrace data directly to the trace funnel, the core TCB system is configured as if only off-chip trace is present. Core TCB register bits which refer to control of on-chip trace resources will behave as it on-chip trace is not implemented.

The CM has its own set of TCBControl registers. It is designated as the ‘master’ which controls trace functionality for the CM, the on-chip trace buffer, and the PIB interface. In addition to the CM2 as trace master, the GCR block itself can function as the trace master in the proAptiv Multiprocessing System . This is done through memory mapped CM_GCR global control registers.

15.3.3 EJTAG Debug Support in the proAptiv Coherence Manager

The EJTAG debug logic in the Coherence Manager is compliant with EJTAG Specification 5.0 and includes:

1. Standard Test Access Port (TAP) for a dedicated connection to a debug host
2. Optional PDtrace capability for program counter/data address/data value trace to On-chip memory or to Trace probe

The following sub-sections describe the TAP and EJTAG operation and registers.

15.3.3.1 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.

EJTAG Internal and External Interfaces

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

Table 15.13 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 CM clock, so the EJTAG probe can drive *TCK* independently of the CM clock frequency. The CM 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 CM 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 CM 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 CM 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 main CM logic. The CM's transaction processing logic is not reset by the assertion of *TRST_N*. The CM 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.

Test Access Port Operation

The TAP controller is controlled by the Test Clock (*TCK*) and Test Mode Select (*TMS*) inputs. These two inputs determine whether an the Instruction register scan or data register scan is performed. The TAP consists of a small controller, driven by the *TCK* input, which responds to the *TMS* input as shown in the state diagram in Figure 15.2. The TAP uses both clock edges of *TCK*. *TMS* and *TDI* are sampled on the rising edge of *TCK*, while *TDO* changes on the falling edge of *TCK*.

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

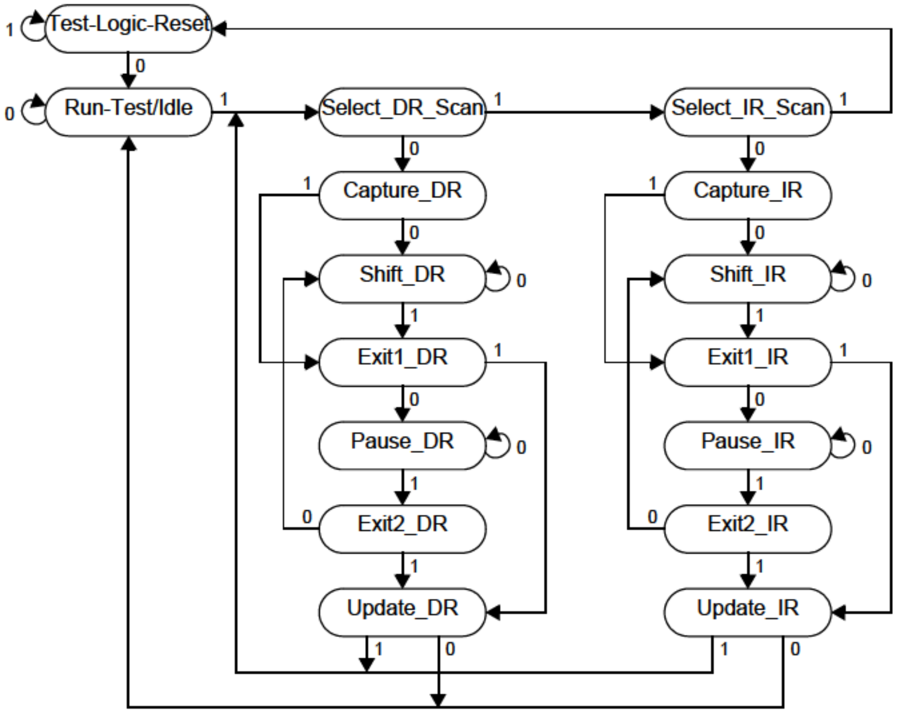
When test access is required, a protocol is applied via the *TMS* and *TCK* inputs, causing the TAP to exit the *Test-Logic-Reset* state and move through the appropriate states. From the *Run-Test/Idle* state, an Instruction register scan or a data register scan can be issued to transition the TAP through the appropriate states shown in Figure 15.2.

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 15.2 TAP Controller State Diagram



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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

15.3.3.2 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 15.14 Implemented EJTAG Instructions

Value	Instruction	Function
0x01	IDCODE	Select Chip Identification data register.
0x03	IMPCODE	Select Implementation register.
0x08	Reserved	Instructions using this code select bypass register.
0x09	Reserved	Instructions using this code select bypass register.
0x0A	CONTROL	Select EJTAG Control register.
0x0B	Reserved	Instructions using this code select bypass register.
0x0C	Reserved	Instructions using this code select bypass register.
0x0D	Reserved	Instructions using this code select bypass register.
0x0E	Reserved	Instructions using this code select bypass register.
0x10	Reserved	Instructions using this code select bypass register.
0x11	TCBCONTROLB	Selects the *TCBCONTROLB* register in the Trace Control Block.
0x12	TCBDATA	Selects the *TCBDATA* register in the Trace Control Block.
0x13	Reserved	Instructions using this code select bypass register.
0x14	Reserved	Instructions using this code select bypass register.
0x15	TCBCONTROLD	Selects the *TCBCONTROLD* register in the Trace Control Block.
0x16	TCBCONTROLE	Selects the *TCBCONTROLE* register in the Trace Control Block.
0x17	Reserved	Instructions using this code select bypass register.
0x1F	BYPASS	Bypass register.

BYPASS Instruction

The required BYPASS instruction selects the Bypass register to be connected between *TDI* and *TDO*. The BYPASS instruction allows serial data to be transferred through the CM 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.

IDCODE Instruction

The IDCODE instruction 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 CM. 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.

IMPCODE Instruction

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

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*.

TCBCONTROLB Instruction

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

TCBDATA Instruction

This instruction is used to select the TCBDATA register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace Control Block is present. If no TCB is present, then this instruction will select the Bypass register. It should be noted that the TCBDATA register is only an access register to other TCB registers. The width of the TCBDATA register is dependent on the specific TCB register.

TCBCONTROLD Instruction

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

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 Control Block is present. If no TCB is present, then this instruction will select the Bypass register.

15.3.3.3 EJTAG TAP Registers

The EJTAG TAP Module has one Instruction register and a number of data registers, all accessible through the TAP:

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 15.14.

15.3.3.4 Data Registers Overview

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

This description applies in general to the following data registers:

- Bypass Register
- Device Identification Register
- Implementation Register
- EJTAG Control Register (ECR)

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.

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 15.15 shows the bit assignments defined for the read-only Device Identification Register, and inputs to the CM 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 15.3 Device Identification Register Format

31 28	27 12	11 1	0
Version	PartNumber	ManufID	R

Table 15.15 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 CM.	R	*EJ_Version[3:0]*
PartNumber	27:12	**Part Number** (16 bits) This field identifies the part number of the CM.	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

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 CM2. The register is selected when the Instruction register is loaded with the IMPCODE instruction.

Figure 15.4 Implementation Register Format

31 29	28 14	13 11	10 1	0
EJTAGver	reserved	Type	TypeInfo	r

Table 15.16 Implementation Register Descriptions

Fields		Description	Read / Write	Reset State
Name	**Bit(s)**			
EJTAGver	31:29	Indicates EJTAG Version 5.0.	R	5
reserved	28:14	reserved	R	0
Type	13:10	Type of Entity associated with this TAP. 2: TAP is attached to a Trace-Master. TypeInfo field is not used.	R	2
TypeInfo	10:1	Identifier Information. Unused because this TAP is conected to a Trace-Master as indicated by the Type field.	R	0
reserved	0	reserved	R	0

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.

The value used for reset indicated in the table below takes effect on CM2 resets, but not on TAP controller resets by e.g. *TRST_N*. *TCK* clock is not required when the CM2 reset occurs, but the bits are still updated to the reset value when the *TCK* is applied. The first 5 *TCK* clocks after CM2 resets may result in reset of the bits, due to synchronization between clock domains.

Figure 15.5 EJTAG Control Register Format

31	28 23	22	21	20 0
Rocc	reserved	Doze	Halt	reserved

Table 15.17 EJTAG Control Register Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
Rocc	31	Reset Occurred The bit indicates if a CM 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
Res	30:23	reserved	R	0
Doze	22	Tied to 0.	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 CM clock is running 1: Internal CM clock is stopped	R	0
Res	20:0	reserved	R	0

15.3.3.5 CM2 Trace Control Block (TCB) Registers

The TCB registers used to control its operation are listed in Table 15.18 and Table 15.19. These registers, except for *TCBDATA*, are accessed via the EJTAG TAP interface as well as by the proAptiv Core via memory-mapped accesses to the Global Debug Control Block in the CM GCRs. *TCBDATA* can only be accessed via the EJTAG TAP interface. Note that none of the TCB registers are implemented if PDTrace is not configured at build time.

Table 15.18 TCB EJTAG Registers

EJTAG Register	Memory-Mapped Address*	Name	Description
0x11	0x0008	*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 15.19. See Section “TCBCONTROLB Register”.
0x15	0x0010	*TCBCONTROLD*	Control register in the TCB used to control tracing from the Coherence Manager Section “TCBCONTROLD Register”
0x16	0x0020	*TCBCONTROLE*	Control Register in the TCB used to control tracing for the performance counter tracing feature. See Section “TCBCONTROLE Register”.

Table 15.19 Registers Selected by TCBCONTROLB$_{REG}$

TCBCONTROLB$_{REG}$ Field	Memory Mapped Address*	Name	Reference	Notes
0	0x0028	TCBCONFIG	Section “TCBCONFIG Register (Reg 0)”	
4	0x0200/0x0208**	TCBTW	Section “TCBTW Register (Reg 4)”	These registers have no function if on-chip memory does not exist.
5	0x0108	TCBRDP	Section “TCBRDP Register (Reg 5)”	
6	0x0110	TCBWRP	Section “TCBWRP Register (Reg 6)”	
7	0x0118	TCBSTP	Section “TCBSTP Register (Reg 7)”	
17-29		reserved		
30	0x0040	TCBSYS	Section “TCBSYS Register (Reg 30)”	
31		TCBBYPASS		

* Memory-Mapped Address relative to the Global Debug Block in the CM GCRs.
** Memory-Mapped Access for TCBTW is split into two 32-bit registers: TCBTW_LO (address 0x0200) accesses TCBTW[31:0]. TCBTW_HI (address 0x0208) accesses TCBTW[63:32]

TCBCONTROLB Register

The TCB includes a second control register, *TCBCONTROLB* (EJTAG Register 0x11). This register generally controls what to do with the trace information received. This register is also mapped to offset 0x0008 in the Global Debug Block of the CM GCRs.

The format of the *TCBCONTROLB* register is shown below, and the fields are described in Table 15.20.

Figure 15.6 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	STCE	TRPAD	0	RM	TR	BF	TM	0	CR	Cal	0	CA	OfC	EN

Table 15.20 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
Reserved	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. The value for the CM is always 0b10 indicationg a four bit source field width.	R	10
REG	25:21	Register select: This field select the registers accessible through the *TCBDATA* register. Legal values are shown in Table 15.19. Note: Although this field can be written via memory-mapped GCR or EJTAG accesses, the *TCBDATA* register is only accessible via EJTAG access.	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. Note: Although this field can be written via memory-mapped GCR or EJTAG accesses, the *TCBDATA* register is only accessible via EJTAG access.	R/W	0

Table 15.20 TCBCONTROLB Register Field Descriptions*(continued)*

Fields		Description	Read / Write	Reset State
Name	Bits			
STCE	19	System Trace capture enable. When asserted, the System Trace port of the Funnel is enabled to capture System Trace stream data. When not asserted, System Trace stream data is not captured regardless of *SI_TC_Sys_Valid[1:0]* input pin state.	R/W	0
TRPAD	18	Trace RAM access disable bit. When set to 1 core reads and writes to the on-chip trace RAM using GCR accesses are inhibited. If TRPAD is set, memory-mapped writes to the GCR_DB_TCBTW_LO and GCR_DB_TCBTW_HI registers have no effect, and memory-mapped reads from GCR_DB_TCBTW_LO and GCR_DB_TCBTW_HI do not access the Trace RAM and 0 is returned. Also, when TRPAD is set, then memory-mapped writes to all CM TCB registers listed in Table 15.19 are inhibited.	R/W	0
Reserved	17	Reserved. Must be written as zero; returns zero on read.	R	0
RM	16	Read on-chip trace memory. When written to 1, the read address-pointer of the on-chip memory in register *TCBRDP* is set to the value held in *TCBSTP*. Subsequent access to the *TCBTW* register (through the *TCBDATA* register), will automatically increment the read pointer in register *TCBRDP* after each read. 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 has no function if on-chip memory is not implemented.	R/W1	0
TR	15	Trace memory reset. Trace memory reset. When written to one, the address pointers for the on-chip trace memory *TCBSTP, TCBRDP and TCBWRP* are reset to zero. Also the RM and BF bits are reset to 0. This bit is automatically reset back to 0, when the reset specified above is completed.	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. This bit is cleared when writing 1 to the *TR* bit. This bit has no function if on-chip memory is not implemented.	R	0

Table 15.20 TCBCONTROLB Register Field Descriptions*(continued)*

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>TM</td><td>13:12</td><td>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.
<table>
<tr><th>TM</th><th>Trace Mode</th></tr>
<tr><td>00</td><td>Trace-To</td></tr>
<tr><td>01</td><td>Trace-From</td></tr>
<tr><td>10</td><td>Reserved</td></tr>
<tr><td>11</td><td>Reserved</td></tr>
</table>
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 CPU.
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.
These bits have no function if on-chip memory is not implemented.</td><td>R/W</td><td>0</td></tr>
<tr><td>Reserved</td><td>11</td><td>Reserved. Must be written as zero; returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>CR</td><td>10:8</td><td>Off-chip Clock Ratio. Writing this field, sets the ratio of the CPU clock to the off-chip trace memory interface clock. The clock-ratio encoding is shown in Table 15.21.
Note: As the Probe interface works in double data rate (DDR) mode, a 1:2 ratio indicates one data packet sent per CPU clock rising edge.
These bits have no function if off-chip memory is not implemented.</td><td>R/W</td><td>100_2</td></tr>
</table>

Table 15.20 TCBCONTROLB Register Field Descriptions*(continued)*

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>Cal</td><td>7</td><td>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.
<table>
<tr><th rowspan="2"></th><th colspan="4">Calibrations pattern</th></tr>
<tr><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td rowspan="13">This pattern is replicated for every 4 bits of <i>TR_DATA</i> pins.</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>1</td></tr>
</table>
<b>Note:</b> The clock source of the TCB and PIB must be running.
These bits have no function if off-chip memory is not implemented.</td><td>R/W</td><td>0</td></tr>
<tr><td>Reserved</td><td>6:2</td><td>Reserved. Must be written as zero; returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>OfC</td><td>1</td><td>If set to 1, trace is sent to off-chip memory using <i>TR_DATA</i> 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).</td><td>R/W</td><td>Preset</td></tr>
<tr><td>EN</td><td>0</td><td>Funnel Trace Enable. When this bit is set, the trace funnels accepts trace information from the CM and/or cores and writes the information to off-chip or on-chip memory.
When this bit is cleared, the trace funnel drops all new trace information from the CM and/or cores . The trace information already accepted by the trace funnel is sent to the off-chip or on-chip memory, but new trace information is dropped and not written out.</td><td>R/W</td><td>0</td></tr>
</table>

The Probe Interface Block (PIB) has been an available component with many previous MIPS cores, including the proAptiv. The proAptiv Multiprocessing System brings two significant changes to the PIB. First, the PIB is now instantiated in mips_css. Second, this new version of the PIB, referred to as PIB2, provides additional clock ratios.

The PIB2 provides available TR_CLK to processor clock ratios of 1:2, 1:4, 1:6, 1:8, 1:10, 1:12, 1:16, and 1:20. The PIB1 supplied by MIPS has only the ratios 1:2, 1:4, 1:6, and 1:8. The PIB1 architecture also has provision for clock multiples, 1:1, 2:1, 4:1, and 8:1, but these are not supported in PIB2.

The PIB2 reports the minimum CR (TC_CRMin) as 3'b111 and maximum (TC_CRMax) as 3'b000 as shown in the table below. This is how software identifies a PIB2 as opposed to PIB.

Table 15.21 Clock Ratio Encoding of the CR field

TC_ClockRatio	TR_CLK : gclk
3'b000	1:20
3'b001	1:16
3'b010	1:12
3'b011	1:10
3'b100	1:2
3'b101	1:4
3'b110	1:6
3'b111	1:8

TCBDATA Register

The *TCBDATA* register (0x12) is used to access the registers defined by the *TCBCONTROLB*$_{REG}$ field; see Table 15.19. Regardless of which register or data entry is accessed through *TCBDATA*, the register is only written if the *TCBCONTROLB*$_{WR}$ bit is set. For read-only registers, *TCBCONTROLB*$_{WR}$ is a don't care.

The format of the *TCBDATA* register is shown below, and the field is described in Table 15.22. The width of *TCBDATA* is 64 bits when on-chip trace words (TWs) are accessed (*TCBTW* access).

Figure 15.7 TCBDATA Register Format

31(63)	0
Data	

Table 15.22 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

TCBCONTROLD Register

The TCB includes a second control register, TCBCONTROLD (EJTAG Register 0x14), whose values are used to control the tracing functions of the Coherence Manager. External software (i.e., debugger) can therefore manipulate the trace output by writing to this register. This register is also mapped to offset 0x0010 in the Global Debug Block of the CM GCRs.

The format of the *TCBCONTROLD* register is shown below, and the fields are described in Table 15.23

Figure 15.8 TCBCONTROLD Register Format.

31 26	25 24	23 22	21 20	19 18	17 16	15 12	9 8		6	5	4 3	2	1	0
Reserved	P4_Ctl	P3_Ctl	P2_Ctl	P1_Ctl	P0_Ctl	Reserved	TWSrcVal	WB	ST_En	IO	TLev	AE	Global_CM_En	CM_EN

Table 15.23 TCBCONTROLD Register Definition

Fields		Description	Read / Write	Reset State
Name	Bits			
Reserved	31:30	Reserved for future use. Must be written as 0.	R	0
P6_Ctl	29:28	Implementation specific finer grained control over tracing Port 6 traffic at the CM. See Table 15.24.	R/W	0
P5_Ctl	27:26	Implementation specific finer grained control over tracing Port 5 traffic at the CM. See Table 15.24.	R/W	0
P4_Ctl	25:24	Implementation specific finer grained control over tracing Port 4 traffic at the CM. See Table 15.24.	R/W	0
P3_Ctl	23:22	Implementation specific finer grained control over tracing Port 3 traffic at the CM. See Table 15.24.	R/W	0
P2_Ctl	21:20	Implementation specific finer grained control over tracing Port 2 traffic at the CM. See Table 15.24.	R/W	0
P1_Ctl	19:18	Implementation specific finer grained control over tracing Port 1 traffic at the CM. See Table 15.24.	R/W	0
P0_Ctl	17:16	Implementation specific finer grained control over tracing Port 0 traffic at the CM. See Table 15.24.	R/W	0
Reserved	15:12	Reserved for future use. Must be written as 0 and read as 0.	R	0
TWSrcVal	11:8	The source ID of the CM.	R/W	0
WB	7	When this bit is set, Coherent Writeback requests are traced. If this bit is not set, all Coherent Writeback requests are suppressed from the CM trace stream.	R/W	0
ST_En	6	System Trace Enable. Driven to the CM ouput pin *SI_TC_Sys_Enable*. External logic can use this output to control generation of the System Trace stream.	R/W	0
IO	5	Inhibit Overflow on CM FIFO full condition. When set to 1 the CM never drops trace words, but instead will stall the request and/or intervention processing until forward progress can be made. When set to 0 the CM will drop trace words when the trace word FIFO overflows.	R/W	0
TLev	4:3	This defines the current trace level being used by CM tracing **Encoding** / **Meaning** 00 / No Timing Information 01 / Include Stall Times, Causes 10 / Reserved 11 / Reserved	R/W	0

Fields		Description	Read / Write	Reset State
Name	Bits			
AE	2	When set to 1, address tracing is always enabled for the CM. This affects trace output from the serialization unit of the CM. When set to 0, address tracing may be enabled through the implementation specific P[x]_Ctl bits.	R/W	0
Global_CM_En	1	Each CPU core can enable or disable CM tracing using this bit. This bit is not routed through the master core, but is individually controlled by each core. Setting this bit can enable tracing from the CM even if tracing is being controlled through software, if all other enabling functions are true.	R/W	0
CM_EN	0	This is the master trace enable switch to the CM. When zero, tracing from the CM is always disabled. When set to one, tracing is enabled if other enabling functions are true.	R/W	0

Table 15.24 P<port>_Ctl Trace Control Bits

Value	Meaning
0	Tracing Enabled, No Address Tracing, assuming AE = 0
1	Tracing Enabled, Address Tracing Enabled, independent of AE
2	Reserved
3	Tracing Disabled

The *TCBCONTROLD.AE* bit enables addresses to be supplied when any request is serialized. This is not typically required because addresses issued from processor CPUs can be inferred from the CPU PDTrace stream.

The *TCBCONTROLB.TLev* bit controls the amount of information to be included the CM trace. Setting *TLev* to 1 may be useful when debugging performance problems.

The *TCBCONTROL.IO* bit determines the action taken by the CM with its internal trace buffers overflow. If the *IO* bit is 0 then trace information is lost when the trace buffer overflows. In this case, the CM temporarily stops producing trace messages, waits until the trace buffer becomes empty, performs a trace resynchronization with the CPUs and then starts producing new trace words.

However, if *TCBCONTROL.IO* bit is 1 then trace information is never lost, but the system performance may be impacted when the trace buffer becomes full and the additional trace words are required. In this case, the CM stalls the processing of requests and/or L1 intervention responses until a trace buffer becomes available.

The *TCBCONTROL.WB* determines if L1 writebacks are traced or not. L1 writebacks are not software visible and do not appear in the CPU PDTrace, so typically writebacks are not traced in the CM (WB set to 0).

The value in the *TCBCONTROLD.TWSrcVal* field appears in all trace words produced by the CM, thus tagging the trace word as coming from the CM. A unique value must be programmed in this field and *TCBCONTROLB.TWSrcVal* for all cores.

The five *P<port>_Ctl* fields in *TCBCONTROLD* give the ability to control the amount of trace information provided for requests received on the specified port. As shown in Table 15.24, requests from a given CM request port can be traced normally, always traced with addresses, or not traced. Typically, the CM request ports connected to CPUs will be traced normally (P0_Ctl, P1_Ctl, P3_Ctl, P4_Ctl set to 0) because the address is traced by the CPU itself. However, requests from the IOCU are only traced by the CM and therefore should have their addresses traced by the CM (P4_Ctl should be set to 2).

TCBCONTROLE Register

The *TCBCONTROLE* register is used top control tracing functions of the Coherence Manager performance counters. The *TCBCONTROLE* register is written by an EJTAG TAP controller instruction, *TCBCONTROLE* (0x16). This register is also mapped to offset 0x0020 in the Global Debug Block of the CM GCRs. The format of the *TCBCONTROLE* register is shown below, and the fields are described in Table 15.25.

Figure 15.9 TCBCONTROLE Register Format

31 9	8	7 1	0
0	TdIDLE	Res	PeC

Table 15.25 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. TrIdle is set when the all cores and the CM have disabled PDTrace and the trace funnels has written all outstanding trace information to the off-chip or on-chip memory.	R	1
0	7:1	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
PeC	0	Performance counter tracing is not implemented.	R	0

TCBCONFIG Register (Reg 0)

The *TCBCONFIG* register holds information about the hardware configuration of the TCB. This register is also mapped to offset 0x0028 in the Global Debug Block of the CM GCRs. The format of the *TCBCONFIG* register is shown below, and the field is described in Table 15.26.

Figure 15.10 TCBCONFIG Register Format

31	30 25	24 21	20 17	16 14	13 11	10 9	8 6	5	4	3 0
CF1	0	0	SZ	CRMax	CRMin	PW	PiN	OnT	OfT	REV

Table 15.26 TCBCONFIG Register Field Descriptions

<table>
<tr><th colspan="2">Fields</th><th rowspan="2">Description</th><th rowspan="2">Read / Write</th><th rowspan="2">Reset State</th></tr>
<tr><th>Name</th><th>Bits</th></tr>
<tr><td>CF1</td><td>31</td><td>This bit is set if a *TCBCONFIG1* register exists. In this revision, *TCBCONFIG1* does not exist and this bit always reads zero.</td><td>R</td><td>0</td></tr>
<tr><td>0</td><td>30:21</td><td>Reserved. Must be written as zero; returns zero on read.</td><td>R</td><td>0</td></tr>
<tr><td>SZ</td><td>20:17</td><td>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.</td><td>R</td><td>Preset</td></tr>
<tr><td>CRMax</td><td>16:14</td><td>Off-chip Maximum Clock Ratio.
This field indicates the maximum ratio of the CPU clock to the off-chip trace memory interface clock. The clock-ratio encoding is shown in Table 15.21.
This bit is reserved if off-chip trace option is not implemented.</td><td>R</td><td>Preset</td></tr>
<tr><td>CRMin</td><td>13:11</td><td>Off-chip Minimum Clock Ratio.
This field indicates the minimum ratio of the CPU clock to the off-chip trace memory interface clock.The clock-ratio encoding is shown in Table 15.21.
This bit is reserved if off-chip trace option is not implemented.</td><td>R</td><td>Preset</td></tr>
<tr><td>PW</td><td>10:9</td><td>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.
<table>
<tr><th>PW</th><th>Number of bits used on *TR_DATA*</th></tr>
<tr><td>00</td><td>4 bits</td></tr>
<tr><td>01</td><td>8 bits</td></tr>
<tr><td>10</td><td>16 bits</td></tr>
<tr><td>11</td><td>reserved</td></tr>
</table>
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.</td><td>R</td><td>Preset</td></tr>
<tr><td>PiN</td><td>8:6</td><td>Pipe number.
Indicates the number of execution pipelines.</td><td>R</td><td>0</td></tr>
<tr><td>OnT</td><td>5</td><td>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.</td><td>R</td><td>Preset</td></tr>
<tr><td>OfT</td><td>4</td><td>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).</td><td>R</td><td>Preset</td></tr>
<tr><td>REV</td><td>3:0</td><td>Revision of TCB.</td><td>R</td><td>0</td></tr>
</table>

TCBTW Register (Reg 4)

The *TCBTW* register is used to read Trace Words from the on-chip trace memory. The TW read is the one pointed to by the *TCBRDP* register. A side effect of reading the *TCBTW* register is that the *TCBRDP* register increments to the next TW in the on-chip trace memory. If *TCBRDP* is at the max size of the on-chip trace memory, the increment wraps back to address zero.

This register is also mapped to offset 0x0200 (lower 32 bits) and 0x0208 (upper 32 bits) in the Global Debug Block of the CM GCRs.

The format of the *TCBTW* register is shown below, and the field is described in Table 15.27.

Figure 15.11 TCBTW Register Format

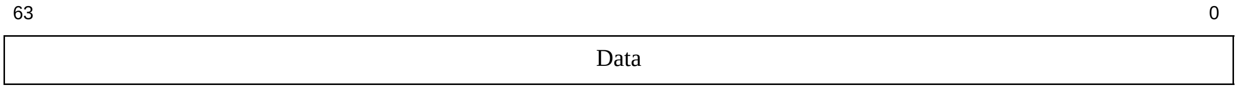


Table 15.27 TCBTW Register Field Descriptions

Fields		Description	Read / Write	Reset State
Names	Bits			
Data	63:0	Trace Word	R/W	0

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 also mapped to offset 0x0108 in the Global Debug Block of the CM GCRs.

The format of the *TCBRDP* register is shown below, and the field is described in Table 15.28. 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 15.12 TCBRDP Register Format

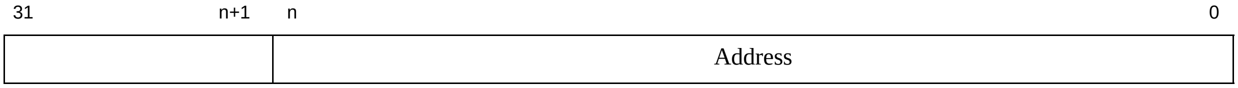


Table 15.28 TCBRDP Register Field Descriptions

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

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 also mapped to offset 0x0110 in the Global Debug Block of the CM GCRs.

The format of the *TCBWRP* register is shown below, and the fields are described in Table 15.29. 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 15.13 TCBWRP Register Format

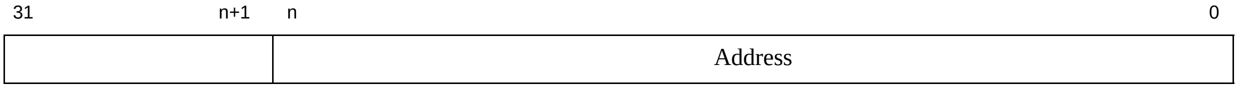


Table 15.29 TCBWRP Register Field Descriptions

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

TCBSTP Register (Reg 7)

The *TCBSTP* register is the start pointer register. This pointer is used to determine when all entries in the trace buffer have been filled (when *TCBWRP* has the same value as *TCBSTP*). 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 also mapped to offset 0x0118 in the Global Debug Block of the CM GCRs.

The format of the *TCBSTP* register is shown below, and the fields are described in Table 15.30. 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 15.14 TCBSTP Register Format

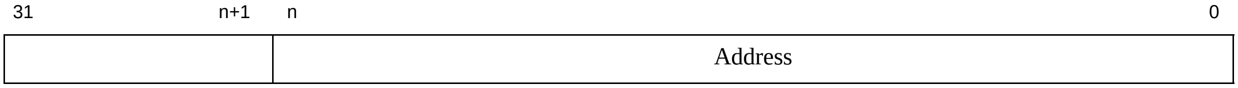


Table 15.30 TCBSTP Register Field Descriptions

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

TCBSYS Register (Reg 30)

The *TCBSYS* register contents are driven to the *SI_TC_Sys_UserCtl[31:0]* output signals. This register is also mapped to offset 0x0040 in the Global Debug Block of the CM GCRs. Thus, any change to this register will be reflected in these output signals. The format of the *TCBSYS* register is shown below, and the fields are described in Table 15.31.

Figure 15.15 TCBSYS Register Format

31	30	0
STA	UsrCtl	

Table 15.31 TCBSYS Register Field Descriptions

Fields			Read /	
Name	Bits	Description	Write	Reset State
STA	31	System Trace Available. Set to 1 if the System Trace Interface is present. Otherwise it is set to 0.	R	present: 1 not present: 0
UsrCtl	30:0	User-defined Control.	R/W	0

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.

15.3.4 MIPS Trace Capability

There are several build-time options for trace support within the proAptiv Multiprocessing System Multiprocessing System (MPS):

1. No trace logic included.

2. Trace logic to support an on-chip trace memory (embedded within the MPS).

3. Trace logic to support an off-chip trace probe (with off-chip trace memory).

4. Combination of options 2 and 3.

15.3.5 Memory-Mapped Access to PDtrace™ Control and On-Chip Trace RAM

PDtrace can be controlled entirely through software and the on-chip trace memory can be accessed directly by software using load and store instructions.

15.3.6 On-Chip Trace Buffer Usage

In order to direct trace data to the on-chip buffer instead of the off-chip interface, the OfC bit in the TCBControlB register of the trace master must be cleared. Once this is done, the trace funnel will combine trace data it receives from the CM and CPUs and write it to the on-chip memory. Tracing can be enabled or disabled on a per CM/CPU basis by setting or clearing the EN bits in the corresponding TCBControlB registers.

To initialize the on-chip trace buffer, the TR bit of the TCBControlB register of the trace master is set by software. This will initialize TCBRDP, TCBWRP and TCBSTP pointers to zero. These pointers do not have to explicitly written by software for initialization, the reset function that is caused by setting the TR bit is sufficient.

When it is desired to read out the Trace Words from the on-chip buffer, software first sets the RM bit.within TCBControlB. This will load the TCBRDP register with the value held in the TCBSTP register. The TraceWord pointed by TCBRDP can be then read out through the TCBTW register. The read will automatically update the TCBRDP value to point to the next newer entry. A subsequent read from TCBTW register will thus read out the next newer TraceWord. Software does not have to explicitly update the TCBRDP register.

If the TM field of TCBControlB register is set to Trace-From mode, the trace-buffer contents stop being updated when the trace-buffer is full (when TCBWRP points to the same entry as TCBSTP). This event is denoted by the BF bit of TCBControlB register. The BF bit can be polled by software to decide when to read out the trace buffer contents.

For production testing, such as stuck-at testing of memory cells within the trace buffer, the TCBRDP and TCBWRP registers can be explicitly written by software to write and read specific entries within the trace buffer. As previously stated, for normal usage these pointer registers do not have to be explicitly written by software.

Chapter 16

o32 Binary Interface and General Purpose Registers

The MIPS32 ABI provide the application binary interface to the proAptiv Multiprocessing System Multiprocessing System. The 32 general purpose registers (GPR) are used by the compiler and provide a set of temporary storage locations used for function calls and other software functions. Some register have unique functions, while others can be used for any purpose.

This chapter contains the following sections:

16.1 o32 ABI Parameters

This chapter contains information on the o32 binary interface. The o32 ABI contains calling and linkage conventions for the MIPS32 architecture.

Table 16.1 shows the o32 ABI parameters.

Table 16.1 MIPS o32 ABI Parameters

Parameter	Value
Registers saved and restored as	32-bit
Argument structure	4-byte slots 8-byte alignment at least 4 slots long
Argument registers	4 integer, 2 FP
Arguments in FP registers?	Leading FP arguments only (doesn’t need correct function prototypes)
Return values	Only scalars are ever returned in registers; v1 is used only for `long long` data.
`long long` type	Implemented with register pairs and hardware type library calls

Table 16.1 MIPS o32 ABI Parameters

Parameter	Value
gp register in PIC code	Not preserved over calls

16.2 Compiler Goals

Compilation systems should adhere to following standards (‘‘ABIs’’) in order to achieve the following goals:

- **Inter-calling** : a binary program built with one compiler should be able to call a subroutine defined in another (so long as address resolution problems are solved). The standards relevant to this are called the ‘‘calling conventions’’; they describe how subroutines pass parameters, return values, and co-operate to share the register set and stack resources. Refer to Section 16.8 “Calling Conventions”.
- **Interlinkable**: object files built with one compiler can be linked successfully with those produced by another. The standard relevant to this is the object code definition, in particular the definition of symbols and relocation mechanisms.
- **Runnable**: a binary produced with a compliant tool kit can be successfully executed on a compliant OS (Linux in particular). For more information, refer to Section 16.6.3 “Linux Application”.
- **Debuggable**: more conventions and standards are required before a program build with a tool kit can be successfully debugged.
- **Profilable**: where available, code profilers have their own requirements - related to but not identical to those of debuggers.

16.3 Register Naming Conventions and Usage

Each proAptiv Multiprocessing System core offers 32 general purpose registers available for program use. They are numbered $0 to $31.

Two, of these registers behave as follows::

- $0 always returns zero, no matter what is stored in it.
- $31 is always used by the normal subroutine-calling instruction (jal) for the return address. Note that the call-by register version (jalr) can use any register for the return address, though use of anything except $31 is not recommended .

In all other respects the general purpose registers are identical and can be used in any instruction (it is legal to use $0 as the destination of instructions, though the result data will not be saved).

The return address of a **jal** is the next instruction, but one in sequence:

```
...
jal printf
move $4, $6
xxx # return here after call
```

The floating point math coprocessor (called FPA for floating point accelerator), if included, adds 32 floating point registers with their own conventions: see Section 16.5 “Floating Point Register Conventions”.

16.4 Conventional Naming and General Purpose Register Usage

Although the hardware makes few rules about the use of registers, their practical use is governed by a number of conventions. As part of those conventions, the registers are referred to by conventional names — typically defined in a header file2 and implemented by using the C preprocessor on assembler files.

MIPS hardware ignores these conventions, but all the benefits of software by register name and associated common name. These common name functions are described in the following subsections.

Table 16.2 CP0 Registers Grouped by Number

Register Name	Common Name	Description
$0	zero	Always has the value 0. Any writes to this register are ignored.
$1	AT	Assembler temporary.
$2	v0	Function result register. Functions return integer results in v0, and 64-bit integer results in v0 and v1 when using 32-bit registers. In cases where floating-point hardware is not present, or when compiler options enable floating-point emulation, functions return single precision floating-point results in v0 and double precision floating-point results in v0 and v1 when using 32-bit registers. v0 and v1 can be temporary registers. Not preserved across function calls.
$3	v1	
$4	a0	Function argument registers that hold the first four words of integer type arguments. Functions use these registers to hold floating-point arguments. When floating-point hardware is not present, or compiler options enable floating-point emulation, functions use a0 to hold the first single precision floating-point argument and a1 to hold the second single precision floating-point argument. Functions use a0-a1 for the first double precision floating-point argument, and a2-a3 to hold the second double precision floating-point argument. Not preserved across function calls.
$5	a1	
$6	a2	
$7	a3	
$8	t0	Temporary registers. Can be used for any puepose. Not preserved across function calls.
$9	t1	
$10	t2	
$11	t3	
$12	t4	
$13	t5	
$14	t6	
$15	t7	
$24	t8	
$25	t9	
$16	s0	Saved registers to use freely. Preserved across function calls. This register must be saved before use by the called function.
$17	s1	
$18	s2	
$19	s3	
$20	s4	
$21	s5	
$22	s6	
$23	s7	
$30	s8	

Table 16.2 CP0 Registers Grouped by Number *(continued)*

Register Name	Common Name	Description
$26	k0	Reserved for use by the operating system kernel and for exception return.
$27	k1	
$28	gp	Global pointer. May be used as save register for called functions.
$29	sp	Stack pointer.
$31	ra	Return address register, saved by the calling function. Available for use after saving.
$f0		Function return register used to return float and double values from function calls.
($f12, $f13) and ($f14 and $f15)		Two pairs of registers used to pass float and double valued parameters to functions. Pairs of registers are parenthesized because they have to pass double values. To pass float values, only $f12 and $f14 are used.

16.4.1 Common Name — AT (GPR $1)

GPR $1 is reserved for the synthetic instructions generated by the assembler. When using this register, such as when saving or restoring registers in an exception handler, there is an assembler directive to stop the assembler from using it without permission (but then some of the assembler’s macro instructions won’t be available.) The assembler directive’s existence is the reason why this name is traditionally used in upper case.

16.4.2 Common Name — v0, v1 (GPR $2, $3)

GPR $2 and $3 are used when returning non-floating-point values from a subroutine. If the value returned is too large to fit in two registers, the compiler will allocate a memory buffer whose address will be passed as an invisible first argument.

While a function is running v0-v1 can be freely used as temporaries. Integer values are returned in these registers. Structure or array types (even if small enough to fit in the two registers) are always returned through a data area defined by the caller and whose address is an invisible first argument.

16.4.3 Common Name — a0 - a3 (GPR $4 - $7)

GPR $4 - $7 are used to pass the first four non-FP parameters to a subroutine. Argument registers which are unused or whose value is no longer needed can be freely used as temporaries. For more information, refer to Section 16.8 “Calling Conventions”

16.4.4 Common Name — t0 - t9 (GPR $8 - $15, GPR $24 - $25

By convention, subroutines may use these registers without doing anything to preserve their previous contents. This makes them a good choice for ‘‘temporaries’’ when evaluating expressions. However, the compiler/programmer must remember that values stored in them may be destroyed by a subroutine call.

16.4.5 Common Name — s0 - s8 (GPR $16 - $23, GPR $30)

By convention, subroutines must guarantee that the values of these registers on exit are the same as they were on entry. This can be accomplished by either not using them, or by saving them on the stack and restoring before exit.

This makes them eminently suitable for use as ‘‘register variables’’ or for storing any value which must be preserved over a subroutine call.

16.4.6 Common Name — k0, k1 (GPR $26, $27)

GPR $26 and GPR $27 are reserved for use by the trap/interrupt handlers in an operating system, which uses them and does not restore their original value. Hence they are of little use to other code. These registers are not used at all by application code.

16.4.7 Common Name — gp (GPR $28)

GPR $28 has two quite different roles. These registers are used in position-independent (PIC) code and non-PIC code. Each case is described below.

16.4.7.1 Position Independent Code (PIC)

In PIC code, typically used only for application and library code in a large OS, it is used in the double-indirection used to reach variables and functions whose location is not known until the program and its libraries are loaded.

In position-independent code, *gp* acts as a pointer to the GOT (‘‘global offset table’’), as described in Section 16.6.4 “PIC Code and the Global Offset Table”. The GOT pointer is loaded by code in the prologue of every function which makes a reference through the GOT. A function call may overwrite the value in *gp* and the compiler must ensure it’s reloaded after any such call.

16.4.7.2 Non-Position Independent Code (Non-PIC)

In non-PIC code, typically used for all non-Linux embedded applications, this register is sometimes used to provide efficient access to C static/extern data.

If used, the *gp* register is initialized to point to a load-time-determined location of the static data. This means that loads and stores to data lying within 32 KBytes on either side of the *gp* value can be performed in a single instruction using *gp* as the base register. Note that the pointer in *gp* is a constant. No application code ever writes to the register once it has been initialised.

Without the global pointer, loading data from a static memory area takes two instructions: one to load the most significant bits of the 32-bit constant address computed by the compiler and loader, and one to do the data load.

To use *gp*, a compiler must know at compile time that a datum will end up linked within a 64 KByte range of memory locations. In practice this cannot be known, only estimated. The usual practice is to put ‘‘small’’ global data items (8 bytes and less in size) in the *gp* area, and to get the linker to complain if it still gets too big. The compiler -Gnn flag can be used to adjust the threshold of what is considered ‘‘small’’.

16.4.8 Common Name — sp (GPR #29)

The GPR #29 register functions as a stack pointer. It takes explicit instructions to raise and lower the stack pointer, so MIPS code usually adjusts the stack only on subroutine entry and exit; and it is the responsibility of the subroutine being called to do this.

sp is normally adjusted, on entry, to the lowest point that the stack will need to reach at any point in the subroutine. Now the compiler can access stack variables by a constant offset from sp.

16.4.9 Common Name — fp (GPR #30)

A subroutine will use a ‘‘frame pointer’’ to keep track of the stack if it wants to do things which involve extending the stack by an amount which is determined at run-time. Some languages, including C++, may do this implicitly.

If the stack bottom can’t be computed at compile time, the stack variables cannot be accessed from sp, so fp is initialized by the function prologue to a constant position relative to the function’s stack frame. Efficient use of register conventions means that this behavior is local to the function, and doesn’t affect either the calling code, or any nested function calls.

16.4.10 Common Name — ra (GPR #31)

The GPR #30 register is used to store the return address. On entry to any subroutine, ra holds the address to which control should be returned. Therefore, a subroutine typically ends with the instruction: jr ra. Subroutines which themselves call subroutines must first save ra, usually on the stack.

16.5 Floating Point Register Conventions

In addition to the 31 general purpose registers, the MIPS32 architecture also includes a corresponding set of standard floating point registers.

16.5.1 MIPS Floating Point Registers

The proAptiv Multiprocessing System architecture contains 32 floating point registers, whose assembler names are $f0 - $f31. The architecture also supports the 64-bit IEEE double-precision format.

The o32 ABI generates code that is compatible with not only previous generation 32-bit MIPS I and MIPS II CPUs, but also the MIPS III CPU’s such as the proAptiv Multiprocessing System Multiprocessing System. All of these architectures do floating point arithmetic using the 16 even-numbered registers $f0 - $f30.

The odd-numbered registers are referred to in move and load/store instructions; but the assembler provides synthetic ‘‘macro’’ instructions for move and load/store double, so the odd-numbered registers will normally not be required when writing o32 code.

16.5.2 Floating Point Register Software Use and Calling Conventions

Like the general-purpose registers, the MIPS calling conventions contains restrictions about register use that have nothing to do with the hardware. These restrictions include which FP registers are used for passing arguments, which

ones’ values are expected to be preserved over function calls, and so on. The division of functions is much as for the integer registers, less the special cases.

Table 16.3 shows the floating point register usage conventions. Note that the o32 ABI assumes that the CPU either has a MIPS II or earlier FP unit or that the CPU has the SR[FR] compatibility bit cleared to zero; in either case only 16 registers are usable for arithmetic, so there are no odd-numbered registers in the table.

Table 16.3 Floating Point Register Usage Conventions

Parameter	Value
Function return values	$f0, $f2

Table 16.3 Floating Point Register Usage Conventions

Parameter	Value
Argument registers	$f12, $f14
Saved over function call (suitable for register variables)	Events $f20 - $f30
Temporaries (not saved over function call, or “caller-saved”	Evens $f4 - $f10, $f16, $f18

16.6 Virtual Memory Layout Overview

Although this section is not technically part of the ABI definition, it is useful to examine the interaction between the various pieces of code and data which might make up an ABI-compliant application. The following subsections describe some aspects of how the virtual address may be used.

16.6.1 Memory Regions and Object Code Naming Conventions

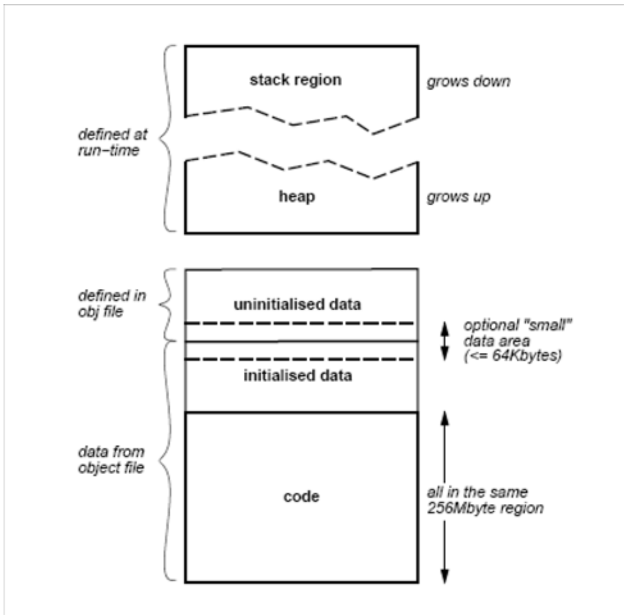
Some of the memory region and object code naming conventions are as follows:

- **Module**: A compilation unit (the assembler is seen as just another compiler...), and also used for an object file generated from one compilation unit.
- **Program**: all the addressable data and code associated with an application. Strictly speaking, that associated with an instance of an application; in Linux there may be many copies of the shell running, and they’re distinct programs in this sense. For Linux applications, this excludes the kernel and other parts of the memory map which are not accessible to the application.
- **Link unit**: a part of a program which has been bound together so that its components are at fixed offsets from each other.
- **Segment3**: a part of a program which is contiguous in the memory image of the running program, and which is distinguished for link/build purposes. By ancient convention segment names begin with a dot, and are called things like .text and .bss. When several modules are being combined into a single link unit during the build process, sections of the same name in different modules are brought together and various sections concatenated to make a segment.
- **_main4**: C programmers often think execution begins with main(); but in reality there’s always a more primitive, machine-dependent startup routine supplied by the build environment. This does things like initializing the *sp* register to mark the stack region, and zero-ing the memory region which contains the ‘‘uninitialised’’ C variables. If you write C++, this will also arrange for initialization routines to be run.

16.6.2 Simple Standalone Application

Figure 16.1 describes the memory map of a statically linked application.

Figure 16.1 Memory Map of a Statically Linked Application



The memory segments used in Figure 16.1 are described below.

- **Code** (including initialiaed and uninitialiaed data): This space forms a single link unit; and their relative positions are fixed when the software is built. In fact, their absolute locations in program memory are also typically fixed at build-time. This code is position-dependent.
- **Stack**: The stack is assigned by the start-up program in accordance with OS and tool chain conventions. It grows down, so is typically placed at the top of the program's memory space.
- **Heap**: The heap defines the data space allocated by the program through C setbrk() or (slightly higher level) malloc() calls. The heap usually starts at the lowest suitably aligned location available after allowing for the linked code and data.
- **Small data area**: For this space, it takes two MIPS instructions to load from or store to a C location declared at module level or as static. When the ''small'' data area is used, the gp register is set to point to the middle of it by __main(). This allows load and store to variables in that area to be accessed with a single instruction.

  Note that for some programs, the entire data set will not fit within the 64 KByte address range limit imposed by the MIPS load/store instruction's 16-bit offset. Therefore, during compilation and build, only data items below a certain size are considered as candidates for this area. That's why it's called ''small'' data. The small data area, if provided, overlaps both the initialized and uninitialized segments (and is implemented as a pair of sub-segments).
- **Common segment names**:

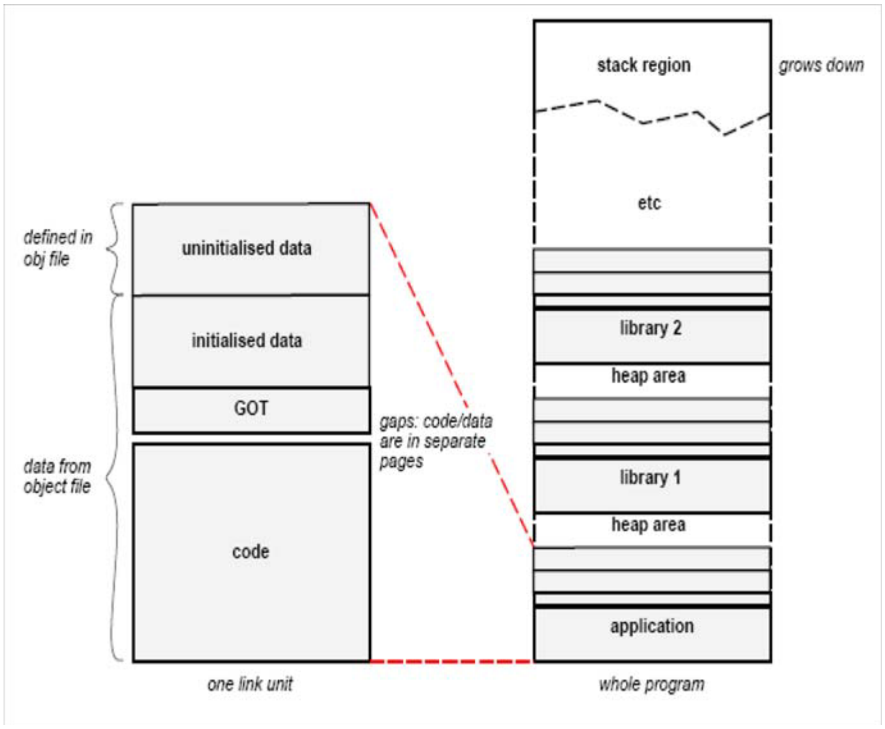
  .text all the code
  .data initialised data possibly excluding...
  .sdata initialised data for the ''small'' area
  .bss uninitialised data possibly excluding
  .sbss uninitialised data for the ''small'' area.

Note that in some embedded operating systems, all the software runs in the same address space and does not take advantage of the MIPS memory management facilities. Such systems have complicated memory maps which are deeply OS dependent.

16.6.3 Linux Application

Many Linux applications are built without their library functions. In this case, the library routines are linked in as the program is loaded into memory. The library routine may have been updated since the application was built, and it should still work. The result is a much more complicated memory map with a number (perhaps quite a large number) of separately linked pieces. This complexity if illustrated in Figure 16.2.

Figure 16.2 Memory Map for a Typical Linux Application



In the memory map shown in Figure 16.2 above, all the link units except the base application are shared libraries of some kind, either built-in shared libraries or dynamically loaded by explicit programming. They are loaded into program memory working upward on a first-come first-served basis.

While the base application runs at program addresses which were known at build time, the libraries must be able to run at arbitrary memory addresses. The requirement that library modules should link in just anywhere and still work (“position-independent code” or PIC) forces considerable changes to the way code is generated.

In the MIPS architecture, the preferred subroutine call instruction is `jal`, and that instruction is not PC-relative; it encodes (most of) the absolute virtual address of the subroutine entry point. Moreover, Linux standards require that the shared libraries should also be able to share extern data.

16.6.4 PIC Code and the Global Offset Table

An application's binary code is built before it is known where data or subroutines in other link units will reside in the program's memory map. Both the absolute and relative position of each link unit depends on what versions of what libraries get loaded in what order. It is not possible for the run-time loader to fix up these addresses in the code itself, because the code itself must be shared between different instances of the application program (and each instance may have a different library layout). As a result, an application's or library's binary will contain the address information needed to reference functions and data in a different link unit.

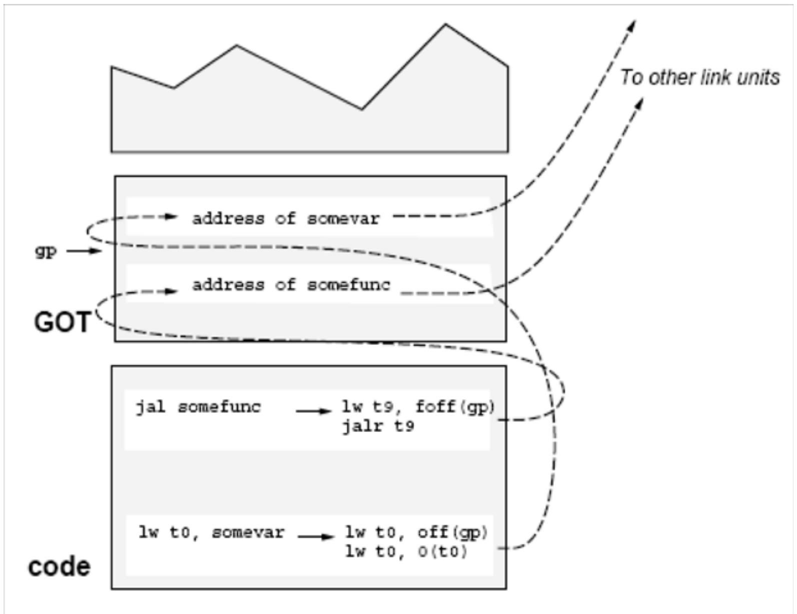
16.6.4.1 Global Offset Table (GOT)

Instead, the compiler generates code which makes every function call and every reference to static/extern data indirect, via a table of pointers. The table of pointers are the Global Offset Table or ‘‘GOT’, which reside in a data segment. Separate copies are kept for each instance of the application, so it can be and is fixed up by the loader.

Figure 16.3 shows an example of a global offset table. The GOT contains an entry for each function or data item that is accessed by any code in the link unit (the loader finds each item by its name, so there is an entry for each symbol). The table offset for a particular symbol is known at build time, and is a constant in the binary code.

For MIPS code, the *gp* register is maintained as a pointer to the GOT of the link unit.

Figure 16.3 Example of a Global Offset Table



The the *gp* register is set to point to the GOT by code included as part of the prologue of each function (at least, each function which makes any use of the GOT.) This is suboptimal, since for intra-link-unit calls it will already hold the right value. However, the compiler cannot (in general) distinguish intra- and inter-link-unit calls.

In the o32 ABI, the calling code must be aware that a function call might overwrite the value in the *gp* register, and the caller must preserve or recalculate the value after the call if required.

16.6.4.2 Loading a PIC Application and its Libraries

The program loader is the Linux application, which runs when any binary is loaded which uses shared libraries. The program loader maps the application code and data and any libraries it needs into the program's address space. The build system leaves a list of required library names in the application's object file, and the program loader finds the library files via a series of search path mechanisms. Conventions about file names (if followed correctly) make sure the program finds the ‘‘right’’ library.

The program loader maintains symbol tables for the data items and subroutine entry points which are exported by the application and each library, so it can tie up references between separate link units.

16.6.4.3 Loading and Binding of Libraries

While it is not necessary to read in all the code of the libraries required by an application (the ordinary virtual memory paging system takes care of that), the process of binding in a link unit, fixing up its GOT and getting it ready for use, is relatively time-consuming. This penalty is paid even for libraries which provide facilities which the application rarely uses. That can slow the application startup.

To optimize the initialization process, Linux defers loading and fixing up libraries until they are first used. By the nature of the PIC code the unresolved references are all in the GOT. Where the first reference to the new library is a function call this is relatively straightforward; the GOT entry for an unresolved subroutine reference is set to point to a function in the run-time loader which then loads the library, patches the GOT so that future calls will go direct, and calls the library function.

There are other more subtle issues. For example, when the same symbol is provided by two different libraries, this can make loading problematic. As a result, the build system is charged with identifying which libraries are safe to load, and to identify them in the application binary. The loader can then load unsafe libraries at startup.

16.6.4.4 Dynamic (explicit) Loading of Libraries - dlopen()

It is also possible to get software to pick its own shared library and then build an explicit software-visible table of calls to it. This mechanism fits naturally onto the object/class concepts of C++, and libraries loaded like this are referred to as ‘‘dynamic shared objects’’.

It is not necessary to build a Linux shared library in a special way to make it fit for dlopen(), any library will do. At the lowest level, a call to *dlopen()* to grab the library and *dlsym()* calls to obtain pointers to named data items or functions in the dynamic shared object. But because dynamic libraries are just shared libraries, some unexpected ‘‘bonus’’ semantics may be observed.

Firstly, the explicitly-loaded library will gain access to any public symbols in the application (or its pre-loaded libraries). Perhaps more unexpectedly, a straightforward extern function pointer reference in the application can bind to a symbol from a library which wasn't mentioned at all at build time, but only brought in with dlopen().

16.6.4.5 PIC and GOT Constraints

There are some PIC and GOT constrains that are worth mentioning.

1. **What to do when your GOT overflows**: On MIPS, GOT pointer loads are usually compiled to a single load relative to the *gp* register; but this can only span a table 64 Kbytes in size (16K pointer entries). Large applications and libraries can use more symbols than that.

   There are two approaches. One is to just let the GOT grow above 64 Kbytes, and require the compiler to generate code which can load/store arbitrary entries in it. This generally uses the gcc -PIC option - it's trouble-free and portable but generates truly awful code.

Some compilers support an option that generates one GOT to each module in a link unit (gcc -multigot). Done properly, this is no trouble, but the dynamic loader has to know about it.

2. **Managing the overheads of PIC code**: Nothing in the ABI obliges the compiler to go through the GOT when accessing data or calling subroutines which are in the same link unit; neither is it strictly necessary for a function to reset the gp register on an intra-link-unit call.

   However, there are several reasons why this hasn't been done:

   - Even within the link unit only relative addresses are known; the MIPS architecture lacks efficient PCrelative call and load instructions.

   - The compiler doesn't know which references are in the link unit. While it's possible to get the linker to do some instruction re-writing to simplify intra-link-unit calls and references, it's bad practice.

   - The PIC calling convention for MIPS requires that on entry to a function the *t9* register holds the address of the function's entry point. Since calls made through the GOT mean the address may be in some register, this seems acceptible — but this requirement is burdensome to any possible future intra-link-unit (or even intra-module) call mechanism.

16.7 Mapping Data Types into Memory

Memory is taken as an array of unsigned 8-bit quantities, whose index is the virtual address. For all MIPS CPU's, this corresponds to a C definition unsigned **`char []`**.

MIPS uses 2s-complement representation for signed integers - so in any data size "-1" is represented by binary all-ones. The overwhelming advantage of 2s-complement numbers is that the basic arithmetic operations (add, subtract, multiply, divide) have the same implementation for signed and unsigned data types.

16.7.1 Sizes of C Data Types

Table 16.4 lists fundamental C data types and how they're implemented for MIPS architecture CPUs.

Table 16.4 Data Types and Memory Representations

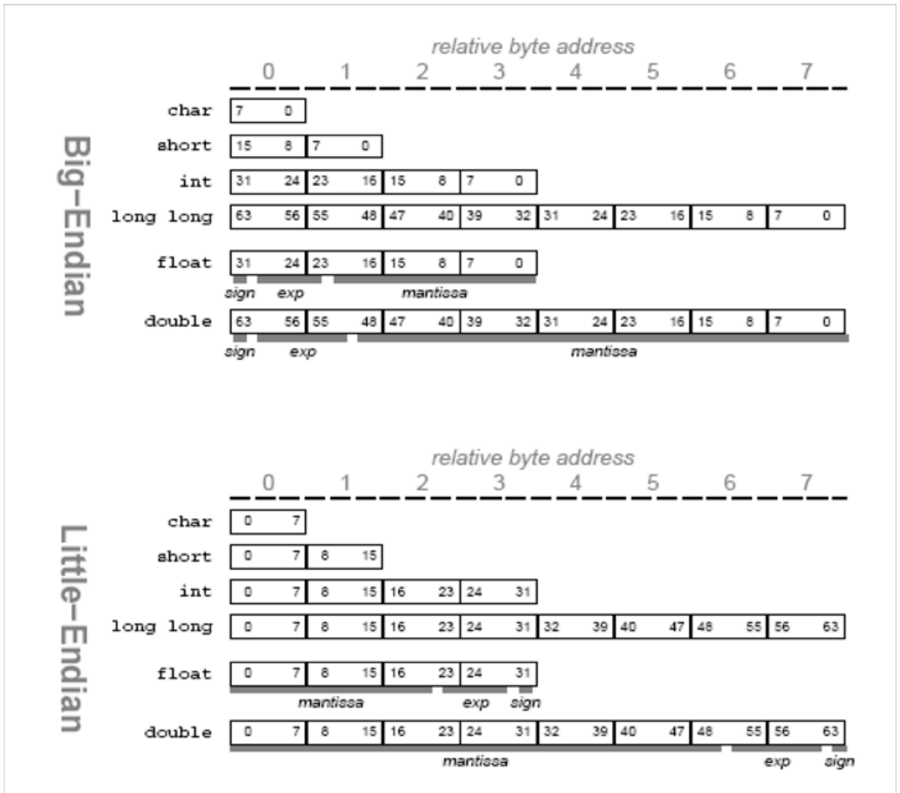
C Type	MIPS ASM Name	Size in Bytes
char	byte	1
short	half	2
int	word	4
long long	dword	8
float	word	4
double	dword	8

Note that all of the primitive data types shown in Table 16.4 can only be directly handled by standard MIPS instructions if they are naturally aligned: that is, a 2-byte datum starts at an address which is even (zero modulo 2), a 4-byte datum starts at an address which is zero modulo 4, and an 8-byte datum starts at an address which is zero modulo 8.

16.7.2 Basic Type Memory Layout and Endianness

Figure 16.4 shows how each basic type is laid out in byte-addressed memory. Note that the arrangement is different for bigendian and little-endian software.

Figure 16.4 C Data Types in Memory — Big Endian and Little Endian



In Figure 16.4, I've given in to the temptation to reverse the bit-numbering within each byte for the big-endian layouts. For memory addressing purposes this is meaningless; bytes are indivisible 8-bit objects. However, reversing the bit numbers as above makes the bitwise depiction of the fields of floating point numbers easier to absorb (and prettier). Each of these data types is naturally aligned, as described above. ‘‘Endianness’’ can be a troubling subject. If you are uneasy about it, read it up in [SMR].

16.7.3 Memory Layout and Alignment of Structure and Array Types

Complex types are built by concatenating simple types, but inserting unused (‘‘padding’’) bytes between items so as to respect the alignment rules.

The following shows the byte offsets of data items in a struct mixed:

```
struct mixed {
        char c; /* byte 0 */
        /* bytes 1-14 are ‘‘padding’’ */
        double d; /* bytes 8-15 */
        short s; /* bytes 16-17 */
};
```

Note that the byte offsets of the fields of constructed data types (other than those using C bitfields, see Section 16.7.3 "Memory Layout and Alignment of Structure and Array Types" below) are unaffected by endianness.

Constructed data types are aligned in memory to the largest alignment boundary required by the data type defined inside them. So a struct mixed will start on an 8-byte boundary; which means that if an array is built using these structures, padding will be required between each array element. C compilers provide for this by ‘‘tail padding’’ the structure to make it usable for an array, so sizeof(struct mixed) == 24 and the structure should reallybe annotated:

```
struct mixed {
        char c; /* byte 0 */
        /* bytes 1-14 are ‘‘padding’’ */
        double d; /* bytes 8-15 */
        short s; /* bytes 16-17 */
        /* bytes 18-23 are ‘‘tail padding’’ */
};
```

16.7.4 Bit Fields in Structures

C allows the user to define structures which pack several short ‘‘bit field’’ members into one or more locations of a standard integer type. This is a useful feature for emulation, hardware interfacing, and perhaps for defining dense data structures, but is fairly incomplete. Bitfield definitions are nominally CPU-dependent but also genuinely endianness-dependent.

One can, for example, define a data structure which permits access to the various fields of a MIPS single-precision-floating point number as shown below:

```
#if BYTE_ORDER == BIG_ENDIAN

struct ifloat {
        unsigned int sign:1;
        unsigned int bexp:8;
        unsigned int mant:23;
};

#else /* little-endian */

struct ifloat {
        unsigned int mant:23;
        unsigned int bexp:8;
        unsigned int sign:1;
};

#endif
```

In this case the three fields are packed into one 32-bit int storage unit. These two cases are the same in that, for both Endian formats, the bitfields are allocated with the first-defined field occupying the lowest byte-addressed part of the int.

These two examples differ as follows: For big-endian, the high-order bits are occupied first. For little-endian, the low-order bits are occupied first.

If one tries to implement bitfields in a less endianness-dependent way, then in the following example struct fourbytes would have a different memory layout from struct fouroctets as shown:

```
struct fourbytes {
            signed char a; signed char b; signed char c; signed char d;
}

struct fouroctets {
            int a:8; int b:8; int c:8; int d:8;
}
```

A field can only be packed inside one storage unit of its defined type. When trying to define a structure for a MIPS double-precision floating point number, the mantissa field contains part of two 32-bit int storage units and can't be defined in one attemp. The best that can be done is something similar to the following:

```
struct ieee754dp_konst {
            unsigned sign:1;
            unsigned bexp:11;
            unsigned manthi:20; /* cannot get 52 bits into... */
            unsigned mantlo:32; /* .. a regular C bitfield */
};
```

16.7.5 Alignment Rules

The full alignment rules for bit-fields are as follows:

- A bit-field must reside entirely in a storage unit that is appropriate for its declared type. Thus a bit-field never crosses its unit boundary.
- Bit-fields can share a storage unit with other struct/union members, including members that are not bit-fields (to pack together, the adjacent structure member must be of a smaller integer type).
- Structures generally inherit their own alignment requirement from the alignment requirement of their most demanding type. Named bit-fields will cause the structure to be aligned (at least) as well as the type requires.

  Unnamed fields - regardless of their defined type - only force the storage unit or overall structure alignment to that of the smallest integer type which can accommodate that many bits.
- It may be adventageous to force subsequent structure members to occupy a new storage unit. In some compilers this can be done with an unnamed zero-width field. Zero-width fields are otherwise illegal.

16.8 Calling Conventions

The calling convention describes how arguments are passed to functions, and how values are returned. It's also a convenient place to describe the stack frame structure which builds up to represent the current function nest.

ANSI C permits pretty much any value - structures and arrays as well as scalars - to be passed as arguments or returned by a function.

16.8.1 Stack Maintenance and Alignment

When the stack is adjusted by functions to make space for local variables, register saves and argument passing, it is always adjusted by a multiple of 8 bytes so that the stack base is aligned to the greatest extent required by any variable.

16.8.2 Registers and the Argument Structure

For efficiency, it is recommended to pass arguments in registers and avoid data loads/stores. But C permits pretty much any non-array data type - no matter how large or complex - to be passed as an argument. It is not‘‘obvious’’ how such arguments should be passed. To make sure the corner cases are handled correctly, the set of arguments passed to a function is mapped as it would be to a memory-based argument structure, and then as much of that structure as will fit is pasted into the available registers. For any arguments left over after all available argument registers have been used up, a copy of that part of the argument structure is placed onto the stack.

The rules are as follows:

1. Each argument is aligned to the start of a new argument slot within the argument structure. These slots are 4 bytes in size, chosen to match the size of the general-purpose registers. If the next slot doesn’t have the correct alignment for a value (for example, a double on o32 requires 8-byte alignment), it is skipped to find a slot which is correctly aligned. Skipped slots remain unused. Large arguments may spill over into more than one slot.

2. Integer values are first converted to the type of the argument (if there is a function prototype) using standard C rules. Where there is no function prototype, the rules are that integer and floating point values are coerced to signed int and double respectively.

3. Integers smaller than int are expanded to int by zero- or sign-extending them in accordance with C rules.

4. Non-integer arguments smaller than a register-sized slot are aligned to the lowest addressed part of the slot.

5. Float arguments are 8-byte aligned and occupy two slots (even though there’s nothing useful in the second four bytes).

6. The argument registers are identified with a particular slot in the argument structure. If for alignment or other reasons a slot cannot be used, then the corresponding register won’t be used to pass an argument.

7. The caller will always build an argument data structure, even though it may remain unused in whole or part. Moreover, the data structure is always a minimum of 16 bytes (four register-sized slots) in size.

8. The first 4 x register-sized (ie 4 byte) slots of the structure are mapped to registers a0-3.

9. o32 does not assume the existence of function prototypes. For reasons to do with the implementation of functions with variable numbers of arguments, it is difficult to ensure that the caller and the called function always agree when to use a floating point rather than a general-purpose register for an argument.

   o32’s rule is that up to two leading floating point arguments will be passed in FP registers, but if the first argument is not an FP a second FP argument will not be put in an FP register. In functions like printf() the first argument is a pointer, so floating point values will be passed in integer registers or on the stack.

16.8.3 Returning Values from a Function

In the o32 ABI, a simple scalar value is returned in a register; v0 for integers, and fv0 for floating point values. A second integer register is defined for returning larger values, and is used when returning a long long value in o32.

For all other structures or larger values which are not accommodated in the registers, the caller must provide a pointer to a memory buffer (usually on the stack, but that’s not mandatory). The caller prepends a pointer to the memory buffer as an implicit first argument, followed by its explicit arguments. The called function should copy the return value to the supplied address.

16.8.4 Calling Conventions Extended for Linux ("MIPS ABI") PIC Code.

In PIC code functions are not called directly; instead the compiler/assembler generate code which loads the function address from the GOT table (see Section 16.6.4 "PIC Code and the Global Offset Table" above). The disassembled code looks something like this:

```
/* (caller) */
        lw t9, <function symbol offset in GOT>(gp)
        # nop
        jalr t9
        # nop
        ...

/* function */
        /* _gp_disp is magic symbol for offset between start of
        function and gp pointer into GOT */
        li gp, _gp_disp
        addu gp, gp, t9
        ...
```

It's mandatory that the *t9* register should be used to compute the function address; the function itself depends on it to recalculate the GOT base register *gp*. _gp_disp is calculated so as to place gp 32 KBytes on from the start of the GOT, to maximise the amount of the table which is in reach of a MIPS load instruction (which has a ±32K offset range).

Chapter 17

Instruction Set Overview

This chapter provides an overview of the proAptiv Multiprocessing System™ CPU instruction set, including the instruction formats and the basic instruction types.

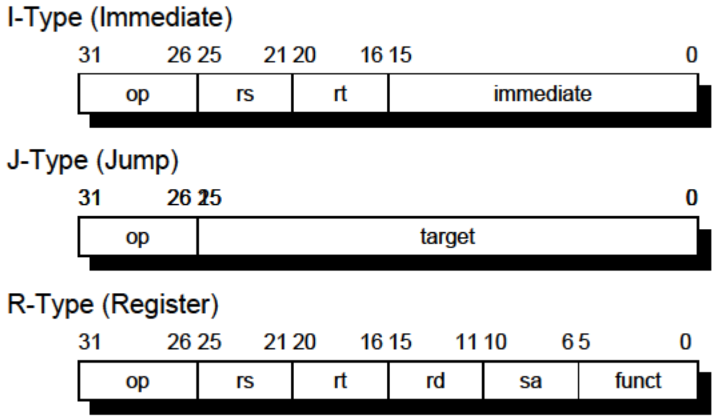
This chapter discusses the following topics:

- Section 17.1 “CPU Instruction Formats”
- Section 17.2 “Load and Store Instructions”
- Section 17.3 “Computational Instructions”
- Section 17.4 “Jump and Branch Instructions”
- Section 17.5 “Control Instructions”
- Section 17.6 “Coprocessor Instructions”
- Section 17.7 “New Instructions for the proAptiv™ Core”
- Section 17.8 “Base Instruction Set for the proAptiv™ CPU”
- Section 17.9 “Instruction Latencies and Repeat Rates”

17.1 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 17.1.

Figure 17.1 Instruction Formats



op	6-bit operation code
rs	5-bit source register specifier
rt	5-bit target (source/destination) register or branch condition
immediate	16-bit immediate value, branch displacement or address displacement
target	26-bit jump target address
rd	5-bit destination register specifier
sa	5-bit shift amount
funct	6-bit function field

17.2 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.

17.2.1 Defining Access Types

Access type indicates the size of a CPU 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 memory. 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 17.1. Only the combinations shown in Table 17.1 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 17.1 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							

17.3 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

17.4 Jump and Branch Instructions

Jump and branch instructions change the control flow of a program. All MIPS32 R3 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.

17.4.1 Overview of Jump Instructions

Subroutine calls in high-level languages are usually implemented with Jump (J) or Jump and Link (JAL) instructions, both of which have the J-type format. 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 and large cross-page jumps are usually implemented with the Jump Register (JR) or Jump and Link Register (JALR) 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*.

17.4.2 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 MIPS32 R3 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.

17.5 Control Instructions

Control instructions allow the software to initiate traps; they are always R-type.

17.6 Coprocessor Instructions

CP0 instructions perform operations on the System Control Coprocessor registers to manipulate the memory management and exception handling facilities of the processor.

CP1 instructions perform operations on the floating point unit (FPU).

CP2 instructions are not implemented in proAptiv.

17.7 New Instructions for the proAptiv™ Core

This section describes the new instructions added to the MIPS32 Release 3 architecture. Table 17.2 lists the instructions in alphabetical order. Additional instruction information in the*MIPS32® Architecture Reference Manual* is not duplicated here.

Table 17.2 proAptiv Multiprocessing System™ CPU Instruction Set

Instruction	Description	Function
`CACHEE`	Cache Instruction EVA	See the description of the CACHE/CACHEE instruction in Chapter 19.
`LBE`	Load Byte EVA	Same as the LB instruction, except that it allows kernel address to user addresses. Refer to the LB instruction in Table 17.3 for functional information.
`LBUE`	Load Byte Unsigned EVA	Same as the LBU instruction, except that it allows kernel address to user addresses. Refer to the LBU instruction Table 17.3 for functional information.
`LHE`	Load Halfword EVA	Same as the LH instruction, except that it allows kernel address to user addresses. Refer to the LH instruction Table 17.3 for functional information.
`LHUE`	Load Halfword Unsigned EVA	Same as the LB instruction, except that it allows kernel address to user addresses. Refer to the LH instruction Table 17.3 for functional information.
`LLE`	Load Linked EVA	Same as the LL instruction, except that it allows kernel address to user addresses. Refer to the LL instruction Table 17.3 for functional information.
`LWE`	Load Word EVA	Same as the LW instruction, except that it allows kernel address to user addresses. Refer to the LW instruction in Table 17.3 for functional information.
`LWLE`	Load Word Left EVA	Same as the LWL instruction, except that it allows kernel address to user addresses. Refer to the LWL instruction in Table 17.3 for functional information.
`LWRE`	Load Word Right EVA	Same as the LWR instruction, except that it allows kernel address to user addresses. Refer to the LWR instruction in Table 17.3 for functional information.
`PREFE`	Prefetch EVA	Load Specified Line into Cache. See also the description of the PREF / PREFE instruction in.Chapter 19.

Table 17.2 proAptiv Multiprocessing System™ CPU Instruction Set *(continued)*

Instruction	Description	Function
SBE	Store Byte EVA	Same as the SB instruction, except that it allows kernel addess to user addresses. Refer to the SB instruction in Table 17.3 for functional information.
SCE	Store Conditional EVA	Same as the SC instruction, except that it allows kernel addess to user addresses. Refer to the SC instruction in Table 17.3 for functional information.
SHE	Store Halfword EVA	Same as the SH instruction, except that it allows kernel addess to user addresses. Refer to the SH instruction in Table 17.3 for functional information.
SWE	Store Word EVA	Same as the SW instruction, except that it allows kernel addess to user addresses. Refer to the SW instruction in Table 17.3 for functional information.
SWLE	Store Word Left EVA	Same as the SWL instruction, except that it allows kernel addess to user addresses. Refer to the SWL instruction in Table 17.3 for functional information.
SWRE	Store Word Right EVA	Same as the SWR instruction, except that it allows kernel addess to user addresses. Refer to the SWR instruction in Table 17.3 for functional information.
TLBINV	TLB Invalidate	TLBINV invalidates a set of TLB entries based on ASID and *Index* match. On execution of the TLBINV instruction, the set of TLB entries with matching ASID are marked invalid, excluding those TLB entries which have their G bit set to 1. For more informaiton, refer to the TLBINV instruction in Chapter 19.
TLBINVF	TLB Invalidate Flush	TLBINV invalidates a set of TLB entries based on ASID and *Index* match. On execution of the TLBINVF instruction, all entries within range of *Index* are invalidated. For more informaiton, refer to the TLBINVF instruction in Chapter 19.

17.8 Base Instruction Set for the proAptiv™ CPU

This section describes the base instructions for the MIPS32 Release 3 architecture. Table 17.3 lists the instructions in alphabetical order. Following the table, the instructions that have implementation-dependent behavior in the proAptiv Multiprocessing System CPU are described individually. The descriptions of other instructions that exist in the*MIPS32® Architecture Reference Manual* are not duplicated here.

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.

Table 17.3 proAptiv Multiprocessing System™ CPU 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`

`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`

Table 17.3 proAptiv Multiprocessing System™ CPU Instruction Set *(continued)*

Instruction	Description	Function

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

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.Refer to the Caches chapter for more information.
CLO	Count Leading Ones	Rd = NumLeadingOnes(Rs)
CLZ	Count Leading Zeroes	Rd = NumLeadingZeroes(Rs)
COP0	Coprocessor 0 Operation	See Software User’s Manual
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

Table 17.3 proAptiv Multiprocessing System™ CPU Instruction Set *(continued)*

Instruction	Description	Function

`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;` $\texttt{Status}_{\texttt{IE}}$ `= 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`

`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]`
`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 837.

`LUI`	Load Upper Immediate	`Rt = immediate << 16`
`LW`	Load Word	`Rt = Mem[Rs+offset]`
`LWPC`	Load Word, PC relative	`Rt = Mem[PC+offset]`

Table 17.3 proAptiv Multiprocessing System™ CPU Instruction Set *(continued)*

Instruction	Description	Function
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]
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

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
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 839.
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$_{31..Rs}$

Table 17.3 proAptiv Multiprocessing System™ CPU Instruction Set *(continued)*

Instruction	Description	Function
`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 841.

`SDBBP`	Software Debug Break Point	`Trap to SW Debug Handler`
`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`
`SWL`	Store Word Left	See Architecture Reference Manual
`SWR`	Store Word Right	See Architecture Reference Manual

Table 17.3 proAptiv Multiprocessing System™ CPU Instruction Set *(continued)*

Instruction	Description	Function
`SYNC`	Synchronize	See the description of the SYNC instruction on page 843.
`SYNCI`	Synchronize Caches to Make Instruction Writes Effective	For D-cache writeback and I-cache invalidate on specified address
`SYSCALL`	System Call	`SystemCallException`

`TEQ`	Trap if Equal	`if Rs == Rt`
		`  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 853.
`TLBWR`	Write Random TLB Entry	See the description of the TLBWR instruction on page 855.
`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 851.

`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 857.
`WRPGPR`	Write to GPR in Previous Shadow Set	SGPR[SRSCtl$_{PSS}$, Rd] = Rt

Table 17.3 proAptiv Multiprocessing System™ CPU Instruction Set *(continued)*

Instruction	Description	Function
WSBH	Word Swap Bytes Within HalfWords	Rd = $Rt_{23..16}$ || $Rt_{31..24}$ || $Rt_{7..0}$ || $Rt_{15..8}$
XOR	Exclusive OR	Rd = Rs ^ Rt
XORI	Exclusive OR Immediate	Rt = Rs ^ (uns)Immed
ZEB	Zero extend byte (MIPS16 only)	Rt = (ubyte) Rs
ZEH	Zero extend half (MIPS16 only)	Rt = (uhalf) Rs

17.9 Instruction Latencies and Repeat Rates

Latency is defined with respect to instruction pair, but for ease of documenting they are defined for the instruction. If the behavior per instruction differs from that of an instruction pair, this difference is mentioned in the Notes column. If the instruction does not loads any general purpose register (GPR) then it is shown as not applicapble (n/a). Note that the DSP instruction latencies are listed in the DSP chapter.

Repeat rate is measured as number of independent instructions that can be sent in 1 cycle.

Table 17.4 proAptiv Instruction Latencies and Repeat Rates

Instruction	Latency	Repeat Rate	Notes
Optimized Operations			
ADD, ADDU, ADDI, ADDIU, AND, ANDI, LUI, OR, ORI, XOR, XORI, NOR, SLL (shift_amount<=8), SRL (24<shift_amount<=31), SEB, SEH, SLT, SLTU, SLTI, SLTIU, SUB, SUBU	1	2	
Larger Shifts and Rotates			
ROTR, ROTRV, SLL (shift_amount>8), SLLV, SRA, SRAV, SRL (shift_amount<=24), SRLV	2	1	
Counting			
CLO, CLZ	2	1	
Data Dependent			
DIV, DIVU	n/a	blocking	
Multiply			
MUL	6	1	
CPU Branch and Jump without Link			
B, BEQ, BNE, BGEZ, BGTZ, BLEZ, BLTZ, J, JR	n/a	1	
CPU Branch and Jump with Link			
BAL, BGEZAL, BLTZAL, JAL, JALR, JALX	n/a, 2	1	
Obsolete CPU Branch Instructions without Link			
BEQL, BNEL, BGEZL, BGTZL, BLEZL, BLTZL	n/a	1	
Obsolete CPU Branch Instructions with Link			
BGEZALL, BLTZALL	n/a, 2	1	
FPU Branch Instructions			
BC1F, BC1T	n/a	1	
Obsolete FPU Branch Instructions			
BC1FL, BC1TL	n/a	1	
Hazard Barriers			
JR.HB, EHB	n/a	1	
Hazard Barriers with Link			
JALR.HB	n/a	2 + 1	
No Operation			

Table 17.4 proAptiv Instruction Latencies and Repeat Rates *(continued)*

Instruction	Latency	Repeat Rate	Notes
NOP, SSNOP, PAUSE	n/a	2	
Loads			
LB, LBU, LH, LHU, LW, LWL, LWR, LL	4	1	Assuming L1 data cache.
Stores			
SB, SH, SW, SWL, SWR, PREF, PREFX, SYNC, SYNCI	n/a	1	
EXT, INS, WSBH	2	1	
SC	13	1	
Integer Move Based On Floating Point Condition Code			
MOVF, MOVT	1	1	
Integer Move Based On Integer Register Value			
MOVN, MOVZ	1	1	
Read Hardware Register			
RDHWR	5	1	
FPU Move Instructions			
CFC1, MFC1, MFHC1, CTC1, MTC1, MTHC1			
Trap Instructions			
TEQ, TEQI, TGE, TGEI, TGEIU, TGEU	n/a	1	
TLT, TLTI, TLTIU, TLTU, TNE, TNEI, BREAK, SYSCALL	n/a	1	
FPU Load/Store Instructions			
LDC1, LDXC1, LUXC1, LWC1, LWXC1, SDC1, SDXC1, SUXC1, SWC1, SWXC1			
Privileged Instructions			
CACHE, ERET	n/a	1	
MFC0	5	1	
RDPGPR	n/a	1	Waits for the instruction to graduate before issuing dependents.
MTC0	n/a	1	
TLBP, TLBR, TLBWI, TLBWR	n/a	1	
WAIT, WRPGPR	n/a	1	
DI, EI	n/a	1	Since this instruction causes execution hazard barrier, software should handle it so the latency is not applicable.
EJTAG Instructions			
DERET, SDBBP	n/a	1	

Chapter 18

MIPS16e Application-Specific Extension to the MIPS32® Instruction Set

This chapter describes the MIPS16e ASE as implemented in the proAptiv 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 18.1 "Instruction Bit Encoding"
- Section 18.2 "Instruction Listing"

18.1 Instruction Bit Encoding

Table 18.2 through Table 18.9 describe the encoding used for the MIPS16e ASE. Table 18.1 describes the meaning of the symbols used in the tables.

Table 18.1 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 18.2 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 18.3 MIPS16e JAL(X) Encoding of the x Field

x	bit 26	
	0	1
	JAL	JALX

Table 18.4 MIPS16e SHIFT Encoding of the f Field

f	bits 1..0			
	0	1	2	3
	00	01	10	11
	SLL	β	SRL	SRA

Table 18.5 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 18.6 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 18.7 MIPS16e RRR Encoding of the f Field

f	bits 1..0			
	0	1	2	3
	00	01	10	11
	β	ADDU	β	SUBU

Table 18.8 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 18.9 MIPS16e I8 Encoding of the s Field when funct=SVRS

s	bit 7	
	0	1
	RESTORE	SAVE

Table 18.10 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 18.11 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	*

18.2 Instruction Listing

The MIPS16e instructions are listed by instruction type in Table 18.12 through Table 18.19.

Table 18.12 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 18.13 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 18.14 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 18.15 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 18.15 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 18.16 MIPS16e Special Instructions

Mnemonic	Instruction	Extensible Instruction
BREAK	Breakpoint	No
SDBBP	Software Debug Breakpoint	No
EXTEND	Extend	No

Table 18.17 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 18.18 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 18.18 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 18.19 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

Chapter 19

Implementation-specific Instructions

This chapter describes the architectural definition for the following implementation-specific instructions in the proAptiv Multiprocessing System.

- CACHE: Cache Operation
- LL: Load Linked Word
- PREF: Prefetch
- SC: Store Conditional
- SYNC: Synchronize Shared Memory
- TLBR: Read Indexed TLB Entry
- TLBWI: Write Indexed TLB Entry
- TLBWR: Write Random TLB Entry
- WAIT: Enter Standby Mode
- CACHEE: Cache Operation EVA
- LLE: Load Link EVA
- PREFE: Prefetch EVA
- SCE: Store Conditional EVA

31 26	25 21	20 16	15 0
CACHE 101111	base	op	offset
6	5	5	16

Format: `CACHE op, offset(base)` **MIPS32**
`CACHEE op, offset(base) — Extended Virtual Address (EVA)`

Purpose: Perform Cache Operation

To perform the cache operation specified by op. The CACHE and CACHEE instructions perform identical operations with one exception — when the processor is configured in Enhanced Virtual Address (EVA) mode, the CACHEE instruction is used to perform the virtual address translation using the user mapping of the address rather than the kernel mapping. The CACHE instruction can be used with unmapped addresses, in non-EVA mode, or when the kernel mapping is required.

Description:

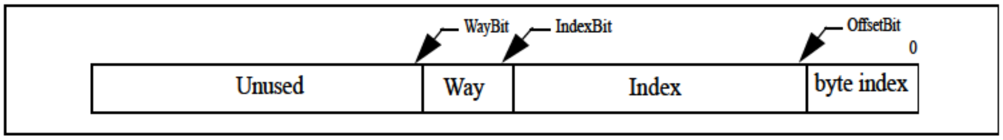
The 16-bit offset is sign-extended and added to the contents of the base register to form an effective address. The effective address is used in one of the following ways based on the operation to be performed and the type of cache as described in the following table.

Table 19.1 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 ← $\mathrm{Addr}_{\mathrm{WayBit-1..IndexBit}}$
		Index ← $\mathrm{Addr}_{\mathrm{IndexBit-1..OffsetBit}}$

Figure 19.1 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. The preferred implementation is not to match on the cache instruction.

Bits [17:16] of the instruction specify the cache on which to perform the operation, as follows

Table 19.2 Encoding of Bits[17:16] of CACHE Instruction

Code	Name	Cache	Cop0 Registers Used
2’b00	I	Primary Instruction	*ITagLo*, *ITagHi*, *IDataLo*, *IDataHi*, *ErrCtl*
2’b01	D	Primary Data	*DTagLo*, *DTagHi*, *DDataLo*, *ErrCtl*
2’b10	T	Tertiary - Not supported	
2’b11	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 19.2 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 or 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 19.3 shows the normal mode condition where the $ErrCtl_{WST}$, $ErrCtl_{SPR}$, and $ErrCtl_{DYT}$ bits of the CP0 *ErrCtl* register are all cleared. Refer to the *ErrCtl* register of the CP0 chapter for more information.

Table 19.3 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST, SPR, DYT] Cleared

Code	Caches	Name	Effective Address Operand Type	Operation
3’b000	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.
	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.

Table 19.3 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST, SPR, DYT] Cleared *(continued)*

Code	Caches	Name	Effective Address Operand Type	Operation
3’b001	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 dword index into the *IDataLo* and *IDataHi* registers. • If parity is implemented, read the parity bits corresponding to the data into the $ErrCtl_{PI}$ field.
3’b001	D	Index Load Tag	Index	• Read the tag for the cache line at the specified index into the CP0 *DTagLo* register. • Read the data corresponding to the word index into the *DDataLo* register. • Data array parity bits are read into the *ErrCtl* register.
3’b001	S	Index Load Tag	Index	• Read the tag for the cache line at the specified index into theCP0 *L23TagLo* register. • Read the data corresponding to the dword index into the *L23DataLo* and *L23DataHi* registers.
3’b010	I	Index Store Tag	Index	• Write the tag for the cache block at the specified index from the *ITagLo* and *ITagHi* registers. • Parity written into the cache is generated by the hardware if $ErrCtl_{PO}$ = 0, or it is obtained from *ITagLo* if $ErrCtl_{PO}$ = 1.
3’b010	D	Index Store Tag	Index	Write the tag for the cache line at the specified index from the CP0 *DTagLo* (data cache) register. By default, the tag parity value will be automatically calculated. For test purposes, the parity bits from the *DTagLo* 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 *DTagLo* register associated with the cache be initialized first.
3’b010	S	Index Store Tag	Index	Write the tag for the L2 cache line at the specified index from the CP0 *L23TagLo* (L2 cache) register. By default, the tag parity value will be automatically calculated. For test purposes, the parity/ECC bits from the *L23TagLo* register will be used if $ErrCtl_{PO}$ is set. The L2 cache ECC bits come from the *L23TagLo* register.
3’b011	I, D	Index Store Data	Unspecified	Write the data for the cache line at the specified index from the associated DataHi/DataLo Coprocessor 0 registers. Writes to instruction cache are doubleword wide and use IDataLo/IDataHi Coprocessor 0 registers. Writes to data cache are word wide and use the data from DDataLo register.
3’b011	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).

Table 19.3 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST, SPR, DYT] Cleared *(continued)*

Code	Caches	Name	Effective Address Operand Type	Operation
3’b100	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.
3’b101	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
	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.
3’b110	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.
3’b111	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.

Table 19.4 shows the condition for the way select test where the *$ErrCtl_{WST}$* bit is set, and the *$ErrCtl_{SPR}$* and *$ErrCtl_{DYT}$* bits of the CP0 *ErrCtl* register are cleared. Refer to the *ErrCtl* register of the CP0 chapter for more information.

Table 19.4 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST] Set, ErrCtl[SPR, DYT] Cleared

Code	Caches	Name	Effective Address Operand Type	Operation
3'b001	All	Index Load WS	Index	Read the WS RAM at the specified index into the associated *ItagLo, DTagLo, or L23TagLo* CP0 register.
3'b010	I	Index Store WS	Index	Update the WS RAM at the specified index from the *ITagLo* CP0 register.
3'b010	D	Index Store WS	Index	Update the WS RAM at the specified index from the *DTagLo* CP0 register. If *$ErrCtl_{PO}$* is set, the dirty parity values in the *DTagLo* register will be written to the WS RAM. Otherwise, the parity will be calculated for the write data.
3'b010	S	Index Store WS	Index	Update the WS RAM at the specified index from the *L23TagLo* CP0 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.
3'b011	I	Index Store Data	Index	Write the *IDataLo* and *IDataHi* CP0 register contents at the way and dword index specified. If *$ErrCtl_{PO}$* is set, the dirty parity values in the *ITagLo* register will be written to the WS RAM. Otherwise, the parity will be calculated for the write data.
3'b011	D	Index Store Data	Index	Write the *DDataLo* CP0 register contents at the way and word index specified.
3'b011	S	Index Store ECC	Index	Write the *L23DataLo* register contents to the *ECC* bits at the way and dword index specified.
All Others	All			Other codes should not be used while *$ErrCtl_{WST}$* is set.

Table 19.5 shows the condition for the SPRAM access test where the *ErrCtl*$_{SPR}$ bit is set, and the *ErrCtl*$_{WTS}$ and *ErrCtl*$_{DYT}$ bits of the CP0 *ErrCtl* register are cleared. Refer to the *ErrCtl* register of the CP0 chapter for more information.

Table 19.5 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[SPR] Set, ErrCtl[WST, DYT] Cleared

Code	Caches	Name	Effective Address Operand Type	Operation
3'b001	I	Index Load Tag	Index	Read the SPRAM tag at the specified index into the *ITagLo* Coprocessor 0 register. Also read the instruction data and precode information corresponding to the byte index into the *IDataHi*,*IDataLo*, and *ErrCtl* registers
3'b001	D	Index Load Tag	Index	Read the SPRAM tag at the specified index into the *TagLo1* Coprocessor 0 register.
3'b010	I	Index Store Tag	Index	Update the SPRAM tag at the specified index from the *TagLo* Coprocessor 0 register.
3'b010	D	Index Store Tag	Index	Update the SPRAM tag at the specified index from the *TagLo* Coprocessor 0 register.
3'b011	I	Index Store Data	Index	Write the *IDataLo* and *IDataHi* Coprocessor 0 register contents into the SPRAM at the dword index specified. If *ErrCtl*$_{PO}$ is set, the dirty parity values in the *ITagLo* register will be written to the WS RAM. Otherwise, the parity will be calculated for the write data.
3'b011	D	Index Store Data	Index	Write the *DDataLo* Coprocessor 0 register contents into the SPRAM at the word 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. If ECC is enabled, the ECC value comes from the *DTagHi* register.
All Others	D			Other codes should not be used while *ErrCtl*$_{SPR}$ is set.
All	S			Secondary (L2) cache operations should not be performed while *ErrCtl*$_{SPR}$ is set.

Table 19.6 shows the condition for the duplicate tag array access where the *ErrCtl*$_{WST}$ and *ErrCtl*$_{SPR}$ bits are set, and the *ErrCtl*$_{DYT}$ bit of the CP0 *ErrCtl* register is cleared. Refer to the *ErrCtl* register of the CP0 chapter for more information.

Table 19.6 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[WST, SPR] Set, ErrCtl[DYT] Cleared

Code	Caches	Name	Effective Address Operand Type	Operation
3’b001	D	Index Load Tag	Index	Read the duplicate tag array into the CP0 *DTagLo* register.
3’b010	D	Index Store Tag	Index	Writes the duplicate tag array from the CP0 *DTagLo* register. By default, the tag parity value will be automatically calculated. For test purposes, the parity bits from the *DTagLo* register will be used if *ErrCtl*$_{PO}$ is set.
All Others	D			Other codes should not be used while *ErrCtl*$_{WST}$ and *ErrCtl*$_{SPR}$ are set.

Table 19.6 shows the condition for the duplicate tag array access where the *ErrCtl*$_{DYT}$ bit is set, and the *ErrCtl*$_{WST}$ and *ErrCtl*$_{SPR}$ bits of the CP0 *ErrCtl* register are cleared. Refer to the *ErrCtl* register of the CP0 chapter for more information.

Table 19.7 Encoding of Bits [20:18] of the CACHE Instruction, ErrCtl[DYT] Set, ErrCtl[WST, SPR] Cleared

Code	Caches	Name	Effective Address Operand Type	Operation
3’b001	D	Index Load Tag	Index	Read the dirty RAM at the specified index into the *DTagLo1* CP0 register.
3’b010	D	Index Store Tag	Index	Update the dirty RAM at the specified index from the *DTagLo1* CP0 register.
All Others	D			Other codes should not be used while *ErrCtl*$_{DYT}$ is set.

Restrictions:

The operation of this instruction is **UNDEFINED** for any operation/cache combination that is not implemented.

The operation of this instruction is **UNDEFINED** if the operation requires an address, and that address is uncacheable.

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:

```
if IsCoprocessorEnabled(0) then
   vAddr ♦ GPR[base] + sign_extend(offset)
   (pAddr, uncached) ♦ AddressTranslation(vAddr, DataReadReference)
   CacheOp(op, vAddr, pAddr)
else
   SignalException(CoprocessorUnusable, 0)
endif
```

Exceptions:

TLB Refill Exception.

TLB Invalid Exception

Coprocessor Unusable Exception

Address Error Exception

Cache Error Exception

Bus Error Exception

31 26	25 21	20 16	15 0
LL 110000	base	rt	offset
6	5	5	16

Format: `LL rt, offset(base)` **MIPS32**
`LLE rt, offset(base) — Extended Virtual Address (EVA)`

Purpose: Load Linked Word

To load a word from memory for an atomic read-modify-write. The LL and LLE instructions perform identical operations with one exception — when the processor is configured in Enhanced Virtual Address (EVA) mode, the LLE instruction is used to perform the virtual address translation using the user mapping of the address rather than the kernel mapping. The LL instruction can be used with unmapped addresses, in non-EVA mode, or when the kernel mapping is required.

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 CPU. There can be only one active RMW sequence per CPU. When an LL is executed, it starts an active RMW sequence replacing any other sequence that was active. The read-modify-write (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 CPU does not cause an action that, by itself, causes an SC for the same block to fail on another CPU.

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 CPU's 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 $vAddr_{1..0}$ … $0^2$ 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**
`PREFE hint, offset(base) — Extended Virtual Address (EVA)`

Purpose: Prefetch

To move data between memory and cache. The PREF and PREFE instructions perform identical operations with one exception — when the processor is configured in Enhanced Virtual Address (EVA) mode, the PREFE instruction is used to perform the virtual address translation using the user mapping of the address rather than the kernel mapping. The PREF instruction can be used with unmapped addresses, in non-EVA mode, or when the kernel mapping is required.

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 19.8 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	
4	load_streamed	Use: Prefetched data is expected to be read (not modified) but not reused extensively; it “streams” through cache.
5	store_streamed	Use: Prefetched data is expected to be stored or modified but not reused extensively; it “streams” through cache.

Table 19.8 Values of *hint* Field for PREF Instruction

6	load_retained	Use: Prefetched data is expected to be read (not modified) and reused extensively; it should be “retained” in the cache.
7	store_retained	Use: Prefetched data is expected to be stored or modified and reused extensively; it should be “retained” in the cache.
8-24	Reserved	
25	writeback_invalidate (also known as “nudge”)	
26-29	Reserved	
30		
31		

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**
`SCE rt, offset(base) — Extended Virtual Address (EVA)`

Purpose: Store Conditional Word

To store a word to memory to complete an atomic read-modify-write. The SC and SCE instructions perform identical operations with one exception — when the processor is configured in Enhanced Virtual Address (EVA) mode, the SCE instruction is used to perform the virtual address translation using the user mapping of the address rather than the kernel mapping. The SC instruction can be used with unmapped addresses, in non-EVA mode, or when the kernel mapping is required.

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**
`SYNCE (stype = 0 implied) — Extended Virtual Address (EVA)`

Purpose: Synchronize Shared Memory

To order loads and stores. The SYNC and SYNCE instructions perform identical operations with one exception — when the processor is configured in Enhanced Virtual Address (EVA) mode, the SYNCE instruction is used to perform the virtual address translation using the user mapping of the address rather than the kernel mapping. The SYNC instruction can be used with unmapped addresses, in non-EVA mode, or when the kernel mapping is required.

Description:

These types of ordering guarantees are available through the SYNC instruction:

- Completion Barriers
- Ordering Barriers

Simple Description of Completion Barrier:

- SYNC affects only *uncached* and *cached coherent* loads and stores. The specified memory instruction (loads or stores or both) that occur before the SYNC / SYNCE instruction must be completed before the loads and stores after the SYNC / SYNCE 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.

Detailed Description of Completion Barrier:

- Every synchronizable specified memory instruction (loads or stores or both) that occurs in the instruction stream before the SYNC instruction must be already globally performed before any synchronizable specified memory instruction that occurs after the SYNC are allowed to be performed, with respect to any other processor or coherent I/O module.
- The barrier does not guarantee the order in which instruction fetches are performed.
- A stype value of zero will always be defined such that it performs the most complete set of synchronization operations that are defined.This means stype zero always does a completion barrier that affects both loads and stores preceding the SYNC instruction and both loads and stores that are subsequent to the SYNC instruction. Non-zero values of stype may be defined by the architecture or specific implementations to perform synchronization behaviors that are less complete than that of stype zero. If an implementation does not use one of these non-zero values to define a different synchronization behavior , then that non-zero value of stype must act the same as stype zero completion barrier. This allows software written for an implementation with a lighter-weight barrier to work on another implementation which only implements the stype zero completion barrier.
- A completion barrier 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 completion barrier is required on entry to and exit from Debug Mode to guarantee that memory affects are handled correctly.

- For the purposes of this description, the CACHE (CACHEE), PREF (PREFE) and PREFX instructions are treated as loads and stores. That is, these instructions and the memory transactions sourced by these instructions obey the ordering and completion rules of the completion barrier SYNC instruction.

Completion Barrier Types:

All completion barrier types will flush any pending writes and generate an external SYNC request. The core will wait for all pending reads to complete as well as the SYNC response.

- 0x2 - Implementation specific stype. Intervention SYNC. When coherence is enabled, this SYNC will generate a CoherentSync request. The CoherenceManager will respond to the SYNC when the interventions for all older coherent requests have been completed. If coherence is not enabled, will default to stype 0x0.

- 0x3 - Implementation specific stype. Memory SYNC. When coherence is enabled, this SYNC will also generate a CoherentSync request. When interventions for all older coherent requests have completed, the sync will be sent to memory interface unit. All pending transactions will be sent out. If the next level device (L2 or system) supports legacy SYNC transactions, as indicated by SI_CM_SyncTxEn = 1, and CM_SYNC_TX_DISABLE in the CM Control GCR is 0, an external SYNC request will also be generated. The CM will send a response to the CoreType-lowercase when all prior requests have completed and a SYNC response is received (if it was externalized). If coherence is not enabled, will default to stype 0x0.

- 0x0 - If coherence is enabled, this will be mapped to either a type 0x2 or 0x3 based on the value of the SYNCCTL bit in the CM Control GCR. If coherence is not enabled, a legacy SYNC request will be generated. This will bypass the intervention pipeline in the CM and go directly to the memory unit. If SyncTxEn = 1 and CM_SYNC_TX_DISABLE in the CM Control GCR is 0, an external SYNC request will be generated.

Simple Description of Ordering Barrier:

- The specified memory instructions (loads or stores or both) that occur before the SYNC instruction must always be ordered and globally visible to all cores before the specified memory instructions after the SYNC.

- Memory instructions which are ordered before other memory instructions are processed by the load/store datapath first before the other memory instructions.

Detailed Description of Ordering Barrier:

- Every synchronizable specified memory instruction (loads or stores or both) that occurs in the instruction stream before the SYNC instruction must reach a stage in the load/store datapath after which no instruction re-ordering is possible before any synchronizable specified memory instruction which occurs after the SYNC instruction in the instruction stream reaches the same stage in the load/store datapath.

  NOTE: cached and uncached operations proceed down different data paths within the Coherence Manager. Because of that, this type of barrier will not enforce the ordering between cached and uncached requests. A Completion Barrier should be used if that ordering is required.

- If any memory instruction before the SYNC instruction in program order, generates a memory request to the external memory and any memory instruction after the SYNC instruction in program order also generates a memory request to external memory of the same type (cached or uncached), the memory request belonging to the older instruction must be globally performed before the time the memory request belonging to the younger instruction is globally performed.

- The barrier does not guarantee the order in which instruction fetches are performed.
- This barrier does not enforce the ordering of CACHE instructions. To ensure ordering of a CACHE instruction with other operations, a completion barrier type of SYNC should be used.
- For the purposes of this description, PREF and PREFX instructions are treated as loads and obey the same ordering rules as loads.

As compared to the completion barrier, the ordering barrier is a lighter-weight operation as it does not require the specified instructions before the SYNC to be already completed. Instead it only requires that those specified instructions which are subsequent to the SYNC in the instruction stream are never re-ordered for processing ahead of the specified instructions which are before the SYNC in the instruction stream. This potentially reduces how many cycles the barrier instruction must stall before it completes.

The Acquire and Release barrier types are used to minimize the memory orderings that must be maintained and still have software synchronization work.

The proAptiv Multiprocessing System core handles all ordering barriers identically.

• No external SYNC request will be generated and the CoreType-lowercase will not wait for pending transactions to complete.

• The LSU will complete any pending evictions and wait until self interventions have been received for all Fill Store Buffers before proceeding.

NOTE: globalized CACHE instructions do not use an FSB entry, thus an Ordering Barrier will not wait for those to be completed. A Completion Barrier should be used to ensure that all prior CACHE instructions have completed.

• The BIU will stop merging on all Write Back Buffer (WBB) entries and put them into the external request queue.

Table 19.9 lists the available completion barrier and ordering barriers behaviors that can be specified using the stype field.

Table 19.9 Encodings of the Bits[10:6] of the SYNC Instruction; the SType Field

Code	Name	Type	Older instructions which must reach the load/store ordering point before the SYNC instruction completes	Younger instructions which must reach the load/store ordering point only after the SYNC instruction completes	Older instructions which must be globally performed when the SYNC instruction completes
0x0	SYNC or SYNC(0)	Completion	Loads, Stores	Loads, Stores	Loads, Stores
0x2	SYNC(2)	Completion	Loads, Stores	Loads, Stores	Loads, Stores
0x3	SYNC(3)	Completion	Loads, Stores	Loads, Stores	Loads, Stores
0x4	SYNC_WMB or SYNC(4)	Ordering	Stores	Stores	
0x10	SYNC_MB or SYNC(16)	Ordering	Loads, Stores	Loads, Stores	

Table 19.9 Encodings of the Bits[10:6] of the SYNC Instruction; the SType Field

Code	Name	Type	Older instructions which must reach the load/store ordering point before the SYNC instruction completes	Younger instructions which must reach the load/store ordering point only after the SYNC instruction completes	Older instructions which must be globally performed when the SYNC instruction completes
0x11	SYNC_ACQUIRE or SYNC(17)	Ordering	Loads	Loads, Stores	
0x12	SYNC_RELEASE or SYNC(18)		Loads, Stores	Stores	
0x13	SYNC_RMB or SYNC(19)		Loads	Loads	
All Others	RESERVED				

Restrictions:

None

Operation:

```
SyncOperation(stype)
```

Exceptions:

None

Software written to use a SYNC instruction with a non-zero stype value, expecting one type of barrier behavior, should only be run on hardware that actually implements the expected barrier behavior for that non-zero stype value or on hardware which implements a superset of the behavior expected by the software for that stype value. If the hardware does not perform the barrier behavior expected by the software, the system may fail.

31 26	25	24 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBINV 000011
6	1	19	6

Format: `TLBINV`

Purpose: TLB Invalidate

TLBINV invalidates a set of TLB entries based on ASID and *Index* match. The virtual address is ignored in the entry match. TLB entries which have their G bit set to 1 are not modified.

Description:

On execution of the TLBINV instruction, the set of TLB entries with matching ASID are marked invalid, excluding those TLB entries which have their G bit set to 1.

The $\textit{EntryHI}_{ASID}$ field has to be set to the appropriate ASID value before executing the TLBINV instruction.

Behavior of the TLBINV instruction applies to all applicable TLB entries and is unaffected by the setting of the *Wired* register.

For JTLB-based MMU($\textit{Config}_{MT}$ = 1):

All matching entries in the JTLB are invalidated. *Index* is unused.

Restrictions:

The operation is **UNDEFINED** if the contents of the *Index* register are greater than or equal to the number of available TLB entries ($\textit{Config4}_{IE}$ = 2).

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:

The TLB invalidate instruction invalidates all entries of the TLB starting from 0.

```
for (i = 0 to $Config1_{MMU\_SIZE}$ - 1)
   if ($EntryHi_{ASID}$ = TLB[i].ASID) then
         $TLB[i]_{VPN2\_invalid}$ ♦ 1
   endif
endfor
```

Exceptions:

Coprocessor Unusable

31 26	25	24 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBINVF 000100
6	1	19	6

Format: `TLBINVF`

Purpose: TLB Invalidate Flush

TLBINVF invalidates a set of TLB entries based on *Index* match. The virtual address and ASID are ignored in the entry match.

Description:

On execution of the TLBINVF instruction, all entries within range of *Index* are invalidated.

Behavior of the TLBINV instruction applies to all applicable TLB entries and is unaffected by the setting of the *Wired* register.

For JTLB-based MMU($\mathit{Config}_{\text{MT}}$=1):

TLBINVF causes all entries in the JTLB to be invalidated. *Index* is unused.

Restrictions:

The operation is **UNDEFINED** if the contents of the *Index* register are greater than or equal to the number of available TLB entries ($\mathit{Config4}_{\mathit{IE}}$=2).

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

Operation:

```
for (i = 0 to $\text{Config1}_{\text{MMU\_SIZE}}$ - 1)
    $\text{TLB[i]}_{\text{VPN2\_invalid}}$ ♦ 1
endfor
```

Exceptions:

Coprocessor Unusable

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. 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:

```
if IsCoprocessorEnabled(0) then
i ♦ Index
if i > (TLBEntries - 1) then
   UNDEFINED
endif
PageMask$_{Mask}$ ♦ TLB[i]$_{Mask}$
EntryHi ♦ TLB[i]$_{VPN2}$ ||
          $0^2$ || TLB[i]$_{VPN2\_invalid}$ || $0^2$ || TLB[i]$_{ASID}$
EntryLo1 ♦ TLB[i]$_{RI1}$ || TLB[i]$_{XI1}$ ||
          TLB[i]$_{PFN1}$ ||
          TLB[i]$_{C1}$ || TLB[i]$_{D1}$ || TLB[i]$_{V1}$ || TLB[i]$_{G}$
EntryLo0 ♦ TLB[i]$_{RI0}$ || TLB[i]$_{XI0}$ ||
          TLB[i]$_{PFN0}$ ||
          TLB[i]$_{C0}$ || TLB[i]$_{D0}$ || TLB[i]$_{V0}$ || TLB[i]$_{G}$

else
   SignalException(CoprocessorUnusable, 0)
endif
```

Exceptions:

Coprocessor Unusable

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, the existing entries are sliently invalidated 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:

```
if IsCoprocessorEnabled(0) then
i ♦ Index
TLB[i]$_{Mask}$ ♦ PageMask$_{Mask}$
    TLB[i]$_{VPN2\_invalid}$ ♦ EntryHi$_{VPN2\_invalid}$
TLB[i]$_{VPN2}$ ♦ EntryHi$_{VPN2}$
TLB[i]$_{ASID}$ ♦ EntryHi$_{ASID}$
TLB[i]$_{G}$ ♦ EntryLo1$_{G}$ and EntryLo0$_{G}$
TLB[i]$_{PFN1}$ ♦ EntryLo1$_{PFN}$
TLB[i]$_{C1}$ ♦ EntryLo1$_{C}$
TLB[i]$_{D1}$ ♦ EntryLo1$_{D}$
TLB[i]$_{RI1}$ ♦ EntryLo1$_{RI}$
TLB[i]$_{XI1}$ ♦ EntryLo1$_{XI}$
TLB[i]$_{V1}$ ♦ EntryLo1$_{V}$
TLB[i]$_{PFN0}$ ♦ EntryLo0$_{PFN}$
TLB[i]$_{C0}$ ♦ EntryLo0$_{C}$
TLB[i]$_{D0}$ ♦ EntryLo0$_{D}$
TLB[i]$_{RI0}$ ♦ EntryLo0$_{RI}$
TLB[i]$_{XI0}$ ♦ EntryLo0$_{XI}$
TLB[i]$_{V0}$ ♦ EntryLo0$_{V}$
else
    SignalException(CoprocessorUnusable, 0)
endif
```

Exceptions:

Coprocessor Unusable

Machine Check

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, the entries are sliently invalidated. 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:

```
if IsCoprocessorEnabled(0) then
    i ♦ Random
    TLB[i]$_{Mask}$ ♦ PageMask$_{Mask}$
    TLB[i]$_{VPN2\_invalid}$ ♦ 0
    TLB[i]$_{VPN2}$ ♦ EntryHi$_{VPN2}$ and not PageMask$_{Mask}$ # Implementation dependent
    TLB[i]$_{ASID}$ ♦ EntryHi$_{ASID}$
    TLB[i]$_{G}$ ♦ EntryLo1$_{G}$ and EntryLo0$_{G}$
    TLB[i]$_{PFN1}$ ♦ EntryLo1$_{PFN}$ and not PageMask$_{Mask}$ # Implementation dependent
    TLB[i]$_{C1}$ ♦ EntryLo1$_{C}$
    TLB[i]$_{D1}$ ♦ EntryLo1$_{D}$
    TLB[i]$_{RI1}$ ♦ EntryLo1$_{RI}$
    TLB[i]$_{XI1}$ ♦ EntryLo1$_{XI}$
    TLB[i]$_{V1}$ ♦ EntryLo1$_{V}$
    TLB[i]$_{PFN0}$ ♦ EntryLo0$_{PFN}$ and not PageMask$_{Mask}$ # Implementation dependent
    TLB[i]$_{C0}$ ♦ EntryLo0$_{C}$
    TLB[i]$_{D0}$ ♦ EntryLo0$_{D}$
    TLB[i]$_{RI0}$ ♦ EntryLo0$_{RI}$
    TLB[i]$_{XI0}$ ♦ EntryLo0$_{XI}$
    TLB[i]$_{V0}$ ♦ EntryLo0$_{V}$
else
    SignalException(CoprocessorUnusable, 0)
endif
```

Exceptions:

Coprocessor Unusable

Enter Standby Mode WAIT

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:

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

Appendix A

References

This appendix lists other publications available from MIPS Technologies, Inc. that are referenced elsewhere in this document. These documents may be included in the `$MIPS_HOME/$MIPS_CORE/doc` area of a typical proAptiv Multiprocessing System core release, or in some cases may be available on the MIPS web site, http://www mips.com.

1. MIPS32® proAptiv Multiprocessing System™ Hardware User’s Manual
   MIPS Document: MD00905

2. MIPS32® Architecture For Programmers, Volume I: Introduction to the MIPS32® Architecture
   MIPS Document: MD0082

3. MIPS32® Architecture For Programmers, Volume II: The MIPS32® Instruction Set
   MIPS Document: MD0086

4. MIPS32® Architecture For Programmers, Volume III: The MIPS32® Privileged Resource Architecture
   MIPS Document: MD0090

5. MIPS32® Architecture For Programmers, Volume IV-a: The MIPS16e™ Application-Specific Extension to the MIPS32® Architecture
   MIPS Document: MD00076

6. MIPS32® Architecture For Programmers, Volume IV-e: The MIPS® DSP Application-Specific Extension to the MIPS32® Architecture
   MIPS Document: MD00374

7. Open Core Protocol Specification
   Available from the OCP International Partnership at http://www.ocpip.org

Appendix B

Revision History

Change bars (vertical lines) in the margins of this document indicate significant changes in the document since its last release. Change bars are removed for changes that are more than one revision old.

Revision	Date	Description
01.00	November 10, 2012	Initial Early Access (EA) release of SUM.
01.10	December 19, 2012	Ongoing work on various chapters. Subsequent EA release of SUM.
01.20	February 8, 2013	Ongoing work on various chapters. Major update to CM2 Registers chapter. Subsequent EA release of proAptiv SUM.
01.21	March 8, 2013	GA release of SUM.
01.22	May 14, 2013	Ongoing work on various chapters. Major update to EJTAG chapter. Miscellaneous updates to other chapter based on internal review. Subsequent GA release of proAptiv SUM.