# IDT79RC5000™ RISC Microprocessor

# Reference Manual

**September 2000**

2975 Stender Way, Santa Clara, California 95054
Telephone: (800) 345-7015 • TWX: 910-338-2070 • FAX: (408) 492-8674
Printed in U.S.A.

DISCLAIMER

Integrated Device Technology, Inc. reserves the right to make changes to its products or specifications at any time, without notice, in order to improve design or performance and to supply the best possible product. IDT does not assume any responsibility for use of any circuitry described other than the circuitry embodied in an IDT product. The Company makes no representations that circuitry described herein is free from patent infringement or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent, patent rights or other rights, of Integrated Device Technology, Inc.

LIFE SUPPORT POLICY

Integrated Device Technology's products are not authorized for use as critical components in life support devices or systems unless a specific written agreement pertaining to such intended use is executed between the manufacturer and an officer of IDT.

1. Life support devices or systems are devices or systems which (a) are intended for surgical implant into the body or (b) support or sustain life and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.

2. A critical component is any components of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

# About This Manual

Notes

## Introduction

This user's manual includes both hardware and software information on the RV5000, a dual-issue RISC processor that serves many performance-critical applications. The processor's 260 Dhrystone MIPS performance and 4 GB/sec aggregate bandwidth makes it ideal for embedded applications, such as high-end internetworking systems, color printers, and graphics terminals, as well as low-cost general computing with special emphasis on flating-point operations and memory management.

### Additional information

Information not included in this manual such as mechanicals, package pin-outs and electrical characteristics can be found in the data sheet for this device, which is available from the IDT website **(www.idt.com)** as well as through your local IDT sales representative.

## Content Summary

**Chapter 1, "Overview,"** provides a complete introduction to the performance capabilities of the RV5000. Included in this chapter is a summary of features for the device as well as a system block diagram and internal register maps.

**Chapter 2, "Central Processing Unit,"** presents the functions of the CPU.

**Chapter 3, "Floating-Point Unit,"** presents the functions of the FPU.

**Chapter 4, "Instruction Pipeline,"** describes the pipeline activities occurring during each ALU pipeline stage, for load, store, and branch instructions.

**Chapter 5, "Integer (CPU) Exceptions,"** describes the integer exception processing done by the CPU, including an explanation of exception procesing followed by the format and use of each CPU exception register.

**Chapter 6, "Floating-Point Exceptions,"** describes the actions taken when the FPU cannot handle in the normal way either the operands or the results of a floating-point operation.

**Chapter 7, "Memory Management Unit,"** describes the processor virtual and physical address spaces, the virtual-to-physical address translation, the operation of the TLB in making these translations, and those System Control Coprocessor (CPO) registers that provide the software interface to the TLB.

**Chapter 8, "Cache,"** describes the on-chip primary cache, the individual operations of the primary cache, and the organization and operations of the on-chip secondary cache controller.

**Chapter 9, "Signal Descriptions,"** describes the signals used by and in conjunction with the R5000 processor, including System Interface, Clock Interface, Secondary Cache Interrace, Interrupt Interface, Joint Test Action Group (JTAG) Interface and Initialization Interface.

**Chapter 10, "System Interface Transactions,"** describes the system interface from the point of view of both the processor and the external agent.

**Chapter 11, "System Interface Protocols,"** contains a cycle-by-cycle description of the system interface protocols for each type of R5000 processor and external request.

**Chapter 12, "Interrupts,"** describes the hardware and nonmaskable interrupts.

**Chapter 13, "Error Checking,"** describes the two major types of data errors that can occur in data transmissio: hard errors and soft effors.

**Chapter 14, "Initialization Interface,"** describes the following reset and control signals: VccOK, Cold-Reset, Reset, ModeIn, and Mode IDT79RC5000  Reference ManualClock.

Notes

**Chapter 15, “Clock Interface and Standby Mode,”** describes how the processor bases all internal and external clocking on the single SysClock input signal.

## Revision History

**April 1998**: Publish Version 1.1 of the manual.

**October 1999**: Added Set BadVA box to flow diagram in Figure 5.19.

**September 26, 2000**: Added explanation to Cache Error Register (27) section and Cache Error Exception section in Chapter 5 that when read responses (cached or uncached) are returned with bad parity, the Cache Error Exception is taken. Also in Cache Error Exception section, changed BEV = 0 to BEV = 1 for vector 0XFFFF FFFF BFC0 0300.

# Table of Contents

Notes

## About This Manual

## 1 Overview

## 2 Central Processing Unit (CPU)

Notes

## 3 Floating-Point Unit (FPU)

## 4 Instruction Pipeline

Notes

## 5 Integer (CPU) Exceptions

Notes

## 6 Floating-Point (FPU) Exceptions

## 7 Memory Management Unit

Notes

Notes

Notes

Notes

# List of Tables

## Notes

Notes

# List of Figures

**Notes**

Notes

Notes

## Notes

# Overview

Notes

## Performance

The IDT79RV5000™ microprocessor (RV5000) is a 64-bit dual-issue RISC processor that serves many performance-critical applications. The processor's 260 Dhrystone MIPS performance and 4Gbytes/sec aggregate bandwidth makes it ideal for embedded applications, such as high-end internetworking systems, color printers, and graphics terminals, as well as low-cost general computing with special emphasis on floating-point operations and memory management. For such applications it offers:

- *A high-performance upgrade path for existing embedded customers in the internetworking, office automation and visualization markets.*
- *Significant improvements in floating-point performance in a moderately priced chip.*
- *Improved desktop-system memory hierarchy through the implementation of large primary instruction and data caches (32KB each) and an on-chip secondary cache controller.*
- *Improved performance through the use of the MIPS-IV instruction-set architecture (ISA).*

## Upward Compatibility

The RV5000 conforms to the MIPS-IV Instruction Set Architecture (ISA) and provides complete upward application-software compatibility with the IDT79R3xxx and IDT79R4xxx families of microprocessors. An array of operating systems and development tools facilitates rapid development of systems that support thousands of application programs.

The R5V000 is a true 64-bit processor but is also fully compatible with 32-bit operating systems and applications. The RV5000 enables 32-bit applications to effortlessly access 64-bit compute power. For embedded applications, the power and bandwidth of 64-bit data types can be used without the memory expansion of 64-bit addressing.

## Block Diagram

Figure 1.1 on page 1-2 is a block diagram of the RV5000's functional units.

## Notes

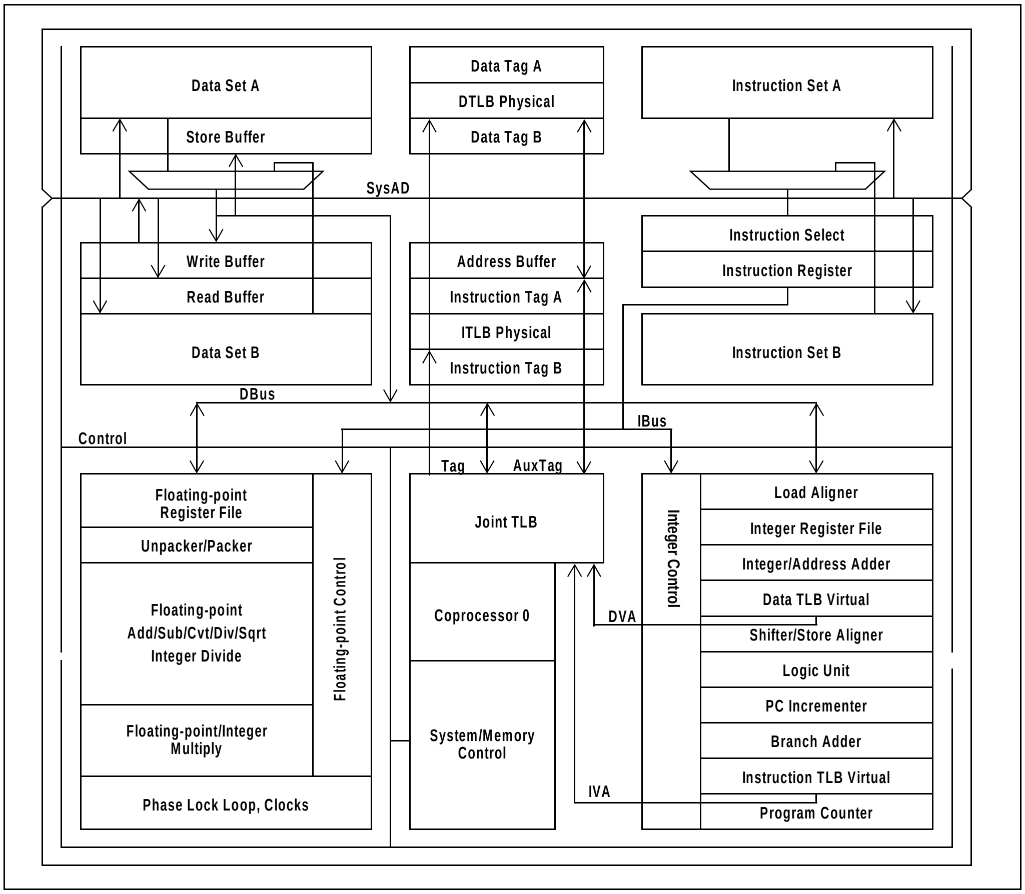


Figure 1.1 RV5000 Block Diagram

# Features

- *True 64-bit Microprocessor*
  - *64-bit integer (CPU) operations*
  - *64-bit floating-point (FPU) IEEE-754 operations*
  - *64-bit registers*
  - *64-bit virtual addresses*
- *7 Execution Resources*
  - *Integer ALU, with bypassing*
  - *Integer multiply/divide unit*
  - *Floating-point ALU, pipelined to allow single-cycle repeat rate for single-precision operations*
  - *Floating-point divide/square-root unit*
  - *Load unit*
  - *Store unit*
  - *Branch unit*
- *Selectable Frequencies*
  - *Bus-to-pipeline frequency ratios of 2, 3, 4, 5, 6, 7 or 8*
- *High Performance*
  - *260 Dhrystone MIPS*
  - *4Gbytes/sec aggregate bandwidth at 200MHz pipeline clock frequency*
- *Efficient Memory Hierarchy*
  - *32KB two-way set associative instruction cache*
  - *32KB two-way set associative data cache*
  - *On-chip secondary cache controller*
  - *64Gbyte physical address space*
  - *1 terabyte ($2^{40}$) maximum user process size*
  - *Flexible MMU with 48-entry TLB*
  - *1.6Gbytes/sec cache bandwidth (each cache) at 200MHz pipeline frequency*

Notes

- *Software Compatibility*
  - *MIPS IV 64-bit instruction set, including CP1 and CP1X functional units*
  - *Software-compatible with R3xxx and R4xxx families*
- *Compatible with Multiple Operating Systems*
  - *JMI C-executive - Windows® CE*
  - *VX Works*
  - *OS9*
  - *PSOS*
- *Development Tools*
  - *Cross compilers*
  - *Logic models*
  - *Logic analyzer support*
- *Low-Power Operation*
  - *3.3V power supply*
  - *25mW/MHz internal power dissipation (5W @ 200MHz, 3.3V)*
  - *Active power management, including WAIT operation*
- *272-pin SuperBGA Package*

# Instruction Pipeline

The RV5000 has a dual-issue, five-stage instruction pipeline that operates at a multiple of the frequency of the input system clock. The pipeline has two parallel paths, one for integer (CPU) instructions and the other floating-point (FPU) instructions. Each stage in the CPU-instruction path takes one processor clock.

## Dual Issue

Dual-issue instruction pairing in a given clock can consist of:

- *1 floating-point ALU instruction*
- *1 instruction of any other type*

These two instruction classes are pre-decoded as they are brought on-chip. The pre-decoded information is stored in the instruction cache. If there are no pending resource conflicts, the RV5000 can issue one instruction per class per pipeline clock cycle. Long-latency operations, such as floating-point DIV or SQRT, or integer (CPU) multiply, can slow the issue of instructions. The RV5000 does not perform out-of-order or speculative execution; instead, the pipeline slips until the required resource becomes available.

There are no alignment restrictions on dual-issue instruction pairs. However, since the RV5000 performs aligned fetches, at two instructions per cycle from the instruction cache, compilers should attempt to align branch targets to allow dual-issue on the first target cycle, in order to optimize performance.

## Integer (CPU) Pipeline

The RV5000 implements a traditional five-stage pipeline, as shown in Figure 1.2 and Table 1.1:

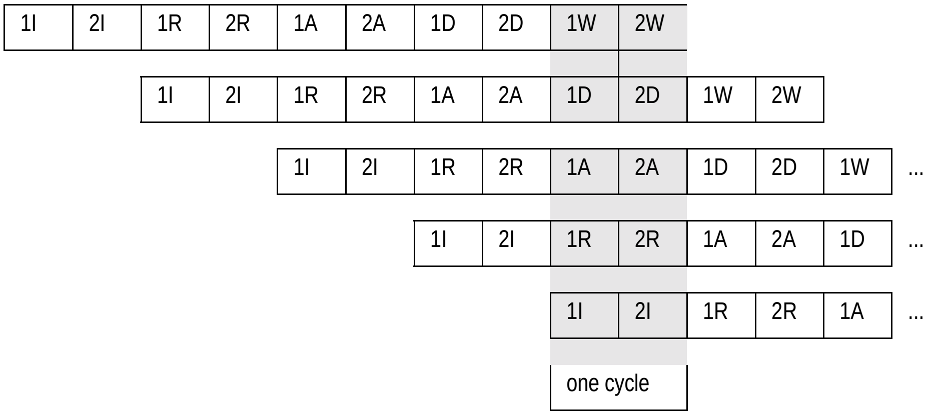

1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
1I 2I 1R 2R 1A 2A 1D 2D 1W ...
1I 2I 1R 2R 1A 2A 1D ...
1I 2I 1R 2R 1A ...
one cycle

**Figure 1.2 Integer (CPU) Pipeline**

Notes

1I - Instruction Fetch, Phase One

2I - Instruction Fetch, Phase Two

1R - Register Read, Phase One

2R - Register Read, Phase Two

1A - Execution, Phase One

2A - Execution, Phase Two

1D - Data Load/Store, Phase One

2D - Data Load/Store, Phase Two

1W - Write Back, Phase One

2W - Write Back, Phase Two

| Stage | Function |
|---|---|
| 1I to 1R | Instruction-cache access |
| 2I | Instruction virtual-to-physical address translation |
| 2R | Register-file read, bypass calculation, instruction decode, branch-address calculation |
| 1A | Issue or slip decision, data virtual-address calculation, branch decision |
| 1A to 2A | Integer add, logical, shift |
| 2A | Store align |
| 2A to 2D | Data-cache access and load align |
| 1D to 2D | Data virtual-to-physical address translation |
| 1W | Resolve exceptions |
| 2W | Register-file write |

**Table 1.1 Key to Integer Pipeline**

Typical integer (CPU) execution latencies are shown in Table 1.2. The RV5000's short pipeline keeps the load and branch latencies low. The caches allow any combination of loads and stores to execute in back-to-back cycles without requiring pipeline slips or stalls if the operation hits in the cache.

| Operation | Latency | Repeat |
|---|---|---|
| Load | 2 | 1 |
| Store | 2 | 1 |
| MULT/MULTU | 5 | 4 |
| DMULT/DMULTU | 9 | 8 |
| DIV/DIVU | 36 | 36 |
| DDIV/DDIVU | 68 | 68 |
| Other Integer ALU | 1 | 1 |
| Branch | 2 | 2 |
| Jump | 2 | 2 |

**Table 1.2 Example Integer (CPU) Instruction Latencies**

Notes

### Floating-Point Unit (FPU) Pipeline

The on-chip floating-point unit (FPU) coprocessor includes a 64-bit floating-point register file. The FPU forms a seamless interface with the integer (CPU) pipeline, decoding and executing instructions in parallel with the CPU.

The FPU supports single- and double-precision arithmetic, as specified in the IEEE Standard 754. It also supports fully precise floating-point exceptions while allowing both overlapped and pipelined operations. Precise exceptions are extremely important in mission-critical environments and are highly desirable for debugging in any environment.

As described above, two instructions can be issued in a given clock if one is a floating-point ALU instruction and the another is any type other than floating-point ALU. Figure 1.3 shows a simplified diagram of this dual-issue mechanism.

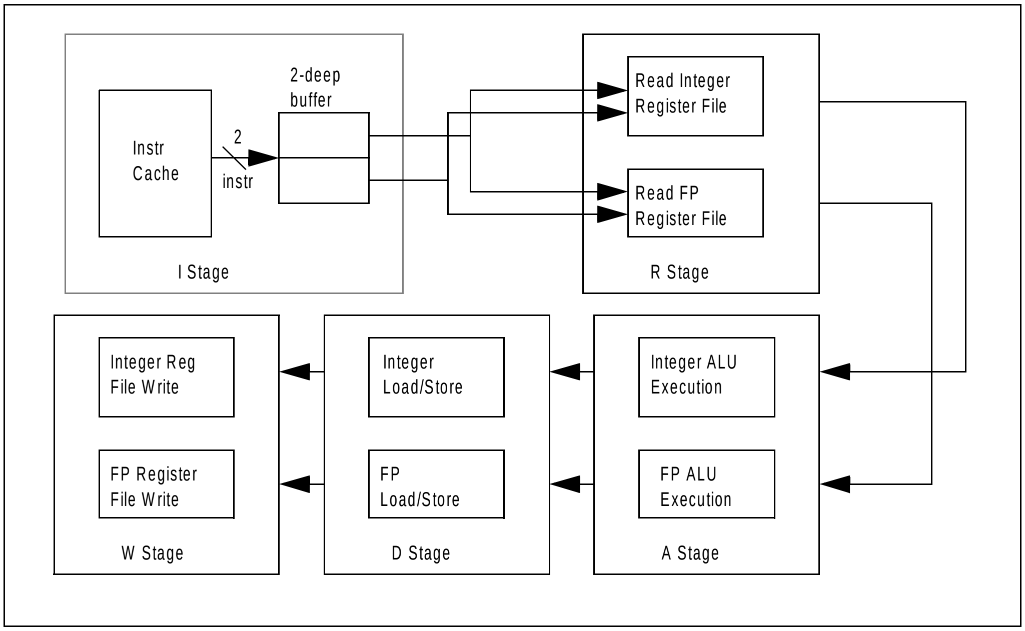


**Figure 1.3 Dual-Issue Mechanism, Showing CPU and FPU Pipelines**

## Virtual-to-Physical Address Mapping

The RV5000 provides three modes of operation:

- *user mode*
- *supervisor mode*
- *kernel mode*

This mechanism is available to system software to provide a secure environment for user processes. Bits in a status register determine the mode of operation. When operating in the kernel mode, up to four distinct virtual-address spaces totalling 1024Gbytes are simultaneously available and are differentiated by the high-order bits of the virtual address. The RV5000 also supports a supervisor mode in which the virtual address space is 256Gbytes, divided into three regions based on the high-order bits of the virtual address. When the RV5000 uses 64-bit virtual addresses, the address space layouts are an upward-compatible extension of the 32-bit virtual address space layout.

### Joint TLB

For fast virtual-to-physical address decoding, the RV5000 uses a fully associative joint TLB which maps 96 virtual pages to their corresponding physical addresses. The TLB is organized as 48 pairs of even-odd entries, and maps a virtual address and address space identifier into the 64Gbyte physical address space.

Notes

Two mechanisms assist in controlling the amount of mapped space, and the replacement characteristics of various memory regions. First, the page size can be configured, on a per-entry basis, at 4Kbytes, 16Kbytes, 64Kbytes, 256Kbytes, 1Mbyte, 4Mbytes, or 16Mbytes. A CP0 register is loaded with the page size of a mapping, and that size is 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 using only one TLB entry.

The second mechanism controls the replacement algorithm when a TLB miss occurs. The RV5000 implements a random replacement algorithm to select a TLB entry to be written with a new mapping. However, the processor provides a mechanism whereby a system-specific number of mappings can be locked into the TLB, and thus avoid being randomly replaced. This facilitates the design of real-time systems, by allowing deterministic access to critical software.

The joint TLB also contains information to control the cache coherency protocol for each page. Specifically, each page has attribute bits to determine whether the coherency algorithm is: uncached, non-coherent write-back, non-coherent write-through write-allocate, non-coherent write-through no write-allocate, sharable, exclusive, or update. Non-coherent write-back is typically used for both code and data on the RV5000. The write-through modes support more efficient frame-buffer accesses than the R4000 family. The coherent modes are supported for R4000 compatibility and generate different transaction types on the system interface; however, cache coherency is not supported.

# Cache

The RV5000 incorporates on-chip instruction and data caches that can be accessed in a single processor cycle. The processor also includes an on-chip secondary cache controller for simple interfacing to large, high-speed second-level cache SRAM.

Both of the on-chip primary caches are 32KB in size, two-way set associative, virtually indexed, and physically tagged. Because the caches are virtually indexed, virtual-to-physical address translation occurs in parallel with the cache access. Each cache has its own 64-bit data path and can be accessed in parallel each pipeline cycle. The cache subsystem provides the integer unit (CPU) and floating-point unit (FPU) with an aggregate bandwidth of 3.2Gbytes per second at a pipeline clock frequency of 200MHz.

## Instruction Cache

The instruction cache is 64-bits wide. Cache lines are eight instructions (32 bytes). Instruction fetches are 8 bytes per cycle, for a peak instruction-cache bandwidth of 1.6Gbytes/sec @ 200MHz. The instruction cache is protected with word parity. The tag holds a 24-bit physical address and valid bit, and is parity protected.

## Data Cache

The data cache is 64-bits wide. Cache lines are 32 bytes. Data loads are 8 bytes per cycle, for a peak data-cache bandwidth of 1.6Gbytes/sec @ 200MHz (in addition to the 1.6Gbytes/sec instruction-cache bandwidth). The data cache is protected with byte parity and its tag is protected with a single parity bit. It is virtually indexed and physically tagged to allow simultaneous address translation and data-cache accesses. The normal write policy is writeback. Software can, however, select write-through on a per-page basis, such as for frame buffers.

The data cache has an associated store buffer. When the RV5000 executes a store instruction, this single-entry buffer gets written with the store data while the tag comparison is performed. If the tag matches, the data is written into the data cache in the next cycle that the data cache is not accessed (the next non-load cycle). The store buffer allows the RV5000 to execute a store every processor cycle and to perform back-to-back stores without penalty.

Notes

### Write buffer

Writes to external memory, whether cache-miss writebacks or stores to uncached or write-through addresses, use the on-chip write buffer. The write buffer holds up to four 64-bit address and data pairs or one cache line to be written back. The entire buffer is used for a data-cache writeback and allows the processor to proceed in parallel with memory update.

### Clocks

The RV5000 uses the system interface clock as its input clock. The pipeline speed is derived from this clock using a PLL to multiply up the input reference. It is assumed that the system designer manages the system clock distribution to fit the needs of the system. Thus, the RV5000 does not output a system reference clock, but rather operates in synchronization with the input clock.

The RV5000 outputs one low-frequency reference clock: the Mode Clock. This clock operates at 1/256 the rate of the input clock, and it is used to clock-in the serial initialization stream during reset.

## System Interface

The RV5000 supports a 64-bit multiplexed system interface that is compatible with the R4xxx system interface. The interface consists of a 64-bit address/data bus with 8 check bits and a 9-bit command bus. In addition, there are 8 handshake signals and 6 interrupt inputs. The interface has a simple timing specification and is capable of transferring data between the processor and memory at a peak rate of 800Mbytes/sec at 100MHz. Figure 1.4 shows a typical system using the RV5000.

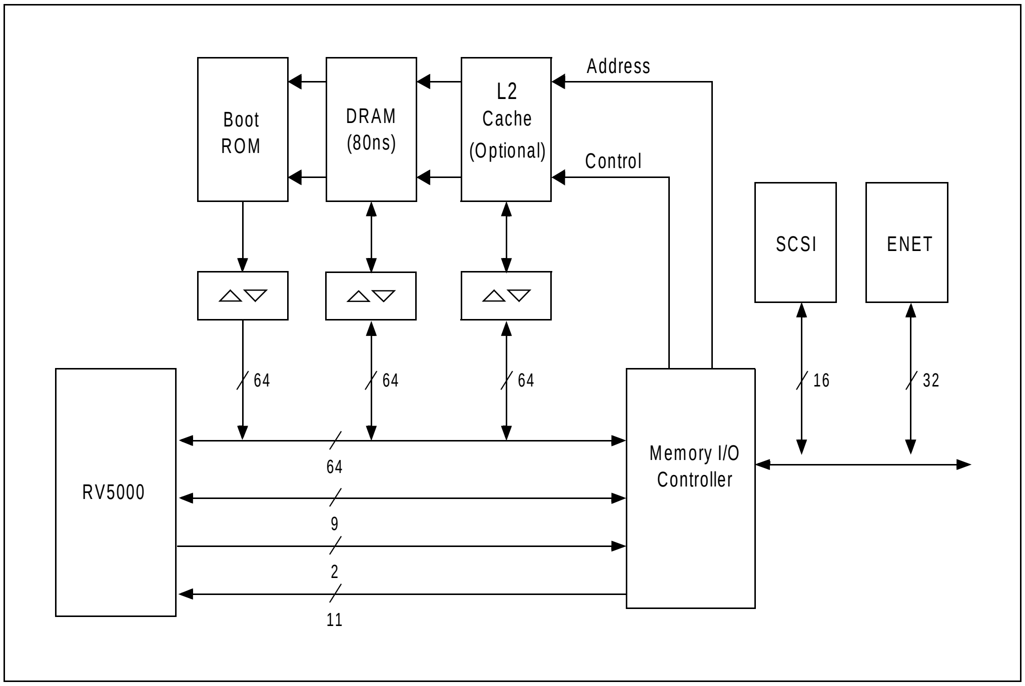


**Figure 1.4 Typical RV5000 System Block Diagram**

Notes

# Central Processing Unit (CPU)

Notes

## Introduction

In the MIPS instruction-set architecture, the central processing unit (CPU) executes integer and system instructions, and the floating-point unit (FPU) coprocessor executes floating-point instructions. This chapter describes only the CPU. Chapter 3 describes with the FPU.

## CPU Registers

The R5000 integer unit has thirty-two general purpose registers. These registers are used for scalar integer operations and address calculation. The register file consists of two read ports and one write port, and is fully bypassed to minimize operation latency in the pipeline. Figure 2.1 shows the R5000 CPU registers.

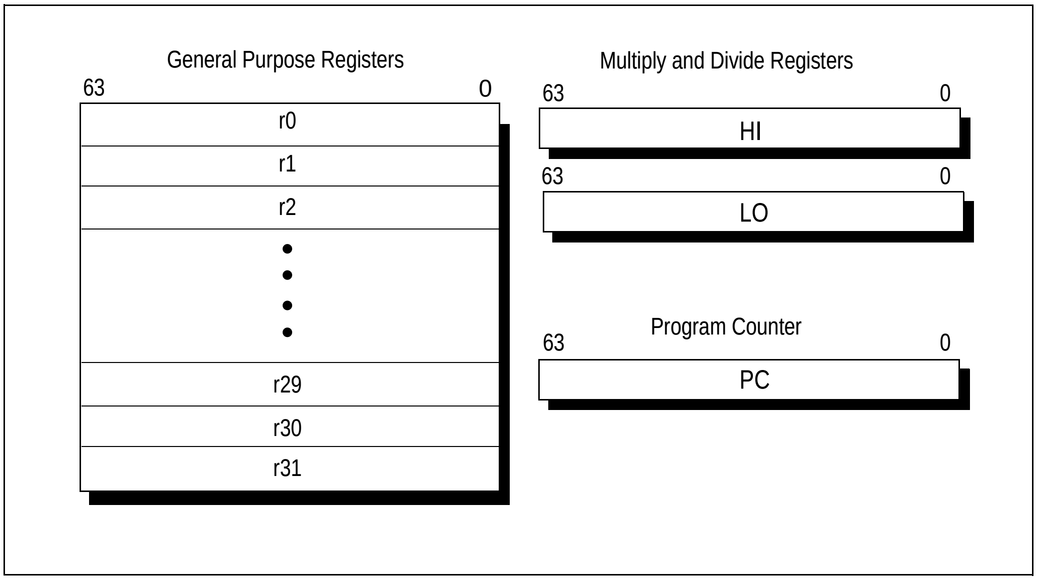


**Figure 2.1 R5000 CPU Registers**

Two of the CPU general purpose registers have assigned functions:

- *r0 is hardwired to a value of zero, and can be used as the target register for any instruction whose result is to be discarded. r0 can also be used as a source when a zero value is needed.*
- *r31 is used as an implicit return destination address register by the JAL series of instructions.*

The CPU has three special purpose registers:

- *PC — Program Counter register*
- *HI — Multiply and Divide register, higher result*
- *LO — Multiply and Divide register, lower result*

The two Multiply and Divide registers (HI, LO) store:

- *the product of integer multiply operations, or*
- *the quotient (in LO) and remainder (in HI) of integer divide operations.*

The R5000 has no Program Status Word (PSW) register as such; this is covered by the *Status* and *Cause* registers incorporated within the System Control Coprocessor (CP0), as described in the next section, below.

Notes

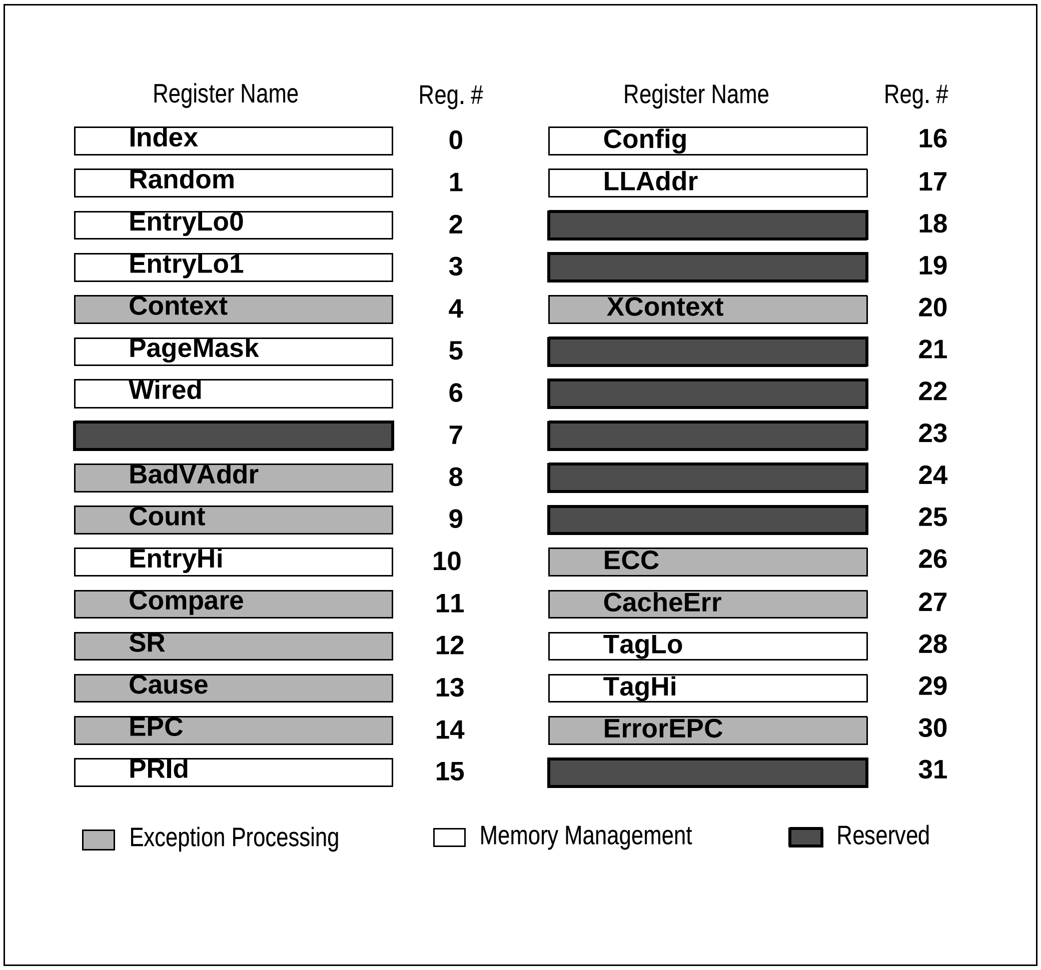
## Coprocessors (CP0-CP2) and Their Registers

The MIPS IV instruction set architecture defines three coprocessors, designated CP0, CP1, and CP2. The R5000 implements the first two:

- *Coprocessor 0 (CP0) supports the virtual memory system and exception handling. CP0 is also referred to as the System Control Coprocessor, and is described below.*
- *Coprocessor 1 (CP1) implements the MIPS floating-point instruction set and is used by the FPU. The registers associated with CP1 are described in Chapter 3.*
- *Coprocessor 2 (CP2) is reserved for future use.*

The registers associated with CP0 are shown in Figure 2.2 and described in Table 2.1. CP0 translates virtual addresses into physical addresses and manages exceptions and transitions between kernel, supervisor, and user states. CP0 also controls the cache subsystem, controls power management, and provides diagnostic control and error recovery facilities. Access to reserved or undefined CP0 register results are undefined. An exception may or may not result.

Power management is implemented in CP0 with the standby mode, which reduces power consumption by the CPU core. The standby mode is entered by executing the WAIT instruction with the SysAD bus idle, and it is exited by any interrupt.



| Register Name | Reg. # | Register Name | Reg. # |
|---|---|---|---|
| Index | 0 | Config | 16 |
| Random | 1 | LLAddr | 17 |
| EntryLo0 | 2 | ■ | 18 |
| EntryLo1 | 3 | ■ | 19 |
| Context | 4 | XContext | 20 |
| PageMask | 5 | ■ | 21 |
| Wired | 6 | ■ | 22 |
| ■ | 7 | ■ | 23 |
| BadVAddr | 8 | ■ | 24 |
| Count | 9 | ■ | 25 |
| EntryHi | 10 | ECC | 26 |
| Compare | 11 | CacheErr | 27 |
| SR | 12 | TagLo | 28 |
| Cause | 13 | TagHi | 29 |
| EPC | 14 | ErrorEPC | 30 |
| PRId | 15 | ■ | 31 |

▒ Exception Processing □ Memory Management ■ Reserved

**Figure 2.2 CP0 Registers**

Notes

| Number | Register | Description |
|---|---|---|
| 0 | Index | Programmable pointer into TLB array |
| 1 | Random | Pseudo-random pointer into TLB array *(read only)* |
| 2 | EntryLo0 | Low half of TLB entry for even virtual page (VPN) |
| 3 | EntryLo1 | Low half of TLB entry for odd virtual page (VPN) |
| 4 | Context | Pointer to kernel virtual page table entry (PTE) for 32-bit address spaces |
| 5 | PageMask | TLB page mask |
| 6 | Wired | Number of wired TLB entries |
| 7 | — | Reserved |
| 8 | BadVAddr | Bad virtual address |
| 9 | Count | Timer count |
| 10 | EntryHi | High half of TLB entry |
| 11 | Compare | Timer compare |
| 12 | SR | Status register |
| 13 | Cause | Cause of last exception |
| 14 | EPC | Exception program counter |
| 15 | PRId | Processor revision identifier |
| 16 | Config | Configuration register |
| 17 | LLAddr | Load linked address |
| 18 - 19 | — | Reserved |
| 20 | XContext | Pointer to kernel virtual PTE table for 64-bit address spaces |
| 21–25 | — | Reserved |
| 26 | ECC | Secondary-cache error checking and correcting (ECC) and primary parity |
| 27 | CacheErr | Cache error and status register |
| 28 | TagLo | Cache tag register |
| 29 | TagHi | Cache tag register |
| 30 | ErrorEPC | Error exception program counter |
| 31 | — | Reserved |

**Table 2.1 System Control Coprocessor (CPO) Register Definitions**

## CPU Data Formats and Addressing

The R5000 processor uses four data formats: a 64-bit doubleword, a 32-bit word, a 16-bit halfword, and an 8-bit byte. Byte ordering within the halfword, word, and doubleword data formats can be configured in either big-endian or little-endian order. Endianness refers to the location of byte 0 within the multi-byte data structure. Figures 1.4 and 1.5 show the ordering of bytes within words and the ordering of words within multiple-word structures for the big-endian and little-endian conventions.

When the R5000 processor is configured as a big-endian system, byte 0 is the most-significant (left-most) byte, thereby providing compatibility with MC 68000 and IBM 370 conventions. Figure 2.3 illustrates this configuration.

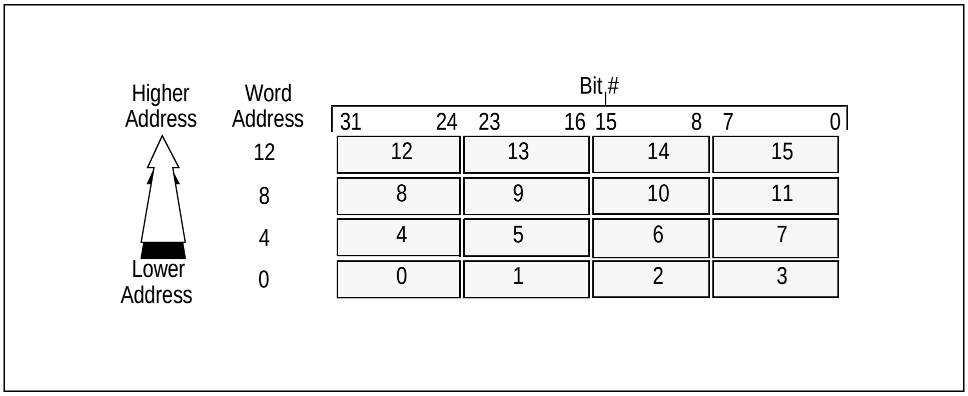


**Figure 2.3 Big-Endian Byte Ordering**

When configured as a little-endian system, byte 0 is always the least-significant (right-most) byte, which is compatible with x86 and DEC VAX conventions. Figure 2.4 illustrates this configuration.

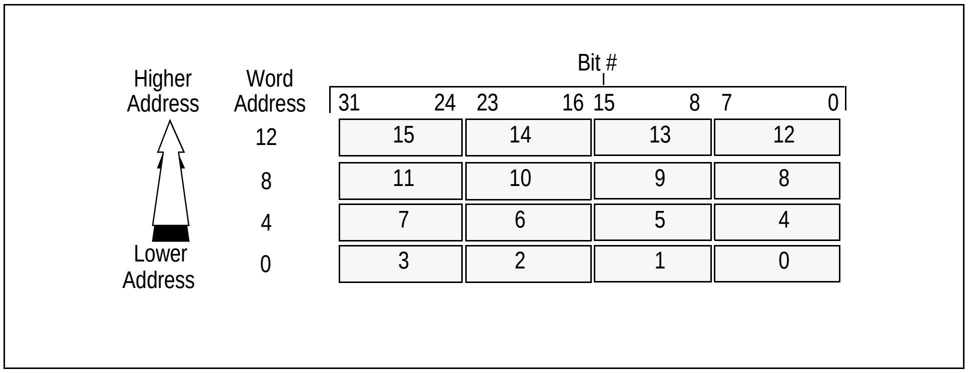


**Figure 2.4 Little-Endian Byte Ordering**

In this text, bit 0 is always the least-significant (right-most) bit; thus, bit designations are always little-endian (although no instructions explicitly designate bit positions within words).

Figures 1.6 and 1.7 show little-endian and big-endian byte ordering in doublewords.

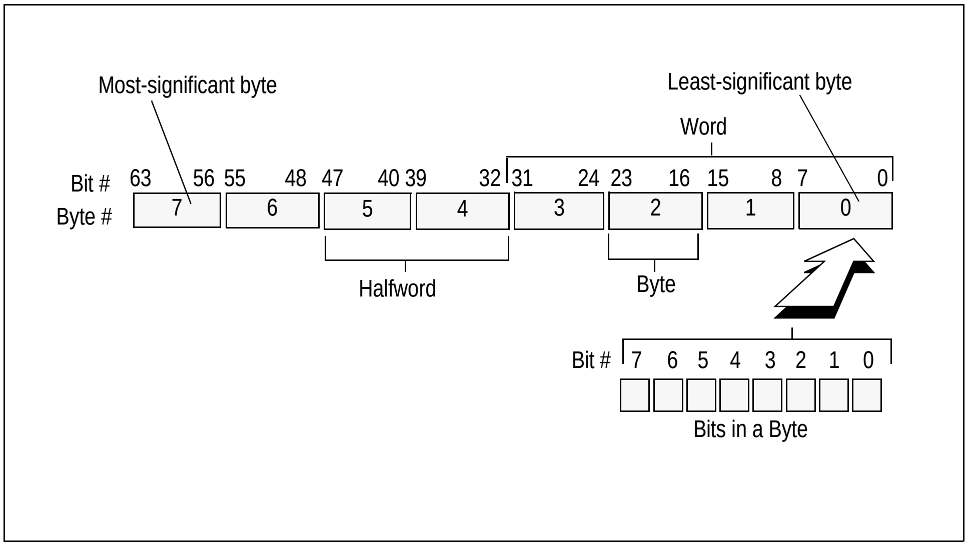


**Figure 2.5 Little-Endian Data in a Doubleword**

Notes

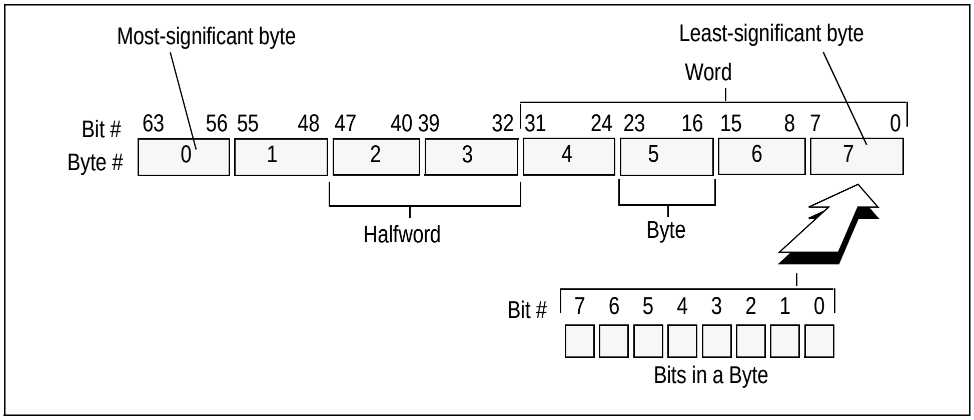


**Figure 2.6 Big-Endian Data in a Doubleword**

The CPU uses byte addressing for halfword, word, and doubleword accesses with the following alignment constraints:

- *Halfword accesses must be aligned on an even byte boundary (0, 2, 4...).*
- *Word accesses must be aligned on a byte boundary divisible by four (0, 4, 8...).*
- *Doubleword accesses must be aligned on a byte boundary divisible by eight (0, 8, 16...).*

The following special instructions load and store words that are not aligned on 4-byte (word) or 8-word (doubleword) boundaries:

| | | | |
|---|---|---|---|
| LWL | LWR | SWL | SWR |
| LDL | LDR | SDL | SDR |

These instructions are used in pairs to provide addressing of misaligned words. Addressing misaligned data incurs one additional instruction cycle over that required for addressing aligned data. This extra cycle is the result of an extra instruction for the “pair” (e.g., LWL and LWR form a pair). Also note that the CPU moves the unaligned data at the same rate as a hardware mechanism.

Figures 1.8 and 1.9 show the access of a misaligned word that has byte address 3.

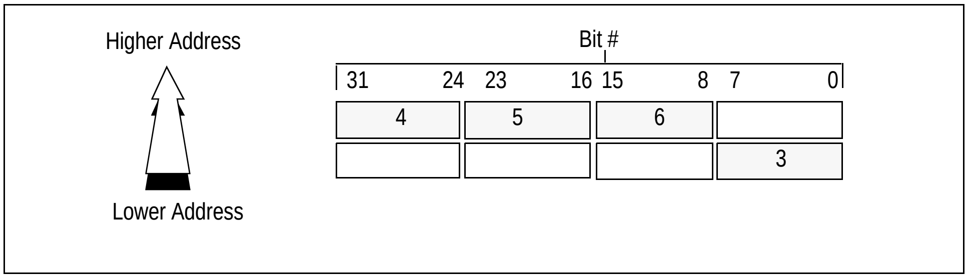


**Figure 2.7 Big-Endian Misaligned Word Addressing**

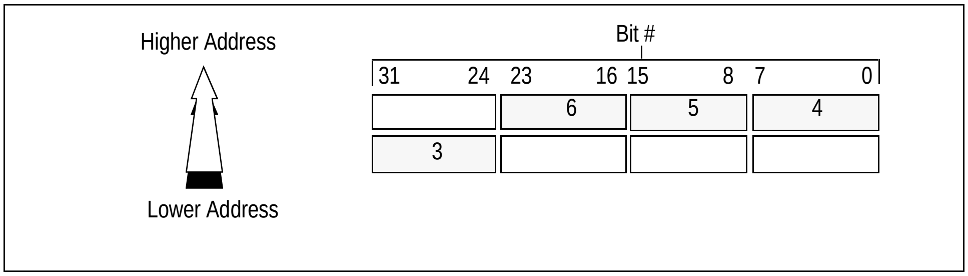


**Figure 2.8 Little-Endian Misaligned Word Addressing**

Notes

# CPU Instruction Set Summary

The R5000 executes the MIPS IV instruction set, which is a superset of the MIPS III instruction set and is backward-compatible with MIPS III. Each CPU instruction consists of a single 32-bit word, aligned on a word boundary. There are three instruction formats—immediate (I-type), jump (J-type), and register (R-type). 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.

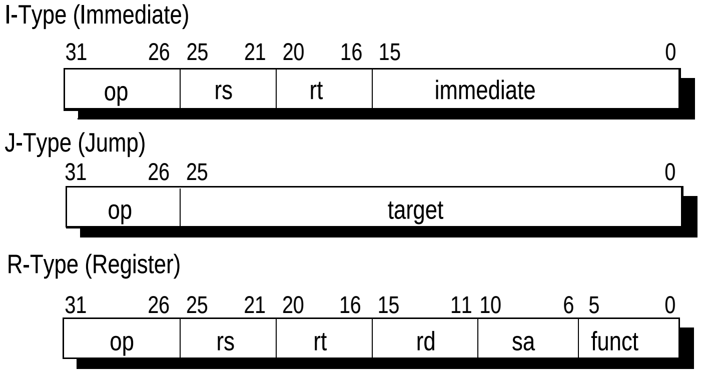
Table 2.2 gives an overview of R5000 CPU-instruction latencies. A summary of the MIPS IV instruction set additions is given in the remainder of this section, along with a brief explanation of each instruction. For more information on the MIPS IV instruction set, refer to the IDT MIPS Microprocessor Family Software Manual.

| Instruction Group | Latency | Repeat |
|---|---|---|
| Arithmetic and Logical | 1 | 1 |
| Shift | 1 | 1 |
| Load | 2 | 1 |
| Store | N/A | 1 |
| Multiply (32-bit) | 5 | 4 |
| Multiply (64-bit) | 9 | 8 |
| Divide (32-bit) | 36 | 36 |
| Divide (64-bit) | 68 | 68 |

**Table 2.2 Some Integer-Instruction Latencies**

## Instruction Formats

The three types of instruction types are shown in Figure 2.9.



| | |
|---|---|
| op | 6-bit operation code |
| rs | 5-bit source register specifier |
| rt | 5-bit target (source/destination) register or branch condition |
| immediate | 16-bit immediate value, branch displacement or address displacement |
| target | 26-bit jump target address |
| rd | 5-bit destination register specifier |
| sa | 5-bit shift amount |
| funct | 6-bit function field |

**Figure 2.9 Instruction Formats**

Notes

### Instruction Types

The CPU instruction set includes the following types of instructions:

- ***Load and Store*** *instructions move data between memory and general registers. They are all immediate (I-type) instructions, since the only addressing mode supported is base register plus 16-bit, signed immediate offset.*
- ***Computational*** *instructions perform arithmetic, logical, shift, multiply, and divide operations on values in registers. They include register (R-type, in which both the operands and the result are stored in registers) and immediate (I-type, in which one operand is a 16-bit immediate value) formats.*
- ***Jump and Branch*** *instructions change the control flow of a program. Jumps are always made to a paged, absolute address formed by combining a 26-bit target address with the high-order bits of the Program Counter (J-type format) or register address (R-type format). Branches have 16-bit offsets relative to the program counter (I-type). Jump And Link instructions save their return address in register 31.*
- ***Coprocessor*** *instructions perform operations in the coprocessors. Coprocessor load and store instructions are I-type.*
- ***Coprocessor 0*** *(system coprocessor) instructions perform operations on CP0 registers to control the memory management and exception handling facilities of the processor and the standby mode for power management.*
- ***Special*** *instructions perform system calls and breakpoint operations. These instructions are always R-type.*
- ***Exception*** *instructions cause a branch to the general exception-handling vector based upon the result of a comparison. These instructions occur in both R-type (both the operands and the result are registers) and I-type (one operand is a 16-bit immediate value) formats.*

### Load and Store Instructions

Load and store are immediate (I-type) instructions that move data between memory and the general registers. The only addressing mode that load and store instructions directly support is *base register plus 16-bit signed immediate offset*.

Table 2.3 lists the load and store instructions.

Notes

| OpCode | Description | MIPS ISA Level |
|---|---|---|
| LB | Load Byte | I |
| LBU | Load Byte Unsigned | I |
| LD | Load Doubleword | III |
| LDL | Load Doubleword Left | III |
| LDR | Load Doubleword Right | III |
| LH | Load Halfword | I |
| LHU | Load Halfword Unsigned | I |
| LL | Load Linked | II |
| LLD | Load Linked Doubleword | III |
| LW | Load Word | I |
| LWL | Load Word Left | I |
| LWR | Load Word Right | I |
| LWU | Load Word Unsigned | III |
| PREF[1] | Prefetch, Register + Offset | IV |
| PREFX[1] | Prefetch Indexed, Register + Register | IV |
| SB | Store Byte | I |
| SC | Store Conditional | II |
| SCD | Store Conditional Doubleword | III |
| SD | Store Doubleword | III |
| SDL | Store Doubleword Left | III |
| SDR | Store Doubleword Right | III |
| SH | Store Halfword | I |
| SW | Store Word | I |
| SWL | Store Word Left | I |
| SWR | Store Word Right | I |
| SYNC | Sync | II |

[1] Prefetch is not implemented in the R5000; these instructions are no-ops.

**Table 2.3 Load and Store Instructions**

### Scheduling a Load Delay Slot

A load instruction that does not allow its result to be used by the instruction immediately following is called a *delayed load instruction*. The instruction slot immediately following this delayed load instruction is referred to as the *load delay slot*.

In the R5000, the instruction immediately following a load instruction can reference the contents of the loaded register, but hardware interlocks insert additional real cycles. Consequently, scheduling load delay slots can be desirable, both for performance and R-Series processor compatibility. However, the scheduling of load delay slots is not required.

Notes

### Defining Access Types

*Access type* indicates the size of an R5000 data item to be loaded or stored. The access type is determined by the load/store instruction opcode. Regardless of access type or byte ordering (endianness), the address specifies the low-order byte in the addressed field. For a big-endian configuration, the low-order byte is the most-significant byte; for a little-endian configuration, the low-order byte is the least-significant byte.

The access type, together with the three low-order bits of the address, define the bytes accessed within the addressed doubleword (shown in Table 2.4). Only the combinations shown in Table 2.4 are permissible; other combinations cause address-error exceptions.

| **Access Type Mnemonic (*Value*)** | **Low Order Address Bits** | | | **Bytes Accessed** | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | **Big endian (63-----------31------------0) Byte** | | | | | | | | **Little endian (63-----------31------------0) Byte** | | | | | | | |
| | **2** | **1** | **0** | | | | | | | | | | | | | | | | |
| Doubleword (*7*) | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Septibyte (*6*) | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | | | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| | 0 | 0 | 1 | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | |
| Sextibyte (*5*) | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 4 | 5 | | | | | 5 | 4 | 3 | 2 | 1 | 0 |
| | 0 | 1 | 0 | | | 2 | 3 | 4 | 5 | 6 | 7 | 7 | 6 | 5 | 4 | 3 | 2 | | |
| Quintibyte (*4*) | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 4 | | | | | | | 4 | 3 | 2 | 1 | 0 |
| | 0 | 1 | 1 | | | | 3 | 4 | 5 | 6 | 7 | 7 | 6 | 5 | 4 | 3 | | | |
| Word (*3*) | 0 | 0 | 0 | 0 | 1 | 2 | 3 | | | | | | | | | 3 | 2 | 1 | 0 |
| | 1 | 0 | 0 | | | | | 4 | 5 | 6 | 7 | 7 | 6 | 5 | 4 | | | | |
| Triplebyte (*2*) | 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 (*1*) | 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 | 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 | | | | | | | |

Table 2.4 Byte Access Within a Doubleword

### 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*

Notes

- *shift*
- *multiply*
- *divide*

These operations fit in the following four categories of computational instructions:

- *ALU Immediate instructions*
- *three-Operand Register-Type instructions*
- *shift instructions*
- *multiply and divide instructions*

Table 2.5 through Table 2.8 list the computational instructions.

| OpCode | Description | MIPS ISA Level |
|---|---|---|
| ADDI | Add Immediate | I |
| ADDIU | Add Immediate Unsigned | I |
| ANDI | AND Immediate | I |
| DADDI | Doubleword Add Immediate | III |
| DADDIU | Doubleword Add Immediate Unsigned | III |
| LUI | Load Upper Immediate | I |
| ORI | OR Immediate | I |
| SLTI | Set on Less Than Immediate | I |
| SLTIU | Set on Less Than Immediate Unsigned | I |
| XORI | Exclusive OR Immediate | I |

**Table 2.5 Arithmetic Instructions (ALU Immediate)**

| OpCode | Description | MIPS ISA Level |
|---|---|---|
| ADD | Add | I |
| ADDU | Add Unsigned | I |
| AND | AND | I |
| DADD | Doubleword Add | III |
| DADDU | Doubleword Add Unsigned | III |
| DSUB | Doubleword Subtract | III |
| DSUBU | Doubleword Subtract Unsigned | III |
| NOR | NOR | I |
| OR | OR | I |
| SLT | Set on Less Than | I |
| SLTU | Set on Less Than Unsigned | I |
| SUB | Subtract | I |
| SUBU | Subtract Unsigned | I |
| XOR | Exclusive OR | I |

**Table 2.6 Arithmetic (3-Operand, R-Type)**

Notes

| OpCode | Description | MIPS ISA Level |
|---|---|---|
| DSLL | Doubleword Shift Left Logical | III |
| DSRL | Doubleword Shift Right Logical | III |
| DSRA | Doubleword Shift Right Arithmetic | III |
| DSLLV | Doubleword Shift Left Logical Variable | III |
| DSRLV | Doubleword Shift Right Logical Variable | III |
| DSRAV | Doubleword Shift Right Arithmetic Variable | III |
| DSLL32 | Doubleword Shift Left Logical + 32 | III |
| DSRL32 | Doubleword Shift Right Logical + 32 | III |
| DSRA32 | Doubleword Shift Right Arithmetic + 32 | III |
| SLL | Shift Left Logical | I |
| SLLV | Shift Left Logical Variable | I |
| SRA | Shift Right Arithmetic | I |
| SRAV | Shift Right Arithmetic Variable | I |
| SRL | Shift Right Logical | I |
| SRLV | Shift Right Logical Variable | I |

Table 2.7 Shift Instructions

| OpCode | Description | MIPS ISA Level |
|---|---|---|
| DIV | Divide | I |
| DIVU | Divide Unsigned | I |
| DMULT | Doubleword Multiply | III |
| DMULTU | Doubleword Multiply Unsigned | III |
| DDIV | Doubleword Divide | III |
| DDIVU | Doubleword Divide Unsigned | III |
| MFHI | Move From HI | I |
| MTHI | Move To HI | I |
| MFLO | Move From LO | I |
| MOVF | Move Conditional on Condition Code False | IV |
| MOVN | Move on Register Not Equal to Zero | IV |
| MOVT | Move Conditional on Condition Code True | IV |
| MOVZ | Move on Register Equal to Zero | IV |
| MTLO | Move To LO | I |
| MULT | Multiply | I |
| MULTU | Multiply Unsigned | I |

Table 2.8 Multiply and Divide Instructions

Notes

### 64-bit Operations

When operating in 64-bit mode, 32-bit operands must be sign-extended. 32-bit operand opcodes include all non-doubleword operations, such as ADD, ADDU, SUB, SUBU, ADDI, SLL, SRA, SLLV, etc. The result of operations that use incorrect sign-extended 32-bit values is unpredictable.

### Cycle Timing for Multiply and Divide Instructions

MFHI and MFLO instructions are interlocked so that any attempt to read them before prior instructions complete, delays the execution of these instructions until the prior instructions finish. Table 2.9 gives the number of processor cycles (PCycles) required to resolve an interlock or stall between various multiply or divide instructions, and a subsequent MFHI or MFLO instruction.

| Instruction | Latency | Repeat Rate |
|---|---|---|
| MULT/MULTU | 5 | 4 |
| DIV/DIVU | 36 | 36 |
| DMULT/DMULTU | 9 | 8 |
| DDIV/DDIVU | 68 | 68 |

Table 2.9 Multiply/Divide Instruction Latency and Repeat Rates

### Jump and Branch Instructions

Jump and branch instructions change the control flow of a program. All jump and branch instructions occur with a delay of one instruction; that is, the instruction immediately following the jump or branch (the instruction in the *delay slot*) always executes while the target instruction is being fetched from storage.

Subroutine calls in high-level languages are usually implemented with Jump or Jump and Link instructions, both of which are J-type instructions. In J-type format, the 26-bit target address shifts left 2 bits and combines with the high-order 4 bits of the current program counter to form an absolute address. Returns, dispatches, and large cross-page jumps are usually implemented with the Jump Register or Jump and Link Register instructions. Both are R-type instructions that take the 32-bit or 64-bit byte address contained in one of the general purpose registers.

All branch-instruction target addresses are computed by adding the address of the instruction in the delay slot to the 16-bit *offset* (shifts left 2 bits and is sign-extended to 32 bits). All branches occur with a delay of one instruction. If a conditional branch is not taken, the instruction in the delay slot is nullified.

Table 2.10 lists the jump and branch instructions.

| OpCode | Description | MIPS ISA Level |
|---|---|---|
| BCzFL | Branch on Coprocessor z False Likely | II |
| BCzTL | Branch on Coprocessor z True Likely | II |
| BEQ | Branch on Equal | I |
| BEQL | Branch on Equal Likely | II |
| BGEZ | Branch on Greater Than or Equal to Zero | I |
| BGEZAL | Branch on Greater Than or Equal to Zero And Link | I |
| BGEZALL | Branch on Greater Than or Equal to Zero And Link Likely | II |
| BGEZL | Branch on Greater Than or Equal to Zero Likely | II |
| BGTZ | Branch on Greater Than Zero | I |
| BGTZL | Branch on Greater Than Zero Likely | II |
| BLEZ | Branch on Less Than or Equal to Zero | I |

Table 2.10 Jump and Branch Instructions (Part 1 of 2)

## Notes

| OpCode | Description | MIPS ISA Level |
|---|---|---|
| BLEZL | Branch on Less Than or Equal to Zero Likely | II |
| BLTZ | Branch on Less Than Zero | I |
| BLTZL | Branch on Less Than Zero Likely | II |
| BLTZAL | Branch on Less Than Zero And Link | I |
| BLTZALL | Branch on Less Than Zero And Link Likely | II |
| BNE | Branch on Not Equal | I |
| BNEL | Branch on Not Equal Likely | II |
| J | Jump | I |
| JAL | Jump And Link | I |
| JALR | Jump And Link Register | I |
| JR | Jump Register | I |

Table 2.10 Jump and Branch Instructions (Part 2 of 2)

### Special Instructions

Special instructions allow the software to initiate traps. They are always R-type. Table 2.11 lists the special instructions.

| OpCode | Description | MIPS ISA Level |
|---|---|---|
| SYSCALL | System Call | I |
| BREAK | Break | I |

Table 2.11 Special Instructions

### Coprocessor Instructions

Coprocessor instructions perform operations in their respective coprocessors. Coprocessor loads and stores are I-type, and coprocessor computational instructions have coprocessor-dependent formats. CP0 instructions perform operations specifically on the System Control Coprocessor registers to manipulate the memory management and exception handling facilities of the processor.

Table 2.12 and Table 2.13 list the coprocessor instructions. Table 2.14 lists the instructions used for exception processing.

| OpCode | Description | MIPS ISA Level |
|---|---|---|
| BCzT | Branch on Coprocessor z True | I |
| BCzF | Branch on Coprocessor z False | I |
| CFCz | Move Control From Coprocessor z | I |
| COPz | Coprocessor Operation z | I |
| CTCz | Move Control to Coprocessor z | I |
| DMFCz | Doubleword Move From Coprocessor z | II |
| DMTCz | Doubleword Move To Coprocessor z | II |
| LDCz | Load Double Coprocessor z | II |
| LWCz | Load Word to Coprocessor z | I |

Table 2.12 Coprocessor Instructions (Part 1 of 2)

## Notes

| OpCode | Description | MIPS ISA Level |
|---|---|---|
| MFCz | Move From Coprocessor z | I |
| MTCz | Move To Coprocessor z | I |
| SDCz | Store Double Coprocessor z | II |
| SWCz | Store Word from Coprocessor z | I |

Table 2.12 Coprocessor Instructions (Part 2 of 2)

| OpCode | Description | MIPS ISA Level |
|---|---|---|
| CACHE | Cache Operation | III |
| DCTR | Data Cache Tag Read | IV |
| DCTW | Data Cache Tag Write | IV |
| DMFC0 | Doubleword Move From CP0 | III |
| DMTC0 | Doubleword Move To CP0 | III |
| ERET | Exception Return | III |
| MFC0 | Move from CP0 | I |
| MTC0 | Move to CP0 | I |
| TLBP | Probe TLB for Matching Entry | I |
| TLBR | Read Indexed TLB Entry | I |
| TLBW | Write TLB Entry | IV |
| TLBWI | Write Indexed TLB Entry | I |
| TLBWR | Write Random TLB Entry | I |
| WAIT | Enter Standby Mode | III |

Table 2.13 CPO Instructions

| OpCode | Description | MIPS ISA Level |
|---|---|---|
| TEQ | Trap if Equal | II |
| TEQI | Trap if Equal Immediate | II |
| TGE | Trap if Greater Than or Equal | II |
| TGEI | Trap if Greater Than or Equal Immediate | II |
| TGEIU | Trap if Greater Than or Equal Immediate Unsigned | II |
| TGEU | Trap if Greater Than or Equal Unsigned | II |
| TLT | Trap if Less Than | II |
| TLTI | Trap if Less Than Immediate | II |
| TLTIU | Trap if Less Than Immediate Unsigned | II |
| TLTU | Trap if Less Than Unsigned | II |
| TNE | Trap if Not Equal | II |
| TNEI | Trap if Not Equal Immediate | II |

Table 2.14 Exception Instructions

Notes

## MIPS IV Instruction Set Additions to CPU Instructions

The following additions to MIPS III CPU instruction set are included in the MIPS IV CPU instruction set.

### Prefetch

- *PREF - Register + Offset Format*
- *PREFX - Register + Register Format*

*The R5000 does not implement prefetch actions; these instruction are executed as NO-OPS, fully compatible with the MIPS IV instruction set architecture.* (In their normal implementation, rather than as no-ops, the two prefetch instructions allow the compiler to issue instructions early so the corresponding data can be fetched and placed as close as possible to the CPU. Each instruction contains a 5-bit hint field which gives the coherency status of the line being prefetched. The line can be either shared, exclusive clean, or exclusive dirty. The contents of the general register specified by the base is added either to the 16 bit sign-extended offset or to the contents of the general register specified by the index to form a virtual address. This address together with the hint field is sent to the cache controller and a memory access is initiated. The region bits, 63:62, of the effective address must be supplied by the base. If the addition alters these bits an address exception occurs. The prefetch instruction never generates TLB-related exceptions. The PREF instruction is considered a standard processor instruction while the PREF instruction is considered a standard Coprocessor 1 instruction.)

### Integer Conditional Moves

- *MOVT - Move Conditional On Condition Code True*
- *MOVF - Move Conditional On Condition Code False*
- *MOVN - Move Conditional On Register Not Equal To Zero*
- *MOVZ - Move Conditional On Register Equal To Zero*

The four Integer Conditional Move instructions are used to test a condition code or a general register and then conditionally perform an integer move. The value of the floating-point condition code specified in the instruction by the 3-bit condition code specifier, or the value of the register indicated by the 5-bit general register specifier, is compared to zero. If the result indicates that the move should be performed, the contents of the specified source register is copied into the specified destination register.

Notes

# Floating-Point Unit (FPU)

Notes

## Introduction

The R5000 floating-point unit (FPU) operational functions consist of an adder, multiplier, and divider. The FPU, with associated system software, conforms fully to the ANSI/IEEE Standard 754–1985, *IEEE Standard for Binary Floating-Point Arithmetic*. In addition, the MIPS architecture fully supports the recommendations of the standard and precise exceptions.

The FPU operates as a coprocessor for the CPU (it is assigned coprocessor label *CP1*), and extends the CPU instruction set to perform operations on floating-point values. It has the following basic features:

- ***32-bit or 64-bit Operation****. The FR bit in the CPU Status register controls the selection of 32-bit 64-bit mode. Each register can hold single- or double-precision values.*
- ***Load and Store Instruction Set****. Like the CPU, the FPU uses a load- and store-oriented instruction set, with single-cycle load and store operations.*
- ***Tightly Coupled Coprocessor Interface****. The FPU resides on-chip to form a tightly coupled unit with a seamless integration of floating-point and fixed-point instruction sets. Since each unit receives and executes instructions in parallel, some floating-point instructions can execute at the same single-cycle-per-instruction rate as fixed-point instructions.*

Figure 3.1 illustrates the functional organization of the FPU.

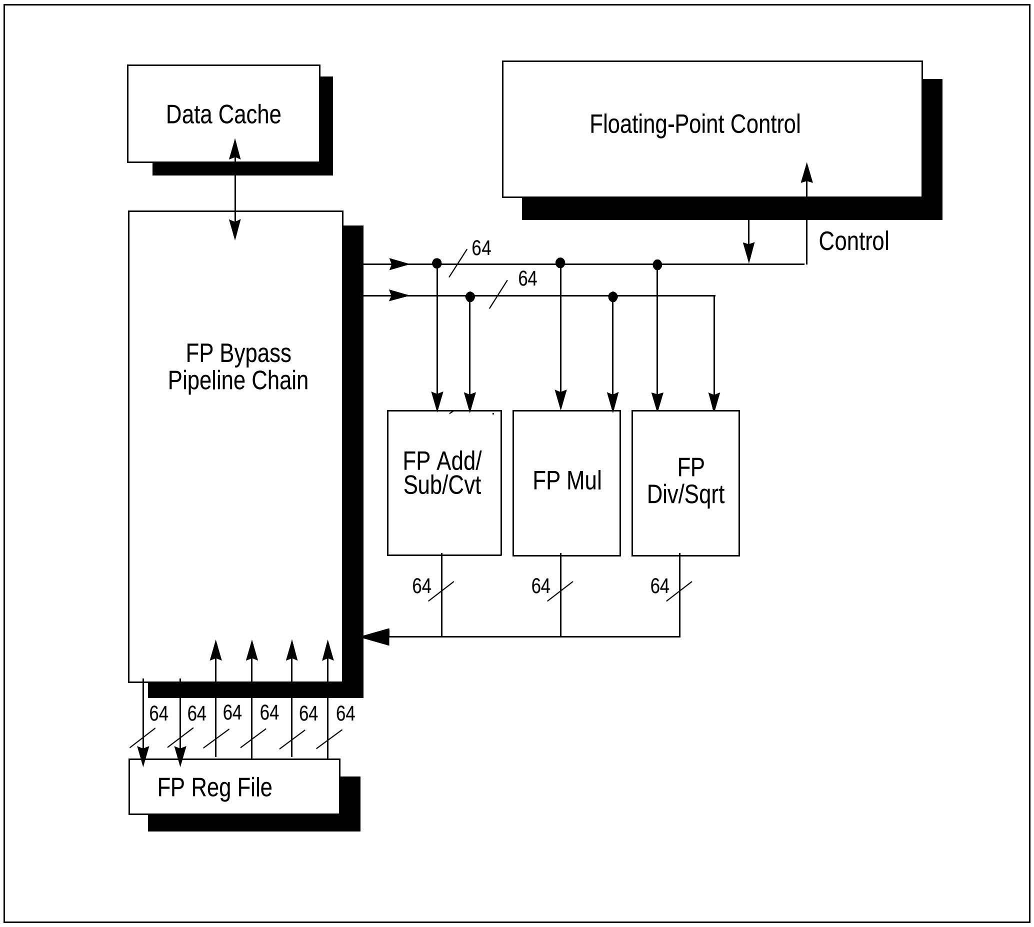


**Figure 3.1 FPU Functional Block Diagram**

Notes

## Floating-Point General Registers (FGRs)

The FPU has a set of *Floating-Point General Registers (FGR*s) that can be accessed in the following ways:

- *As 32 general purpose registers (32 FGRs), each of which is 32 bits wide, when the FR bit in the CPU Status register equals 0; or as 32 general purpose registers (32 FGRs), each of which is 64-bits wide, when FR equals 1. The CPU accesses these registers through move, load, and store instructions.*
- *As 16 floating-point registers (see the next section for a description of FPRs), each of which is 64 bits wide, when the FR bit in the CPU Status register equals 0. The FPRs hold values in either single- or double-precision floating-point format. Each FPR corresponds to adjacently numbered FGRs, as shown in Figure 3.2.*
- *As 32 floating-point registers (see the next section for a description of FPRs), each of which is 64 bits wide, when the FR bit in the CPU Status register equals 1. The FPRs hold values in either single- or double-precision floating-point format. Each FPR corresponds to an FGR as shown in Figure 3.2.*

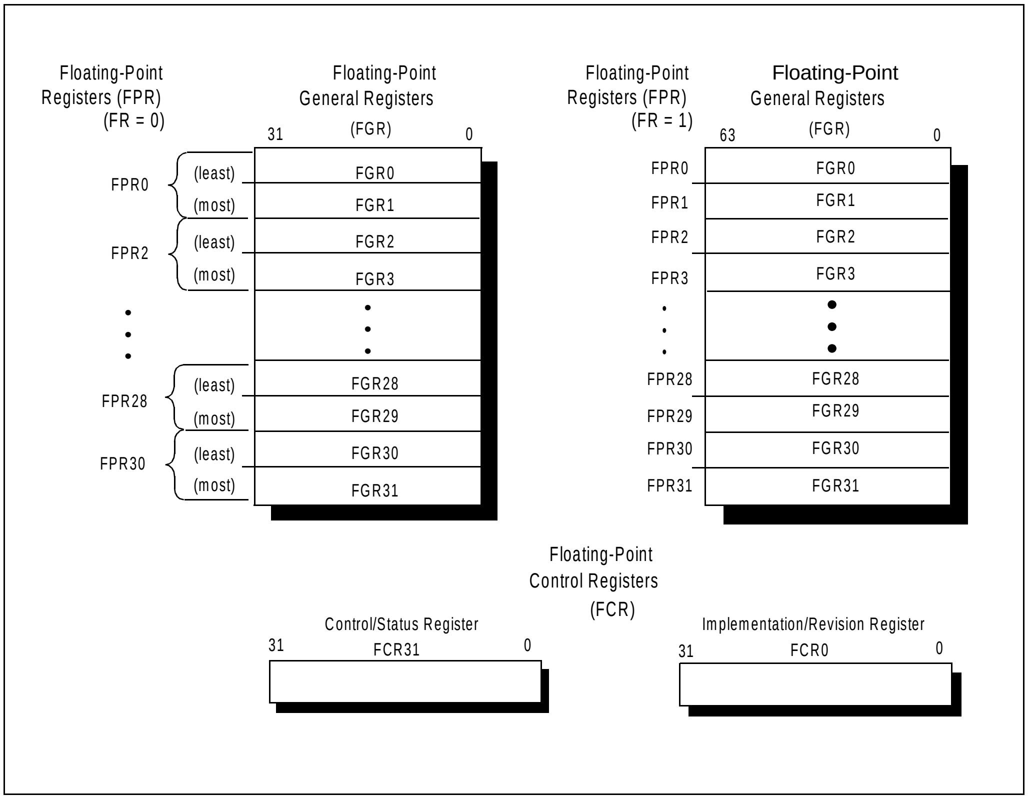

Floating-Point Registers (FPR) (FR = 0)
Floating-Point General Registers (FGR)
31
0
FPR0
(least)
(most)
FGR0
FGR1
FPR2
(least)
(most)
FGR2
FGR3
FPR28
(least)
(most)
FGR28
FGR29
FPR30
(least)
(most)
FGR30
FGR31
Floating-Point Registers (FPR) (FR = 1)
Floating-Point General Registers (FGR)
63
0
FPR0
FGR0
FPR1
FGR1
FPR2
FGR2
FPR3
FGR3
FPR28
FGR28
FPR29
FGR29
FPR30
FGR30
FPR31
FGR31
Floating-Point Control Registers (FCR)
Control/Status Register
31
FCR31
0
Implementation/Revision Register
31
FCR0
0

**Figure 3.2 FPU Registers**

## Floating-Point Registers (FPRs)

The FPRs are shown in Figure 3.2. These 64-bit registers hold floating-point values during floating-point operations and are physically formed from the *Floating-Point General Registers* (*FGRs*).

When the *FR* bit is set to a 1, the *FPR* references a single 64-bit *FGR* and all *FPR* register numbers are valid. If the *FR* bit equals 0, only even numbers (the *least* register, as shown in Figure 3.2) can be used to address *FPR*s. If the *FR* bit equals 0 during a double-precision floating-point operation, the general registers are accessed in double pairs. Thus, in a double-precision operation, selecting *Floating-Point Register 0* (*FPR0*) actually addresses adjacent *Floating-Point General Purpose* registers *FGR0* and *FGR1*.

Notes

# Floating-Point Control Registers (FCRs)

Table 3.1 lists the floating-point control registers (*FCR*s). These can only be accessed by Move operations. The *FCR*s include:

| FCR Number | Use |
|---|---|
| FCR0 | Implementation/Revision register: holds revision information about the FPU. |
| FCR1 to FCR30 | Reserved |
| FCR31 | Control/Status register: controls and monitors exceptions, holds the result of compare operations, and establishes rounding modes. |

**Table 3.1 Floating-Point Control Register Assignments**

## Implementation and Revision Register (FCR0)

The read-only *Implementation and Revision* register (*FCR0*) specifies the implementation and revision number of the FPU. This information can determine the coprocessor revision and performance level, and can also be used by diagnostic software. Figure 3.3 shows the layout of the register. Table 3.2 describes the register fields.

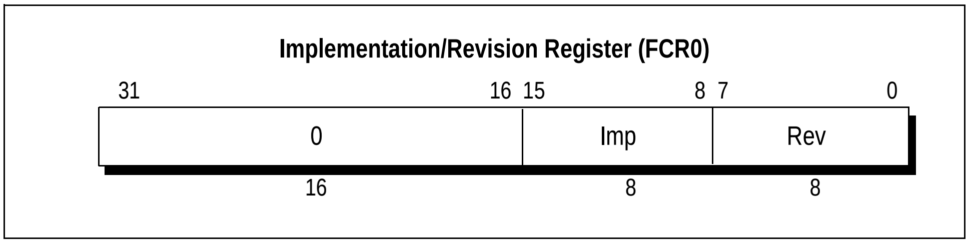


**Figure 3.3 Implementation/Revision Register**

| Field | Description |
|---|---|
| Imp | Implementation number (0x23) |
| Rev | Revision number in the form of *y.x* |
| 0 | Reserved. Must be written as zeroes, and returns zeroes when read. |

**Table 3.2 FCRO Fields**

The revision number is a value of the form *y.x*, where:

- *y is a major revision number held in bits 7:4.*
- *x is a minor revision number held in bits 3:0.*

The revision number distinguishes some chip revisions; however, MIPS does not guarantee that changes to its chips are necessarily reflected by the revision number, or that changes to the revision number necessarily reflect real chip changes. For this reason, revision number values are not listed, and software should not rely on the revision number to characterize the chip.

## Control/Status Register (FCR31)

The *Control/Status* register *(FCR31*) contains control and status information that can be accessed by instructions in either Kernel or User mode. *FCR31* also controls the arithmetic rounding mode and enables User-mode traps, as well as identifying any exceptions that may have occurred in the most recently executed instruction, along with any exceptions that may have occurred without being trapped.

Figure 3.4 shows the format of the *Control/Status* register, and Table 3.3 describes the register fields. Figure 3.5 shows the *Control/Status* register *Cause, Flag,* and *Enable* fields*.*

Notes

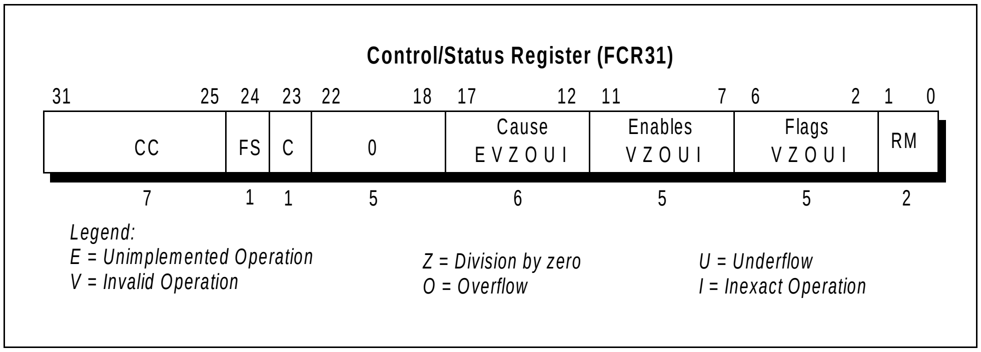


**Figure 3.4 FP Control/Status Register Bit Assignments**

| Field | Description |
|---|---|
| CC | Condition code. |
| FS | When set, denormalized results are flushed to 0 instead of causing an unimplemented operation exception. |
| C | Condition bit. See description of *Control/Status* register *Condition* bit. |
| Cause | Cause bits. See description of *Control/Status* register *Cause, Flag,* and *Enable* bits. |
| Enables | Enable bits. See description of *Control/Status* register *Cause, Flag,* and *Enable* bits. |
| Flags | Flag bits. See description of *Control/Status* register *Cause, Flag,* and *Enable* bits. |
| RM | Rounding mode bits. See description of *Control/Status* register *Rounding Mode Control* bits. |

**Table 3.3 Control/Status Register Fields**

### Accessing the Control/Status Register

When the *Control/Status* register is read by a Move Control From Coprocessor 1 (CFC1) instruction, all unfinished instructions in the pipeline are completed before the contents of the register are moved to the main processor. If a floating-point exception occurs as the pipeline empties, the FP exception is taken and the CFC1 instruction is re-executed after the exception is serviced.

The bits in the *Control/Status* register can be set or cleared by writing to the register using a Move Control To Coprocessor 1 (CTC1) instruction. *FCR31* must only be written to when the FPU is not actively executing floating-point operations; this can be ensured by reading the contents of the register to empty the pipeline.

### IEEE Standard 754

IEEE Standard 754 specifies that floating-point operations detect certain exceptional cases, raise flags, and invoke an exception handler when an exception occurs. These features are implemented in the MIPS architecture with the *Cause*, *Enable*, and *Flag* fields of the *Control/Status* register. The *Flag* bits implement IEEE-754 exception-status flags, and the *Cause* and *Enable* bits implement exception handling.

### Control/Status Register FS Bit

When the *FS* bit is set, denormalized results are flushed to 0 instead of causing an unimplemented operation exception.

### Control/Status Register Condition Bit

When a floating-point Compare operation takes place, the result is stored at bit 23, the *Condition* bit, to save or restore the state of the condition line. The *C* bit is set to 1 if the condition is true; the bit is cleared to 0 if the condition is false. Bit 23 is affected only by Compare and Move Control To FPU instructions.

Notes

## Control/Status Register Cause, Flag, and Enable Fields

Figure 3.5 illustrates the *Cause*, *Flag*, and *Enable* fields of the *Control/Status* register.

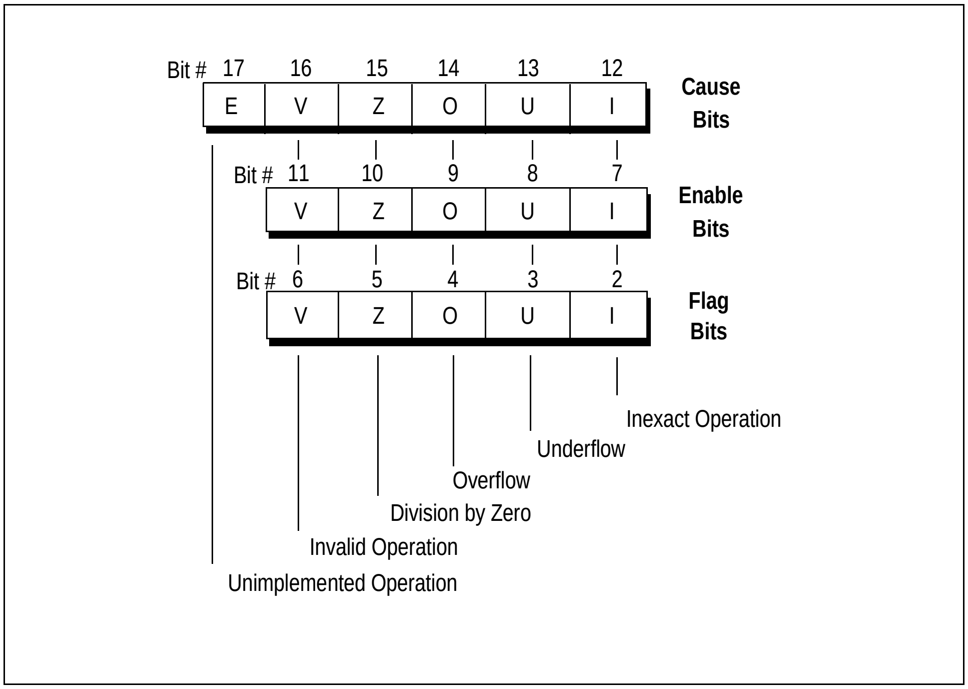


**Figure 3.5 Control/Status Register Cause, Flag, and Enable Fields**

### Cause Bits

Bits 17:12 in the *Control/Status* register contain *Cause* bits, as shown in Figure 3.5. The *Cause* bits reflect the results of the most recently executed instruction. The bits are a logical extension of the CP0 *Cause* register; they identify the exceptions raised by the last floating-point operation and raise an interrupt or exception if the corresponding enable bit is set. If more than one exception occurs on a single instruction, each appropriate bit is set.

The *Cause* bits are written by each floating-point operation (but not by Load, Store, or Move operations). The Unimplemented Operation (*E*) bit is set to 1 if software emulation is required, otherwise it remains 0. The other bits are set to 1 or cleared to 0 to indicate the occurrence or non-occurrence (respectively) of an IEEE-754 exception.

When a floating-point exception is taken, no results are stored, and the only state affected is the *Cause* bit.

### Enable Bits

A floating-point exception is generated any time a *Cause* bit and the corresponding *Enable* bit are set. A floating-point operation that sets an enabled *Cause* bit forces an immediate exception, as does setting both *Cause* and *Enable* bits with CTC1.

There is no enable for Unimplemented Operation (*E*). Setting Unimplemented Operation always generates a floating-point exception.

Before returning from a floating-point exception, software must first clear the enabled *Cause* bits with a CTC1 instruction to prevent a repeat of the interrupt. Thus, User-mode programs can never observe enabled *Cause* bits set; if this information is required in a User-mode handler, it must be passed somewhere other than the *Status* register.

For a floating-point operation that sets only unenabled *Cause* bits, no exception occurs and the default result defined by IEEE-754 is stored. In this case, the exceptions that were caused by the immediately previous floating-point operation can be determined by reading the *Cause* field.

Notes

### Flag Bits

When an exception case is detected and the exception Enable is not set, the corresponding flag bit is set. If an exception is taken, none of the flag bits are modified. Note, however, that system software may set the flag bits before invoking a user exception handler.

The *Flag* bits are cumulative and indicate that an exception was raised by an operation that was executed since they were explicitly reset. *Flag* bits are set to 1 if an IEEE-754 exception is raised, otherwise they remain unchanged. The *Flag* bits are never cleared as a side effect of floating-point operations; however, they can be set or cleared by writing a new value into the *Status* register, using a Move-To-Coprocessor Control instruction.

### Control/Status Register Rounding Mode Control Bits

Bits 1 and 0 in the *Control/Status* register constitute the *Rounding Mode* (*RM*) field.

As shown in Table 3.4, these bits specify the rounding mode that the FPU uses for all floating-point operations.

| Rounding Mode RM(1:0) | Mnemonic | Description |
|---|---|---|
| 0 | RN | Round result to nearest representable value; round to value with least-significant bit 0 when the two nearest representable values are equally near. |
| 1 | RZ | Round toward 0: round to value closest to and not greater in magnitude than the infinitely precise result. |
| 2 | RP | Round toward +∞: round to value closest to and not less than the infinitely precise result. |
| 3 | RM | Round toward – ∞: round to value closest to and not greater than the infinitely precise result. |

**Table 3.4 Rounding Mode Bit Decoding**

## FPU Data Formats

The FPU supports both floating-point and fixed-point data formats. The floating-point formats can be single-precision binary or double-precision binary. The fixed-point formats can be 32- or 64-bit binary.

### Floating-Point Formats

The FPU performs both 32-bit (single-precision) and 64-bit (double-precision) IEEE standard floating-point operations. The 32-bit single-precision format has a 24-bit signed-magnitude fraction field (*f*+*s*) and an 8-bit exponent (*e*), as shown in Figure 3.6.

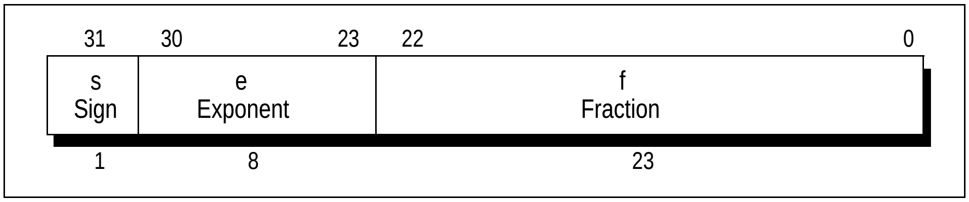


**Figure 3.6 Single-Precision Floating-Point Format**

The 64-bit double-precision format has a 53-bit signed-magnitude fraction field (*f*+*s)* and an 11-bit exponent, as shown in Figure 3.7.

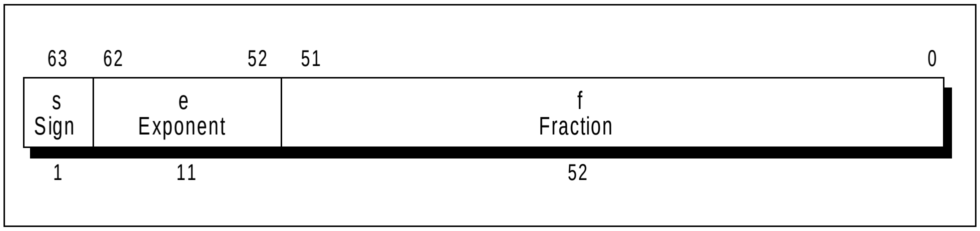


**Figure 3.7 Double-Precision Floating-Point Format**

As shown in the above figures, numbers in floating-point format are composed of three fields:

- *sign field, s*
- *biased exponent, e = E + bias*
- *fraction, f = .$b_1b_2$....$b_{p-1}$*

The range of the unbiased exponent *E* includes every integer between the two values $E_{min}$ and $E_{max}$ inclusive, together with two other reserved values:

- *$E_{min}$ -1 (to encode ±0 and denormalized numbers)*
- *$E_{max}$ +1 (to encode ±• and NaNs [Not a Number])*

For single- and double-precision formats, each representable non-zero numerical value has just one encoding.

For single- and double-precision formats, the value of a number, *v*, is determined by the equations shown in Table 3.5.

| No. | Equation |
|---|---|
| (1) | if E = $E_{max}$+1 and *f* ¼ 0, then *v* is NaN, regardless of *s* |
| (2) | if E = $E_{max}$+1 and f = 0, then *v* = $(–1)^s$ • |
| (3) | if $E_{min}$ £ E £ $E_{max}$, then *v* = $(–1)^s2^E$(1.*f*) |
| (4) | if E = $E_{min}$–1 and f ¼ 0, then *v* = $(–1)^s2^{Emin}$(0.*f*) |
| (5) | if E = $E_{min}$–1 and f = 0, then *v* = $(–1)^s$0 |

**Table 3.5 Calculating Values in Single and Double-Precision Formats**

For all floating-point formats, if *v* is NaN, the most-significant bit of *f* determines whether the value is a signaling or quiet NaN: *v* is a signaling NaN if the most-significant bit of *f* is set, otherwise, *v* is a quiet NaN.

Table 3.6 defines the values for the format parameters; minimum and maximum floating-point values are given in Table 3.7.

| Parameter | Format | |
|---|---|---|
| | Single | Double |
| $E_{max}$ | +127 | +1023 |
| $E_{min}$ | –126 | –1022 |
| Exponent *bias* | +127 | +1023 |
| Exponent width in bits | 8 | 11 |
| Integer bit | hidden | hidden |
| f (Fraction width in bits) | 24 | 53 |
| Format width in bits | 32 | 64 |

**Table 3.6 Floating-Point Format Parameter Values**

Notes

| Type | Value |
|---|---|
| Float Minimum | 1.40129846e–45 |
| Float Minimum Norm | 1.17549435e–38 |
| Float Maximum | 3.40282347e+38 |
| Double Minimum | 4.9406564584124654e–324 |
| Double Minimum Norm | 2.2250738585072014e–308 |
| Double Maximum | 1.7976931348623157e+308 |

Table 3.7 Minimum and Maximum Floating-Point Values

### Binary Fixed-Point Format

Binary fixed-point values are held in two's complement format. Unsigned fixed-point values are not directly provided by the floating-point instruction set. Figure 3.8 illustrates binary fixed-point format; Table 3.7 lists the binary fixed-point format fields.

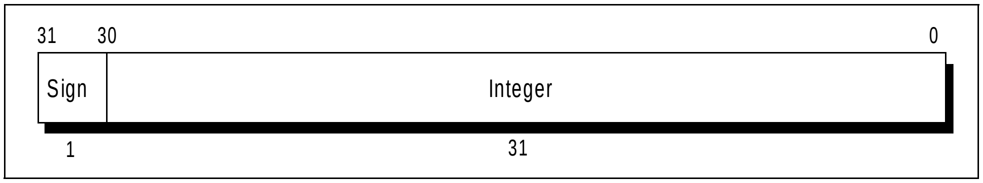


Figure 3.8 Binary Fixed-Point Format

| Field | Description |
|---|---|
| sign | sign bit |
| integer | integer value |

Table 3.8 Binary Fixed-Point Format Fields

## Floating-Point Instruction Set Summary

All FPU instructions are 32 bits long, aligned on a word boundary. They can be divided into the following groups:

- ***Load, Store, and Move*** *instructions move data between memory, the main processor, and the FPU General Purpose registers.*
- ***Conversion*** *instructions perform conversion operations between the various data formats.*
- ***Computational*** *instructions perform arithmetic operations on floating-point values in the FPU registers.*
- ***Compare*** *instructions perform comparisons of the contents of registers and set a conditional bit based on the results.*
- ***Branch on FPU Condition*** *instructions perform a branch to the specified target if the specified coprocessor condition is met.*

In the instruction formats shown in Table 3.9 through Table 3.12, the *fmt* appended to the instruction opcode specifies the data format: *S* specifies single-precision binary floating-point, *D* specifies double-precision binary floating-point, *W* specifies 32-bit binary fixed-point, and *L* specifies 64-bit (long) binary fixed-point.

## Notes

| OpCode | Description |
|---|---|
| LWC1 | Load Word to FPU |
| LWXC1 | Load Word Indexed to FPU |
| SWC1 | Store Word from FPU |
| SWXC1 | Store Word Indexed from FPU |
| LDC1 | Load Doubleword to FPU |
| LDXC1 | Load Doubleword Indexed to FPU |
| SDC1 | Store Doubleword From FPU |
| SDXC1 | Store Doubleword Indexed From FPU |
| MTC1 | Move Word To FPU |
| MFC1 | Move Word From FPU |
| CTC1 | Move Control Word To FPU |
| CFC1 | Move Control Word From FPU |
| DMTC1 | Doubleword Move To FPU |
| DMFC1 | Doubleword Move From FPU |
| PREF | Prefetch - Register + Offset |
| PREFX | Prefetch Indexed - Register + Register |

Table 3.9 FPU Instruction Summary: Load, Move and Store Instructions

| OpCode | Description |
|---|---|
| CVT.S.fmt | Floating-Point Convert to Single FP |
| CVT.D.fmt | Floating-Point Convert to Double FP |
| CVT.W.fmt | Floating-Point Convert to 32-bit Fixed Point |
| CVT.L.fmt | Floating-Point Convert to 64-bit Fixed Point |
| ROUND.W.fmt | Floating-Point Round to 32-bit Fixed Point |
| ROUND.L.fmt | Floating-Point Round to 64-bit Fixed Point |
| TRUNC.W.fmt | Floating-Point Truncate to 32-bit Fixed Point |
| TRUNC.L.fmt | Floating-Point Truncate to 64-bit Fixed Point |
| CEIL.W.fmt | Floating-Point Ceiling to 32-bit Fixed Point |
| CEIL.L.fmt | Floating-Point Ceiling to 64-bit Fixed Point |
| FLOOR.W.fmt | Floating-Point Floor to 32-bit Fixed Point |
| FLOOR.L.fmt | Floating-Point Floor to 64-bit Fixed Point |

Table 3.10 FPU Instruction Summary: Conversion Instructions

Notes

| OpCode | Description |
|---|---|
| ADD.fmt | Floating-Point Add |
| SUB.fmt | Floating-Point Subtract |
| MUL.fmt | Floating-Point Multiply |
| DIV.fmt | Floating-Point Divide |
| ABS.fmt | Floating-Point Absolute Value |
| MOV.fmt | Floating-Point Move |
| NEG.fmt | Floating-Point Negate |
| SQRT.fmt | Floating-Point Square Root |
| RECIP | Floating-Point Reciprocal |
| RSQRT | Floating-Point Reciprocal Square Root |

**Table 3.11 FPU Instruction Summary: Computational Instructions**

| OpCode | Description |
|---|---|
| C.cond.fmt | Floating-Point Compare |
| BC1T | Branch on FPU True |
| BC1F | Branch on FPU False |
| BC1TL | Branch on FPU True Likely |
| BC1FL | Branch on FPU False Likely |

**Table 3.12 FPU Instruction Summary: Compare and Branch Instructions**

### Floating-Point Load, Store, and Move Instructions

This section discusses the manner in which the FPU uses the load, store and move instructions listed in Table 3.9.

### Transfers Between FPU and Memory

All data movement between the FPU and memory is accomplished by using one of the following instructions:

- *Load Word To Coprocessor 1 (LWC1) or Store Word From Coprocessor 1 (SWC1) instructions, which reference a single 32-bit word of the FPU general registers.*
- *Load Doubleword (LDC1) or Store Doubleword (SDC1) instructions, which reference a 64-bit doubleword.*

These load and store operations are unformatted; no format conversions are performed and therefore no floating-point exceptions can occur due to these operations.

### Transfers Between FPU and CPU

Data can also be moved directly between the FPU and the CPU by using one of the following instructions:

- *Move To Coprocessor 1 (MTC1).*
- *Move From Coprocessor 1 (MFC1).*
- *Doubleword Move To Coprocessor 1 (DMTC1).*
- *Doubleword Move From Coprocessor 1 (DMFC1).*

Like the floating-point load and store operations, these operations perform no format conversions and never cause floating-point exceptions.

### Load Delay and Hardware Interlocks

The instruction immediately following a load can use the contents of the loaded register. In such cases the hardware interlocks, requiring additional real cycles; for this reason, scheduling load-delay slots is desirable, although it is not required.

### Data Alignment

All coprocessor loads and stores reference the following aligned data items:

- *For word loads and stores, the access type is always WORD, and the low-order 2 bits of the address must always be 0.*
- *For doubleword loads and stores, the access type is always DOUBLEWORD, and the low-order 3 bits of the address must always be 0.*

### Endianness

Regardless of byte-numbering order (endianness) of the data, the address specifies the byte that has the smallest byte address in the addressed field. For a big-endian system, it is the leftmost byte; for a little-endian system, it is the rightmost byte.

### Floating-Point Conversion Instructions

Conversion instructions perform conversions between the various data formats such as single- or double-precision, fixed- or floating-point formats.

### Floating-Point Computational Instructions

Computational instructions perform arithmetic operations on floating-point values, in registers. There are two categories of computational instructions:

- *3-operand register-type instructions, which perform floating-point addition, subtraction, multiplication, and division.*
- *2-operand register-type instructions, which perform floating-point absolute value, move, negate, and square-root operations.*

For a detailed description of each instruction, refer to the IDT MIPS Microprocessor Family Software Manual.

### Branch on FPU Condition Instructions

The Branch on FPU (coprocessor unit 1) condition instructions that can test the result of the FPU compare (C.cond) instructions. For a detailed description of each instruction, refer to the IDT MIPS Microprocessor Family Software Manual.

### Floating-Point Compare Operations

The floating-point compare (C.fmt.cond) instructions interpret the contents of two FPU registers (*fs, ft*) in the specified format (*fmt*) and arithmetically compare them. A result is determined based on the comparison and conditions (*cond*) specified in the instruction.

Table 3.13 lists the mnemonics for the compare-instruction conditions.

Notes

| Mnemonic | Definition | Mnemonic | Definition |
|---|---|---|---|
| T | True | F | False |
| OR | Ordered | UN | Unordered |
| NEQ | Not Equal | EQ | Equal |
| OLG | Ordered or Less Than or Greater Than | UEQ | Unordered or Equal |
| UGE | Unordered or Greater Than or Equal | OLT | Ordered Less Than |
| OGE | Ordered Greater Than | ULT | Unordered or Less Than |
| UGT | Unordered or Greater Than | OLE | Ordered Less Than or Equal |
| OGT | Ordered Greater Than | ULE | Unordered or Less Than or Equal |
| ST | Signaling True | SF | Signaling False |
| GLE | Greater Than, or Less Than or Equal | NGLE | Not Greater Than or Less Than or Equal |
| SNE | Signaling Not Equal | SEQ | Signaling Equal |
| GL | Greater Than or Less Than | NGL | Not Greater Than or Less Than |
| NLT | Not Less Than | LT | Less Than |
| GE | Greater Than or Equal | NGE | Not Greater Than or Equal |
| NLE | Not Less Than or Equal | LE | Less Than or Equal |
| GT | Greater Than | NGT | Not Greater Than |

Table 3.13 Mnemonics of Compare-Instruction Conditions

## MIPS IV Instruction Set Additions to FPU Instructions

The following additions to MIPS III FPU instruction set are included in the MIPS IV FPU instruction set.

### Indexed Floating-Point Load

- *LWXC1 - Load word indexed to Coprocessor 1.*
- *LDXC1 - Load doubleword indexed to Coprocessor 1.*

The two Indexed Floating-Point Load instructions transfer floating-point data types from memory to the floating-point registers using register + register addressing mode. The contents of the general register specified by the base is added to the contents of the general register specified by the index to form a virtual address. The contents of the word or doubleword specified by the effective address are loaded into the floating-point register specified in the instruction. There are no indexed loads to general registers.

The region bits (63:62) of the effective address must be supplied by the base. If the addition alters these bits an address exception occurs. Also, if the address is not aligned, an address exception occurs.

### Indexed Floating-Point Store

- *SWXC1 - Store word indexed to Coprocessor 1.*
- *SDXC1 - Store doubleword indexed to Coprocessor 1.*

The two Indexed Floating-Point Store instructions transfer floating-point data types from the floating-point registers to memory using register + register addressing mode. The contents of the general register specified by the base is added to the contents of the general register specified by the index to form a virtual address. The contents of the floating-point register specified in the instruction is stored to the memory location specified by the effective address.

The region bits (63:62) of the effective address must be supplied by the base. If the addition alters these bits an address exception occurs. Also, if the address is not aligned, an address exception occurs.

Notes

### Branch on Floating-Point Coprocessor

- *BC1T - Branch on FP Condition True*
- *BC1F - Branch on FP Condition False*
- *BC1TL - Branch on FP Condition True Likely*
- *BC1FL - Branch on FP Condition False Likely*

The four Branch on Floating-Point Coprocessor instructions are extensions of the branch instructions in various prior MIPS instruction sets, with which they are upward-compatible. The BC1T and BC1F instructions are extensions of MIPS I. BC1TL and BC1FL are extensions of MIPS III. These instructions test one of eight floating-point condition codes. If no condition code is specified, condition code bit zero is selected. This encoding is downward-compatible with previous MIPS architectures.

The branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended to 64 bits. If the contents of the floating-point condition code specified in the instruction are equal to the test value, the target address is branched to with a delay of one instruction. If the conditional branch is not taken and the nullify delay bit in the instruction is set, the instruction in the branch delay slot is nullified.

### Floating-Point Multiply-Add/Subtract

- *MADD - Floating-Point Multiply-Add*
- *MSUB - Floating-Point Multiply-Subtract*
- *NMADD - Floating-Point Negative Multiply-Add*
- *NMSUB - Floating-Point Negative Multiply-Subtract*

The four Floating-Point Multiply-Add/Subtract instructions compute two floating-point operations with one instruction. Each of the instructions performs intermediate rounding.

### Floating-Point Compare

- *C.cond - Compare*
- *C.cond - Implies cc=0*

The two Floating-Point Compare instructions are upward-compatible extensions of the floating-point compare instructions of the MIPS I instruction set and produce a boolean result which is stored in one of the condition codes.

The contents of the two FP source registers specified in the instruction are interpreted and arithmetically compared. A result is determined based on the comparison and the conditions specified in the instruction. If one of the values is not a number and the high order bit of the condition field is set, an invalid operations trap occurs. Comparisons are exact and neither overflow or underflow.

The implication for compiler code scheduling is that a compare instruction may be immediately followed by a dependent floating-point conditional move instruction, but may not be immediately followed by a dependent branch on floating-point coprocessor condition instruction or a dependent integer conditional move instruction. This restriction applies only to the condition code specified in the 3-bit condition code specifier of the instruction. All other condition codes are unaffected.

### Floating-Point Conditional Moves

- *MOVT.fmt - Floating-Point Conditional Move on condition code true*
- *MOVF.fmt - Floating-Point Conditional Move on condition code false*
- *MOVN.fmt - Floating-Point Conditional Move on register not equal to zero*
- *MOVZ.fmt - Floating-Point Conditional Move on register equal to zero*

The four Floating-Point Conditional Move instructions are used to test a condition code or a general register and then conditionally perform a floating-point move. The value of the floating-point condition code specified by the 3-bit condition code specifier, or the value of the register indicated by the 5-bit general register specifier, is compared to zero. If the result indicates that the move should be performed, the contents of the specified source register is copied into the specified destination register. All of these conditional floating-point move operations are non-arithmetic. Consequently, no IEEE-754 exceptions occur as a result of these instructions.

### Reciprocal's

- *RECIP.fmt - Reciprocal Approximation*
- *RSQRT.fmt - Reciprocal Square Root Approximation*

The Reciprocal Approximation instruction performs a reciprocal approximation on a floating-point value. The reciprocal of the value in the floating-point source register is approximated and placed in a destination register. The numerical accuracy of this operation is implementation-dependent, based on the rounding mode used.

The Reciprocal Square Root Approximation instruction performs a reciprocal square root approximation on a floating-point value. The reciprocal of the positive square root of a value in the floating-point source register is approximated and placed in a destination register. The numerical accuracy of this operation is implementation-dependent, based on the rounding mode used.

The approximation is due to the fact that neither of these instructions meets IEEE accuracy requirements. In both cases a small amount of precision has been sacrificed, thereby significantly reducing execution time. For example, in the case of a RECIP instruction, X/Y is computed by taking the reciprocal of Y and multiplying that result by X. The reduced execution time of the reciprocal operation allows a RECIP followed by a MUL (multiply) instruction to be executed faster than a single DIV (divide) instruction. The performance difference between a RSQRT instruction and a SQRT followed by a DIV instruction is implementation-dependent.

On the R5000, the RECIP instruction has the same latency as a DIV instruction, but a RSQRT is faster than a SQRT followed by a RECIP.

### FPU-Instruction Latencies

Table 3.14 shows the execution-stage latencies and repeat throughput for FPU instructions. The values assume the result of the operation is immediately used in a succeeding operation.

| Instruction Group | Latency | Repeat |
|---|---|---|
| Absolute | 1 | 1 |
| Add | 4 | 1 |
| BC1T | 1 | 1 |
| BC1F | 1 | 1 |
| BC1TL | 1 | 1 |
| BC1FL | 1 | 1 |
| CEIL.w | 4 | 1 |
| CEIL.l | 4 | 1 |
| CFC1 | 2 | 1 |
| Compare | 1 | 1 |
| CTC1 | 3 | 3 |
| CVT.s.d | 4 | 1 |
| CVT.s.w | 6 | 3 |

[1] Trap on greater than 53 bits of significance

[2] Trap on greater than 52 bits of significance.

**Table 3.14 Floating-Point Instruction Latencies**

## Notes

| Instruction Group | Latency | Repeat |
|---|---|---|
| CVT.s.l[2] | 6 | 3 |
| CVT.d.s | 4 | 1 |
| CVT.d.w | 4 | 1 |
| CVT.d.l[2] | 4 | 1 |
| CVT.w.s | 4 | 1 |
| CVT.w.d | 4 | 1 |
| CVT.l.s | 4 | 1 |
| CVT.l.d | 4 | 1 |
| DIV.s | 21 | 19 |
| DIV.d | 36 | 34 |
| DMFC1 | 2 | 1 |
| DMTC1 | 2 | 1 |
| FLOOR.w | 4 | 1 |
| FLOOR.l | 4 | 1 |
| LDC1 | 2 | 2 |
| Load | 2 | 1 |
| Load Indexed | 3 | 2 |
| LWC1 | 1 | 1 |
| MADD.s | 4 | 1 |
| MADD.d | 5 | 2 |
| MFC1 | 2 | 1 |
| Move | 1 | 1 |
| Move Conditional | 1 | 1 |
| MSUB.s | 4 | 1 |
| MSUB.d | 5 | 2 |
| MTC1 | 2 | 1 |
| MUL.s | 4 | 1 |
| MUL.d | 5 | 2 |
| Negative | 1 | 1 |
| NMADD.s | 4 | 1 |
| NMADD.d | 5 | 2 |
| NMSUB.s | 4 | 1 |
| NMSUB.d | 5 | 2 |
| Prefetch | N/A | 1 |
| Prefetch Indexed | N/A | 2 |

[1] Trap on greater than 53 bits of significance

[2] Trap on greater than 52 bits of significance.

Table 3.14 Floating-Point Instruction Latencies

**Notes**

| Instruction Group | Latency | Repeat |
|---|---|---|
| RECIP.s | 21 | 19 |
| RECIP.d | 36 | 34 |
| ROUND.w | 4 | 1 |
| ROUND.l[1] | 4 | 1 |
| RSQRT.s | 38 | 36 |
| RSQRT.d | 68 | 66 |
| SDC1 | 1 | 1 |
| SQRT.s | 21 | 19 |
| SQRT.d | 36 | 34 |
| Store | N/A | 1 |
| Store Indexed | N/A | 2 |
| Subtract | 4 | 1 |
| SWC1 | 1 | 1 |
| TRUNC.w | 4 | 1 |
| TRUNC.l | 4 | 1 |

[1] Trap on greater than 53 bits of significance

[2] Trap on greater than 52 bits of significance.

**Table 3.14 Floating-Point Instruction Latencies**

# Instruction Pipeline

Notes

## Introduction

The R5000 processor has a dual-issue, five-stage instruction pipeline with two parallel paths, one for integer (CPU) instructions and the other floating-point (FPU) instructions. Each stage in the CPU-instruction path takes one PCycle (one cycle of the processor clock, which runs at a multiple of the frequency of the system clock, SysClock). Thus, the execution of each CPU instruction takes at least five PCycles. A CPU instruction can take longer—for example, if the required data is not in the cache, the data must be retrieved from main memory. In the FPU-instruction path, most FPU instructions require more than one PCycle in the execution stage.

Once the pipeline has been filled, five instructions can be executed simultaneously. Figure 4.1 shows the five stages of the instruction pipeline.

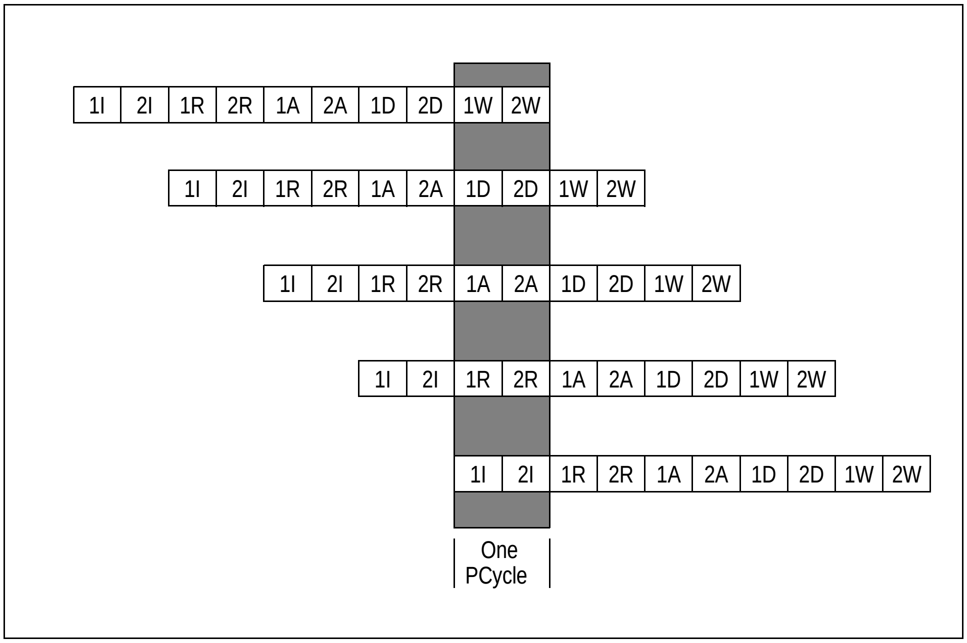


Figure 4.1 Instruction Pipeline Stages

## Instruction Pipeline Stages

1I - Instruction Fetch, Phase One

2I - Instruction Fetch, Phase Two

1R - Register Read, Phase One

2R - Register Read, Phase Two

1A - Execution, Phase One

2A - Execution, Phase Two

1D - Data Load/Store, Phase One

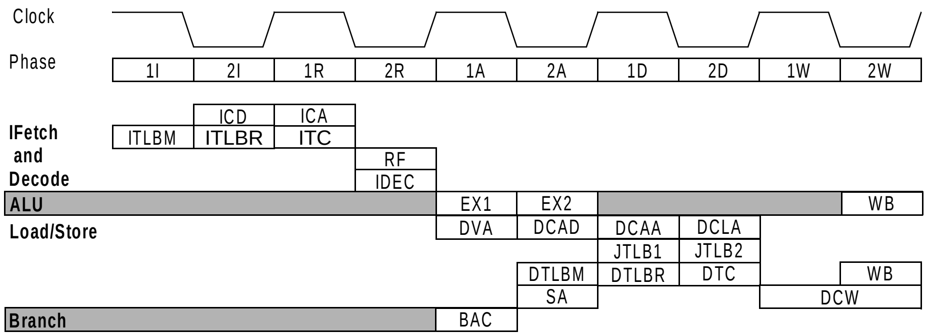
2D - Data Load/Store, Phase Two

1W - Write Back, Phase One

2W - Write Back, Phase Two

Notes

Figure 4.2 shows the CPU pipeline activities occurring during each ALU pipeline stage, for load, store, and branch instructions.



| | | | |
|---|---|---|---|
| ICD | Instruction-cache address decode | ICA | Instruction-cache array access |
| ITLBM | Instruction address translation match | ITLBR | Instruction address translation read |
| ITC | Instruction tag check | RF | Register operand read |
| IDEC | Instruction address translation phase 2 | EX1 | Execute operation - phase 1 |
| EX2 | Execute operation - phase two | WB | Write back to register file |
| DVA | Data virtual-address calculation | DCAD | Data-cache address decode |
| DCAA | Data-cache array access | DCLA | Data-cache load align |
| JTLB1 | JTLB address translation - phase 1 | JTLB2 | JTLB address translation - phase 2 |
| DTLBM | Data address translation match | DTLBR | Data address translation read |
| DTC | Data tag check | SA | Store align |
| DCW | Data-cache write | BAC | Branch address calculation |

**Figure 4.2 Integer (CPU) Pipeline Activities**

## Dual Issue

The R5000 dual-issue mechanism allows two instructions to be dispatched per processor cycle (PCycle) under the following condition: a floating-point ALU operation can be dispatched along with any other type of instruction, as long as the other instruction is not another floating-point ALU operation. In this context, “any other type of instruction” includes all integer instructions as well as floating-point loads and stores.

Figure 4.3 shows a simplified diagram of the dual issue mechanism.

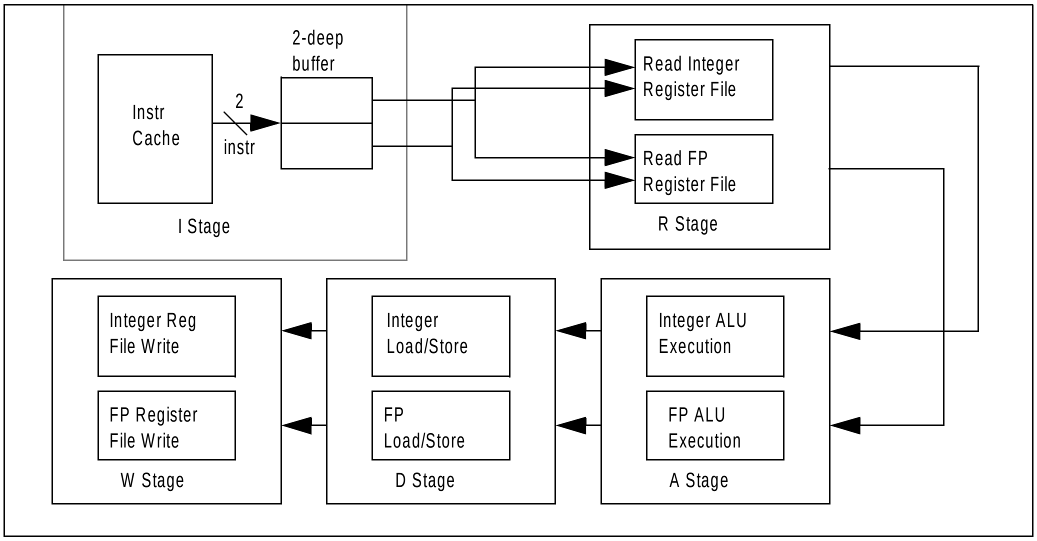


**Figure 4.3 Dual-Issue Mechanism, Showing CPU and FPU Pipelines**

Notes

The events that occur in each stage are:

### I - Stage

Two instructions are fetched from the instruction cache and placed in a 2-deep instruction buffer. Issue logic determines the type of instruction and which pipeline the instruction is routed to. Also, the instruction cache tag is checked against the page frame number (PFN) obtained from the ITLB.

### R - Stage

Any required operands are fetched from the appropriate register file, and the decision is made to either proceed or slip the instruction based on any interlock conditions. For branch instruction, the branch address is calculated.

### A - Stage

The appropriate ALU begins the arithmetic, logical, or shift operation. The data virtual address is calculated for any load or store instructions. The appropriate ALU determines whether the branch condition is true. The data cache access is started.

### D - Stage

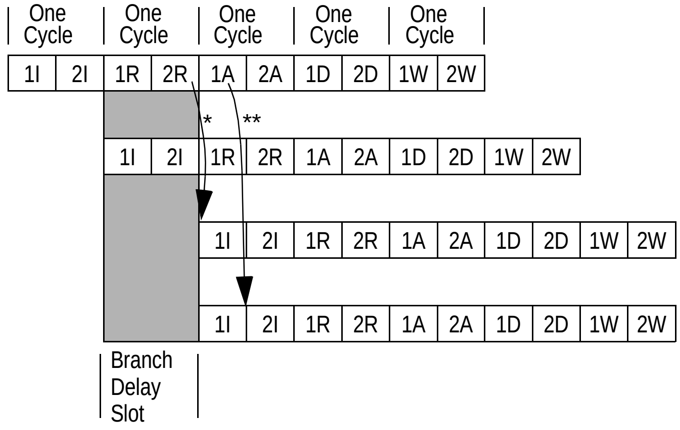
The data cache access is completed. Data is shifted down and extended. Data address translation in the DTLB completes. The virtual to physical address translation in the JTLB is performed. The data cache tag is checked against the PFN from the DTLB or JTLB for any data cache access.

### W - Stage

The processor resolves all exceptions. For register-to-register and load instructions, the result is written back to the appropriate register file.

## Branch Delay

The CPU pipeline has a branch delay of one cycle and a load delay of one cycle. The one-cycle branch delay is a result of the branch comparison logic operating during the 1A pipeline stage of the branch. This allows the branch-target address calculated in the previous stage to be used for the instruction access in the following 1I stage. Figure 4.4 illustrates the branch delay.


One Cycle
One Cycle
One Cycle
One Cycle
One Cycle
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
*
**
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
Branch Delay Slot

* Branch and fall-through address calculated
** Address selection made

**Figure 4.4 CPU-Pipeline Branch Delay**

## Load Delay

The completion of a load at the end of the 2D pipeline stage produces an operand that is available for the 1A pipeline stage of the subsequent instruction following the load delay slot. Figure 4.5 shows the load delay.

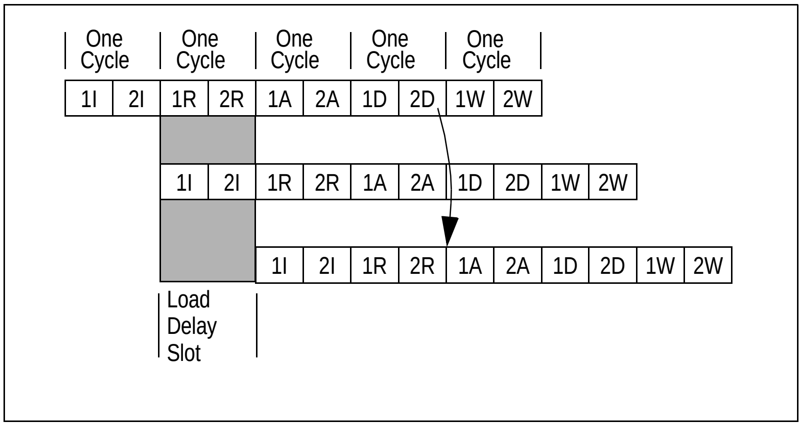

One Cycle
One Cycle
One Cycle
One Cycle
One Cycle
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
Load Delay Slot

**Figure 4.5 CPU-Pipeline Load Delay**

## Interlock and Exception Handling

Smooth pipeline flow is interrupted when cache misses or exceptions occur, or when data dependencies are detected. Interruptions handled using hardware, such as cache misses, are referred to as *interlocks*, while those that are handled using software are called *exceptions*.

There are two types of interlocks:

- *Stalls, which are resolved by halting the pipeline.*
- *Slips, which require one part of the pipeline to advance while another part of the pipeline is held static.*

At each cycle, exception and interlock conditions are checked for all active instructions. Because each exception or interlock condition corresponds to a particular pipeline stage, a condition can be traced back to the particular instruction in the exception/interlock stage. For instance, a Reserved Instruction (RI) exception is raised in the execution (A) stage.

| **State** | **Pipeline Stage** | | | | |
|---|---|---|---|---|---|
| | **I** | **R** | **A** | **D** | **W** |
| Stall | ITM | ICM | | DCM | |
| | | | | CPE | |
| Slip | | LDI | | | |
| | | MDSt | | | |
| | | FCBusy | | | |
| Exceptions | ITLB | IBE | RI | DBE | |
| | | IPErr | CUn | NMI | |
| | | | BP | Reset | |
| | | | SC | DPErr | |
| | | | DTLB | OVF | |
| | | | DTMod | Trap | |
| | | | Intr | | |

**Table 4.1 Relationship of CPU-Pipeline Stage to Interlock Condition**

Notes

| Exception | Description |
|---|---|
| ITLB | Instruction Translation or Address Exception |
| Intr | External Interrupt |
| IBE | IBus Error |
| RI | Reserved Instruction |
| BP | Breakpoint |
| SC | System Call |
| CUn | Coprocessor Unusable |
| IPErr | Instruction Parity Error |
| OVF | Integer Overflow |
| FPE | FP Interrupt |
| ExTrap | EX Stage Traps |
| DTLB | Data Translation or Address Exception |
| TLBMod | TLB Modified |
| DBE | Data Bus Error |
| DPErr | Data Parity Error |
| NMI | Non-maskable Interrupt |
| Reset | Reset |

**Table 4.2 CPU-Pipeline Exceptions**

| Interlock | Description |
|---|---|
| ITM | Instruction TLB Miss |
| ICM | Instruction Cache Miss |
| CPBE | Coprocessor Possible Exception |
| DCM | Data Cache Miss |
| LDI | Load Interlock |
| MDSt | Multiply/Divide Start |
| FCBsy | FP Busy |

**Table 4.3 CPU-Pipeline Interlocks**

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

When an exception condition is detected, the processor aborts the instruction which caused the exception, as well as all subsequent instructions. When this instruction reaches the W stage, three events occur;

- *The exception flag causes the instruction to write various CP0 registers with the exception state,*
- *The current PC is changed to the appropriate exception vector address,*
- *The exception bits of earlier pipeline stages are cleared.*

Notes

This implementation allows all instructions which occurred before the exception to complete, and all instructions which occurred after the instruction to be aborted. Hence the value of the EPC is such that execution can be restarted. In addition, all exceptions are guaranteed to be taken in order. Figure 4.6 illustrates the exception detection mechanism for a Reserved Instruction (RI) exception.

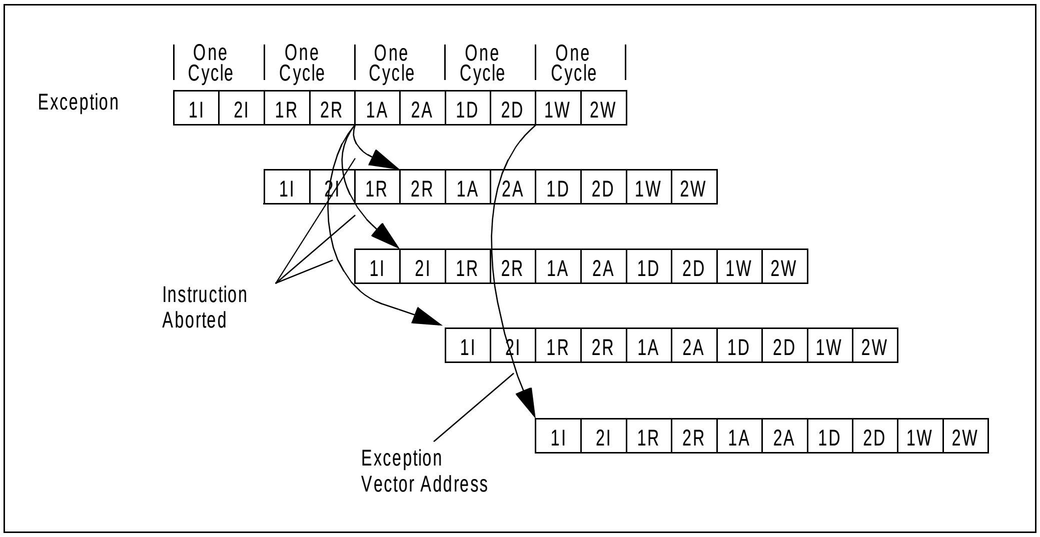
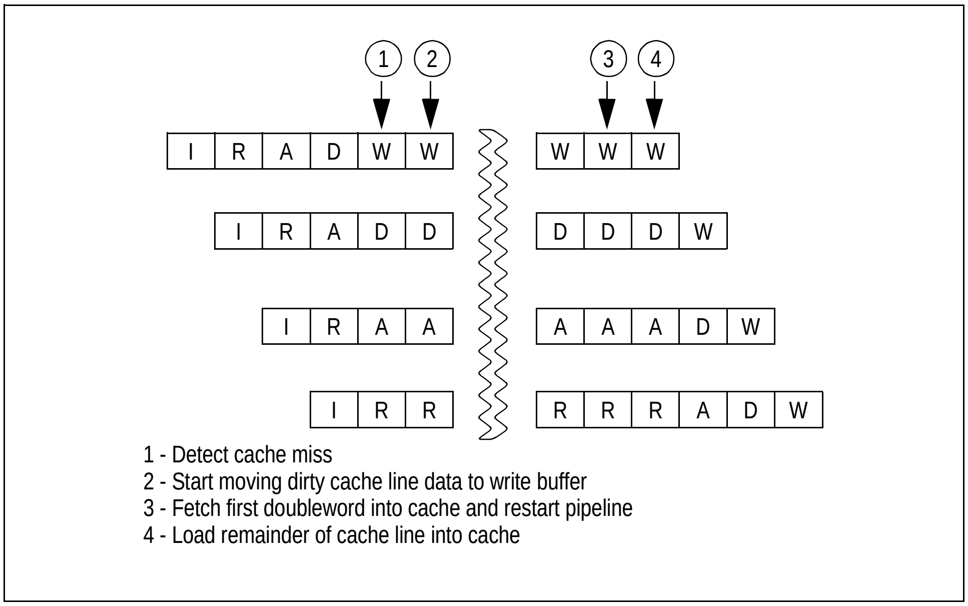

One Cycle
One Cycle
One Cycle
One Cycle
One Cycle
Exception
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
Instruction Aborted
Exception Vector Address

**Figure 4.6 CPU-Pipeline Exception Detection Mechanism**

## Stall Conditions

A stall condition is used to suspend the pipeline for conditions detected after the R pipeline stage. When a stall occurs, the processor resolves the condition and then restarts the pipeline. Once the interlock is removed, the restart sequence begins two cycles before the pipeline resumes execution. The restart sequence reverses the pipeline overrun by inserting the correct information into the pipeline. Figure 4.7 shows a data cache miss stall.



1 - Detect cache miss
2 - Start moving dirty cache line data to write buffer
3 - Fetch first doubleword into cache and restart pipeline
4 - Load remainder of cache line into cache

**Figure 4.7 CPU-Pipeline Servicing of Data Cache Miss**

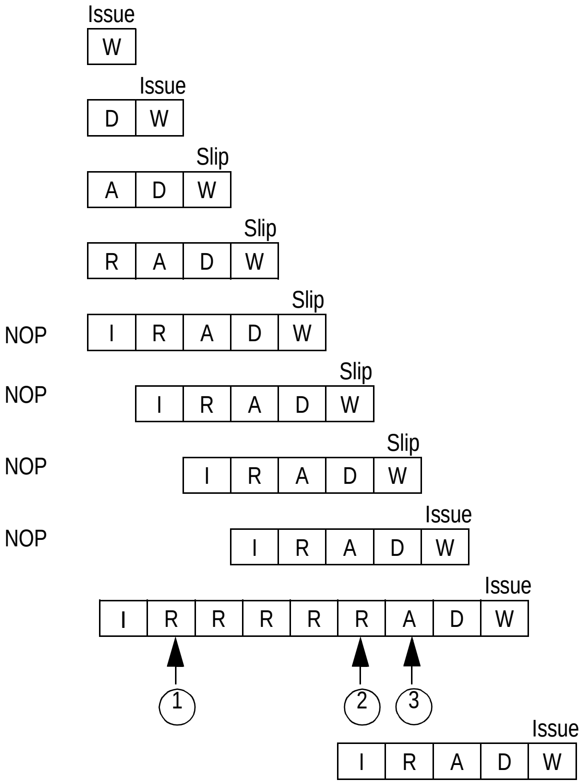
Notes

The data cache miss is detected in the D stage of the pipeline. If the cache line to be replaced is dirty, the W bit is set and data is moved to the internal write buffer in the next cycle. The squiggly line in Figure 4.7 indicates the memory access. Once the memory is accessed and the first doubleword of data is returned, the pipeline is restarted. The remainder of the cache line is returned in subsequent cycles. The dirty data in the write buffer is written out to memory after the cache-line fill is completed.

## Slip Conditions

During the 2R and 1A pipeline stages, internal logic determines whether it is possible to start the current instruction in this cycle. If all required source operands are available, as well as all hardware resources needed to complete the operation, the instruction is issued. Otherwise, the instruction slips. Slipped cycles are retried on subsequent cycles until they are issued. Pipeline stages D and W advance normally during slips in an attempt to resolve the conflict. NOPs are automatically inserted into the bubbles which are created in the pipeline. Instructions caused by “branch likely” instructions, ERET, or exceptions do not cause slips.

Figure 4.8 shows how an instruction can slip during an instruction-cache miss.



1 - Detect cache miss
2 - Start moving dirty cache line data to write buffer
3 - Fetch first doubleword into cache and restart pipeline
4 - Load remainder of cache line into cache

**Figure 4.8 Slips During Instruction-Cache Miss**

Instruction-cache misses are detected in the R-stage of the pipeline. Slips are detected in the A stage. Instruction-cache misses never require a writeback operation because writes are not allowed to the instruction cache. Unlike the data cache, early restart, where the pipeline is restarted after only a portion of the cache-line fill has occurred, is not implemented for the instruction cache. The requested cache line is loaded into the instruction cache in its entirety before the pipeline is restarted.

Notes

## Write Buffer

The processor has a write buffer which improves the performance of write operations to external memory. All write cycles use the write buffer. The write buffer holds up to four 64-bit address and data pairs.

On a cache miss requiring a write-back, the entire buffer is used for the write-back data and allows the processor to proceed in parallel with the memory update. For uncached and write-through stores, the write buffer decouples the CPU from the write to memory. If the write buffer is full, additional stores are stalled until there is room for them in the write buffer.

# Integer (CPU) Exceptions

Notes

## Introduction

This section describes the integer exception processing done by the CPU, including an explanation of exception processing, followed by the format and use of each CPU exception register. FPU exception processing is described in a later chapter.

The processor receives exceptions from a number of sources, including translation lookaside buffer (TLB) misses, arithmetic overflows, I/O interrupts, and system calls. When the CPU detects one of these exceptions, the normal sequence of instruction execution is suspended and the processor enters Kernel mode. The processor then disables interrupts and forces execution of a software exception processor (called a *handler*) located at a fixed address. The handler typically saves the context of the processor, including the contents of the program counter, the current operating mode (User or Supervisor), and the status of the interrupts (enabled or disabled). This context is saved so it can be restored when the exception has been serviced.

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

## Exception Processing Registers

The *System Control Coprocessor (CP0)* registers are used in exception processing. Table 5.1 lists these registers and their unique *register numbers*. For instance, the *ECC* register is register number 26. The remaining CP0 registers are used in memory management and are described in Chapter 7.

Software examines the CP0 registers during exception processing to determine the cause of the exception and the state of the CPU at the time the exception occurred. The registers in Table 5.1 are used in exception processing, and are described in the sections that follow.

| Register Name | R/W | Register Number |
|---|---|---|
| Context | R/W | 4 |
| BadVAddr (Bad Virtual Address) | R | 8 |
| Count | R/W | 9 |
| Compare register | W[1] | 11 |
| Status | R/W | 12 |
| Cause | R/W | 13 |
| EPC (Exception Program Counter) | R/W | 14 |
| XContext | R/W | 20 |
| ECC | R/W | 26 |
| CacheErr (Cache Error and Status) | R | 27 |
| ErrorEPC (Error Exception Program Counter) | R/W | 30 |

[1] For diagnostics, this register is R/W.

Table 5.1 CPO Exception Processing Registers

Notes

CPU general registers are interlocked and the result of an instruction can normally be used by the next instruction; if the result is not available right away, the processor stalls until it is available. CP0 registers and the TLB are not interlocked, however; there may be some delay before a value written by one instruction is available to following instructions. This delay may need to be explicitly coded in software.

### Context Register (4)

The *Context* register is a read/write register containing the pointer to an entry in the page table entry (PTE) array; this array is an operating system data structure that stores virtual-to-physical address translations. When there is a TLB miss, the CPU loads the TLB with the missing translation from the PTE array. Normally, the operating system uses the *Context* register to address the current page map which resides in the kernel-mapped segment, *kseg3*. The *Context* register duplicates some of the information provided in the *BadVAddr* register, but the information is arranged in a form that is more useful for a software TLB exception handler. Figure 5.1 shows the format of the *Context* register; Table 5.2 describes the *Context* register fields.

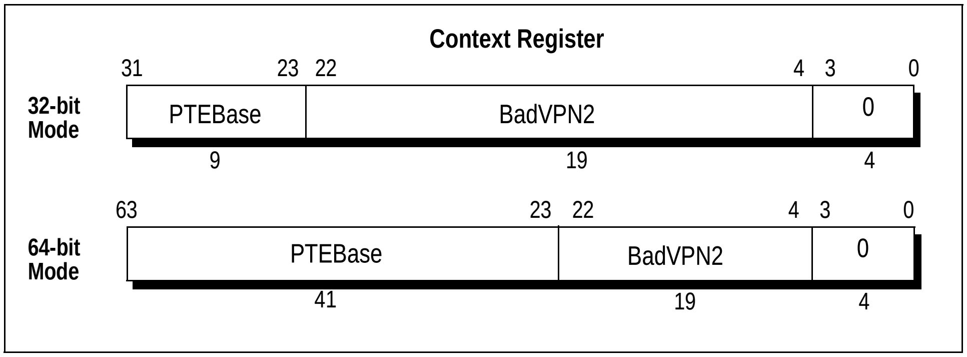


**Figure 5.1 Context Register Format**

| Field | R/W | Description |
|---|---|---|
| BadVPN2 | R | This field is written by hardware on a miss. It contains the virtual page number (VPN) of the most recent virtual address that did not have a valid translation. |
| PTEBase | R/W | This field is for use by the operating system. It is normally written with a value that allows the operating system to use the *Context* register as a pointer into the current PTE array in memory. |

**Table 5.2 Context Register Fields**

The 19-bit *BadVPN2* field contains bits 31:13 of the virtual address that caused the TLB miss; bit 12 is excluded because a single TLB entry maps to an even-odd page pair. For a 4-Kbyte page size, this format can directly address the pair-table of 8-byte PTEs. For other page and PTE sizes, shifting and masking this value produces the appropriate address.

### Bad Virtual Address Register (BadVAddr) (8)

The Bad Virtual Address register (*BadVAddr*) is a read-only register that displays the most recent virtual address that caused one of the following exceptions: TLB Invalid, TLB Modified, TLB Refill, Virtual Coherency Data Access, or Virtual Coherency Instruction Fetch. Figure 5.2 shows the format of the *BadVAddr* register. The *BadVAddr* register does not save any information for bus errors, since bus errors are not addressing errors.

Notes

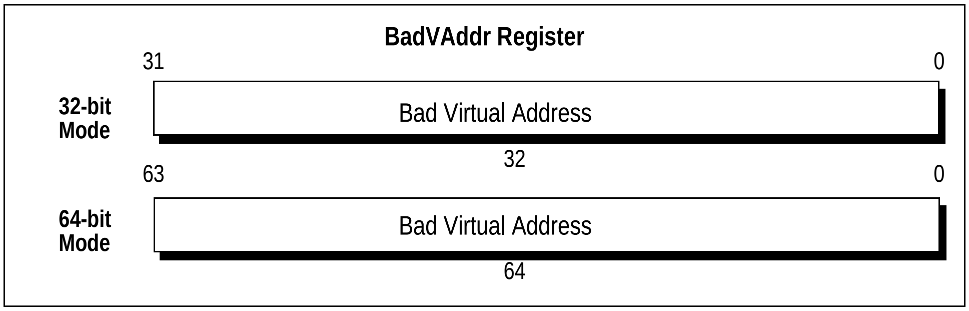


**Figure 5.2 BadVAddr Register Format**

### Count Register (9)

The *Count* register acts as a timer, incrementing at a constant rate—half the maximum instruction issue rate—whether or not an instruction is executed, retired, or any forward progress is made through the pipeline. This register can be read or written. It can be written for diagnostic purposes or system initialization; for example, to synchronize processors. Figure 5.3 shows the format of the *Count* register.

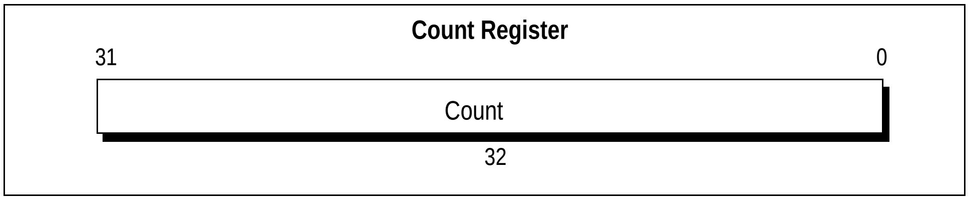


**Figure 5.3 Count Register Format**

### Compare Register (11)

The *Compare* register acts as a timer (see also the *Count* register); it maintains a stable value that does not change on its own. When the value of the *Count* register equals the value of the *Compare* register, interrupt bit *IP(7)* in the *Cause* register is set. This causes an interrupt as soon as the interrupt is enabled.

Writing a value to the *Compare* register, as a side effect, clears the timer interrupt. For diagnostic purposes, the *Compare* register is a read/write register. In normal use however, the *Compare* register is write-only. Figure 5.4 shows the format of the *Compare* register.

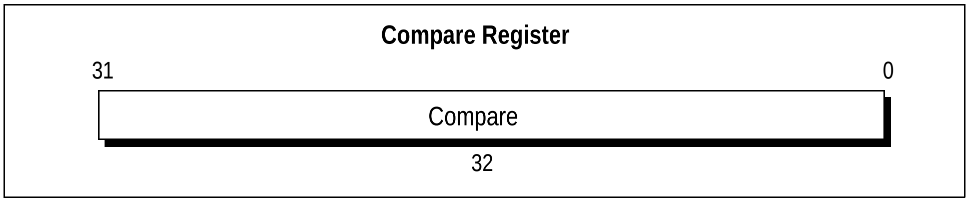


**Figure 5.4 Compare Register Format**

### Status Register (12)

The *Status* register (SR) is a read/write register that contains the operating mode, interrupt enabling, and the diagnostic states of the processor. The following list describes the more important *Status* register fields; Figures 34 and 35 show the format of the entire register, including descriptions of the fields. Some of the important fields include:

- *The 8-bit Interrupt Mask (IM) field controls the enabling of eight interrupt conditions. Interrupts must be enabled before they can be asserted, and the corresponding bits are set in both the Interrupt Mask field of the Status register and the Interrupt Pending field of the Cause register. IM[1:0] are software interrupt masks, while IM[7:2] correspond to Int[5:0].*
- *The 4-bit Coprocessor Usability (CU) field controls the usability of 4 possible coprocessors. Regardless of the CU0 bit setting, CP0 is always usable in Kernel mode. For all other cases, an access to an unusable coprocessor causes an exception.*

Notes

- *The 9-bit Diagnostic Status (DS) field is used for self-testing, and checks the cache and virtual memory system.*
- *The Reverse-Endian (RE) bit, bit 25, reverses the endianness of the machine. The processor can be configured as either little-endian or big-endian at system reset; reverse-endian selection is used in Kernel and Supervisor modes, and in the User mode when the RE bit is 0. Setting the RE bit to 1 inverts the User mode endianness.*

Figure 5.5 shows the format of the *Status* register. Table 5.3 describes the *Status* register fields. Figure 5.6 and Table 5.4 provide additional information on the *Diagnostic Status* (*DS*) field. All bits in the *DS* field except *TS* are readable and writable.

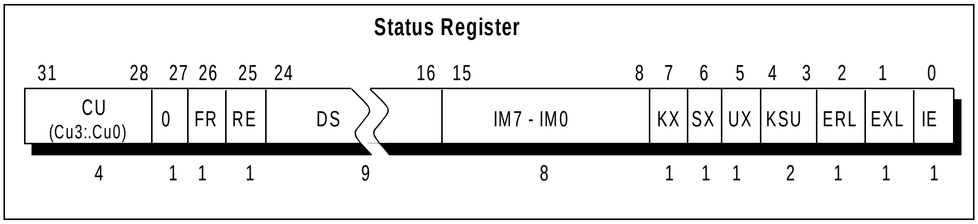


Figure 5.5 Status Register

| Field | Description |
|---|---|
| CU | Controls the usability of each of the four coprocessor unit numbers. CP0 is always usable when in Kernel mode, regardless of the setting of the $CU_0$ bit. Setting $CU_3$ enables the MIPS IV instruction set,<br>1 → usable<br>0 → unusable |
| 0 | Reserved. Set to 0. |
| FR | Enables additional floating-point registers<br>0 → 16 registers<br>1 → 32 registers |
| RE | *Reverse-Endian* bit, valid in User mode. |
| DS | *Diagnostic Status* field (see Figure 5.6). |
| IM | *Interrupt Mask*: controls the enabling of each of the external, internal, and software interrupts. An interrupt is taken if interrupts are enabled, and the corresponding bits are set in both the *Interrupt Mask* field of the *Status* register and the *Interrupt Pending* field of the *Cause* register.<br>0 → disabled<br>1→ enabled |
| KX | Enables 64-bit addressing in Kernel mode. The extended-addressing TLB refill exception is used for TLB misses on kernel addresses.<br>0 → 32–bit<br>1 → 64–bit |
| SX | Enables 64-bit addressing and operations in Supervisor mode. The extended-addressing TLB refill exception is used for TLB misses on supervisor addresses.<br>0 → 32–bit<br>1 → 64–bit |
| UX | Enables 64-bit addressing and operations in User mode. The extended-addressing TLB refill exception is used for TLB misses on user addresses.<br>0 → 32–bit<br>1 → 64–bit |
| KSU | Mode bits<br>$10_2$ → User<br>$01_2$ → Supervisor<br>$00_2$ → Kernel |

Table 5.3 Status Register Fields (Part 1 of 2)

## Notes

| Field | Description |
|---|---|
| ERL | Error Level; set by the processor when Reset, Soft Reset, NMI, or Cache Error exception are taken.<br>0 → normal<br>1 → error |
| EXL | Exception Level; set by the processor when any exception other than Reset, Soft Reset, NMI, or Cache Error exception are taken.<br>0 → normal<br>1 → exception |
| IE | Interrupt Enable<br>0 → disable interrupts<br>1 → enables interrupts |

**Table 5.3 Status Register Fields (Part 2 of 2)**

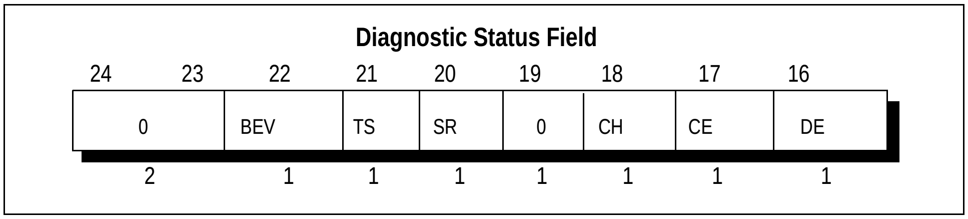


**Figure 5.6 Status Register DS Field**

| Bit | Description |
|---|---|
| BEV | Controls the location of TLB refill and general exception vectors.<br>0 → normal<br>1→ bootstrap |
| 0 | Reserved. Must be written as zeroes. Returns zeroes when read. |
| SR | 1→ Indicates that a soft reset or NMI has occurred. |
| CH | Hit (tag match and valid state) or miss indication for last CACHE Hit Invalidate, Hit Write Back Invalidate, Hit Write Back, Hit Set Virtual, or Create Dirty Exclusive for a secondary cache.<br>0 → miss<br>1 → hit |
| CE | Contents of the ECC register set or modify the check bits of the caches when CE = 1; see description of the *ECC* register. |
| DE | Specifies that cache parity or ECC errors cannot cause exceptions.<br>0 → parity/ECC remain enabled<br>1 → disables parity/ECC |
| 0 | Reserved. Must be written as zeroes, and returns zeroes when read. |

**Table 5.4 Status Register Diagnostic Status Bits**

Fields of the *Status* register set the following modes and access states:

- *Interrupt Enable: Interrupts are enabled by the settings of the IM bits when all of the following conditions are true:*
  - *IE = 1*
  - *EXL = 0*
  - *ERL = 0*
- *Operating Modes: The following CPU Status register bit settings are required for User, Kernel, and Supervisor modes.*
  - *User Mode: KSU = $10_2$, EXL = 0, and ERL = 0.*
  - *Supervisor Mode: KSU = $01_2$, EXL = 0, and ERL = 0.*
  - *Kernel Mode: KSU = $00_2$, or EXL = 1, or ERL = 1.*

Notes

- *32- and 64-bit Modes: The following CPU Status register bit settings select 32- or 64-bit operation for User, Kernel, and Supervisor operating modes. Enabling 64-bit operation permits the execution of 64-bit opcodes and translation of 64-bit addresses. 64-bit operation for User, Kernel and Supervisor modes can be set independently.*
  - *64-bit addressing for Kernel mode is enabled when KX = 1. 64-bit operations are always valid in Kernel mode.*
  - *64-bit addressing and operations are enabled for Supervisor mode when SX = 1.*
  - *64-bit addressing and operations are enabled for User mode when UX = 1.*

Access to the kernel address space is allowed when the processor is in Kernel mode. Access to the supervisor address space is allowed when the processor is in the Kernel or Supervisor operating mode. Access to the user address space is allowed in any of the three operating modes.

The contents of the *Status* register are undefined at reset, except for the *ERL* and *BEV* bits, which are set to 1. The *SR* bit distinguishes between the Reset exception and the Soft Reset exception (caused either by Reset* or Nonmaskable Interrupt [NMI]).

## Cause Register (13)

The 32-bit read/write *Cause* register describes the cause of the most recent exception. Figure 5.7 shows the fields of this register. Table 5.5 describes the *Cause* register fields. All bits in the *Cause* register, with the exception of the *IP(1:0)* bits, are read-only; *IP(1:0)* are used for software interrupts.

| Field | Description |
|---|---|
| BD | Indicates whether the last exception taken occurred in a branch-delay slot.<br>1 → delay slot<br>0 → normal |
| CE | Coprocessor unit number referenced when a Coprocessor Unusable exception is taken. |
| IP | Indicates an interrupt is pending.<br>1 → interrupt pending<br>0 → no interrupt |
| ExcCode | Exception code field (see Table 5.6) |
| 0 | Reserved. Must be written as zeroes, and returns zeroes when read. |

**Table 5.5 Cause Register Fields**

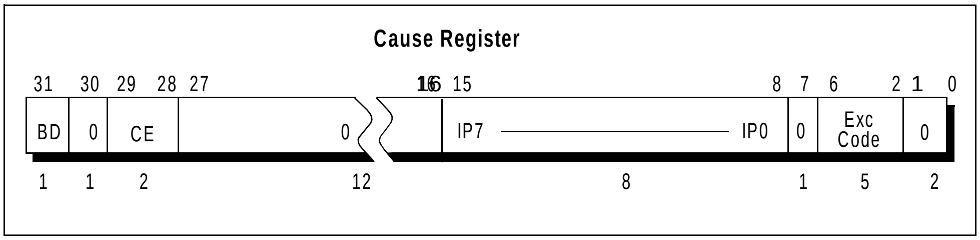


**Figure 5.7 Cause Register Format**

| Exception<br>Code Value | Mnemonic | Description |
|---|---|---|
| 0 | Int | Interrupt |
| 1 | Mod | TLB modification exception |
| 2 | TLBL | TLB exception (load or instruction fetch) |

**Table 5.6 Cause Register ExcCode Fields (Part 1 of 2)**

Notes

| Exception<br>Code Value | Mnemonic | Description |
|---|---|---|
| 3 | TLBS | TLB exception (store) |
| 4 | AdEL | Address error exception (load or instruction fetch) |
| 5 | AdES | Address error exception (store) |
| 6 | IBE | Bus error exception (instruction fetch) |
| 7 | DBE | Bus error exception (data reference: load or store) |
| 8 | Sys | Syscall exception |
| 9 | Bp | Breakpoint exception |
| 10 | RI | Reserved instruction exception |
| 11 | CpU | Coprocessor Unusable exception |
| 12 | Ov | Arithmetic Overflow exception |
| 13 | Tr | Trap exception |
| 14 | ---- | Reserved |
| 15 | FPE | Floating-Point exception |
| 16–31 | –-- | Reserved |

**Table 5.6 Cause Register ExcCode Fields (Part 2 of 2)**

### Exception Program Counter (EPC) Register (14)

The Exception Program Counter (*EPC*) is a read/write register that contains the address at which processing resumes after an exception has been serviced. For synchronous exceptions, the *EPC* register contains either:

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

The processor does not write to the *EPC* register when the *EXL* bit in the *Status* register is set to a 1. Figure 5.8 shows the format of the *EPC* register.

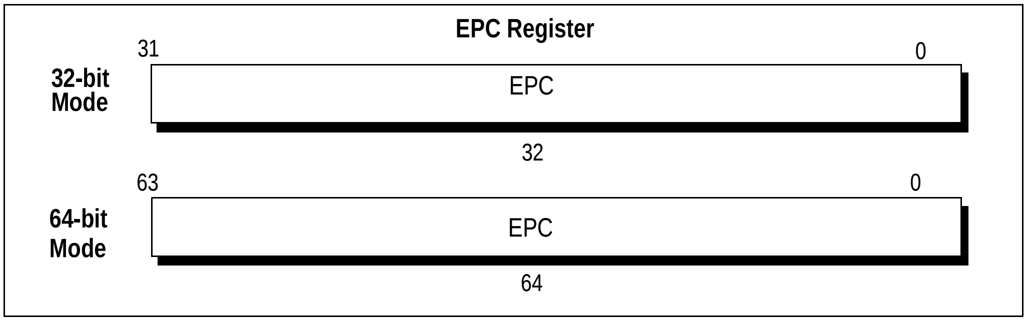


**Figure 5.8 EPC Register Format**

### XContext Register (20)

The read/write *XContext* register contains a pointer to an entry in the page table entry (PTE) array, an operating system data structure that stores virtual-to-physical address translations. When there is a TLB miss, the operating system software loads the TLB with the missing translation from the PTE array. The *XContext* register duplicates some of the information provided in the *BadVAddr* register, and puts it in a form useful for a software TLB exception handler. The *XContext* register is for use with the XTLB refill handler, which loads TLB entries for references to a 64-bit address space, and is included solely for oper-

**Notes**

ating system use. The operating system sets the PTE base field in the register, as needed. Normally, the operating system uses the *Context* register to address the current page map, which resides in the kernel-mapped segment *kseg3*. Figure 5.9 shows the format of the *XContext* register; Table 5.7 describes the *XContext* register fields.

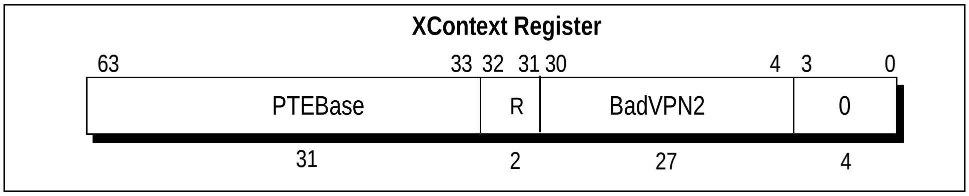


**Figure 5.9 XContext Register Format**

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

| Field | Description |
|---|---|
| BadVPN2 | The *Bad Virtual Page Number*/2 field is written by hardware on a miss. It contains the VPN of the most recent invalidly translated virtual address. |
| R | The *Region* field contains bits 63:62 of the virtual address.<br>$00_2$ = user<br>$01_2$ = supervisor<br>$11_2$ = kernel. |
| PTEBase | The *Page Table Entry Base* read/write field is normally written with a value that allows the operating system to use the *Context* register as a pointer into the current PTE array in memory. |

**Table 5.7 XContext Register Fields**

### Error Checking and Correcting (ECC) Register (26)

The 8-bit *Error Checking and Correcting* (*ECC*) register reads or writes primary-cache data parity bits for cache initialization, cache diagnostics, or cache error processing. (Tag ECC and parity are loaded from and stored to the *TagLo* register.) Figure 5.10 shows the format of the *ECC* register; Table 5.8 describes the register fields.

The *ECC* register is loaded by the Data Cache Index Load Tag operation. The content of the ECC register is:

- *written into the primary data cache on store instructions (instead of the computed parity) when the CE bit of the Status register is set.*
- *substituted for the computed instruction parity for the Instruction Cache Line Fill operation.*

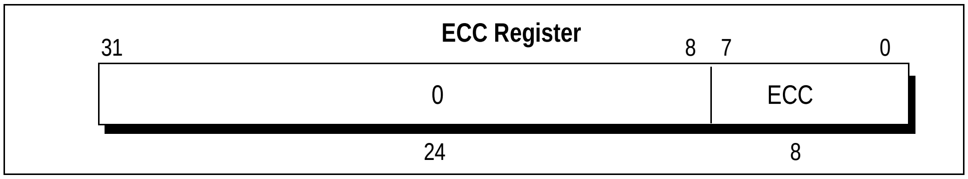


**Figure 5.10 ECC Register Format**

| Field | Description |
|---|---|
| ECC | An 8-bit field specifying the parity bits read from or written to a primary cache. |
| 0 | Reserved. Must be written as zeroes, and returns zeroes when read. |

**Table 5.8 ECC Register Fields**

Notes

### Cache Error (CacheErr) Register (27)

The 32-bit read-only *CacheErr* register processes ECC errors in the secondary cache and parity errors in the primary cache. The register holds cache index and status bits that indicate the source and nature of the error; it is loaded when a Cache Error exception is taken. Parity errors cannot be corrected. When a read response (cached or uncached) returns with bad parity, this exception is also taken.

Figure 5.11 shows the format of the *CacheErr* register and Table 5.9 describes the *CacheErr* register fields.

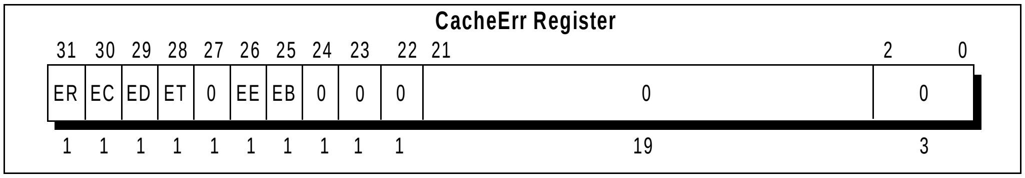


**Figure 5.11 CacheErr Register Format**

| Field | Description |
|---|---|
| ER | Type of reference<br>0 → instruction<br>1 → data |
| EC | Cache level of the error<br>0 → primary<br>1 → reserved |
| ED | Indicates if a data field error occurred<br>0 → no error<br>1 → error |
| ET | Indicates if a tag field error occurred<br>0 → no error<br>1 → error |
| EE | This bit is set if the error occurred on the SysAD bus. |
| EB | This bit is set if a data error occurred in addition to the instruction error (indicated by the remainder of the bits). If so, this requires flushing the data cache after fixing the instruction error. |
| 0 | Reserved. Must be written as zeroes, and returns zeroes when read. |

**Table 5.9 CacheErr Register Fields**

### Error Exception Program Counter (Error EPC) Register (30)

The read/write *ErrorEPC* register is similar to the *EPC* register, except that *ErrorEPC* is used on parity-error exceptions. It is also used to store the program counter (PC) on Reset, Soft Reset, and nonmaskable interrupt (NMI) exceptions. The *ErrorEPC* register contains the virtual address at which instruction processing can resume after servicing an error. This address can be:

- *the virtual address of the instruction that caused the exception*
- *the virtual address of the immediately preceding branch or jump instruction, when this address is in a branch-delay slot.*

There is no branch-delay slot indication for the *ErrorEPC* register. Figure 5.12 shows the format of the *ErrorEPC* register.

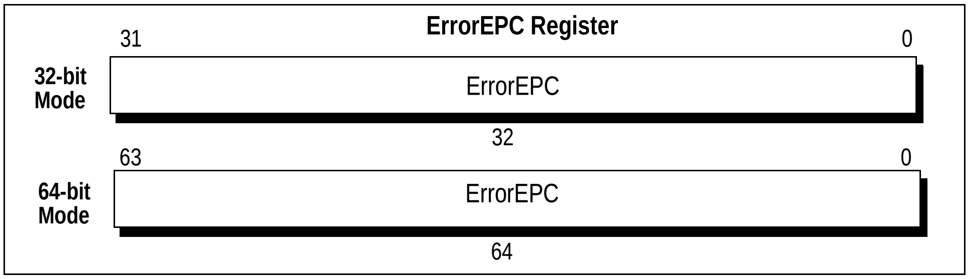


**Figure 5.12 ErrorEPC Register Format**

## Overview of Exception Types and Handling

When the *EXL* bit in the *Status* register is 0, either User, Supervisor, or Kernel operating mode is specified by the *KSU* bits in the *Status* register. When the *EXL* bit is a 1, the processor is in Kernel mode. When the processor takes an exception, the *EXL* bit is set to 1, which means the system is in Kernel mode. After saving the appropriate state, the exception handler typically changes *KSU* to Kernel mode and resets the *EXL* bit back to 0. When restoring the state and restarting, the handler restores the previous value of the *KSU* field and sets the *EXL* bit back to 1. Returning from an exception also resets the *EXL* bit to 0.

### Sample Hardware Processes For Various Exceptions

In the following sections, sample hardware processes for various exceptions are shown, together with the servicing required by the handler (software).

### Reset

Figure 5.13 shows the Reset exception process.

```
T:   undefined
     Random ¨ TLBENTRIES–1
     Wired ¨ 0
     Config ¨ 0 || EC || EP || 00000000 || BE || 110 || 010 || 1 || 1 || 0 || undefined
                    || DC || undefined$^6$
     ErrorEPC ¨ PC
     SR ¨ $SR_{31:23}$ || 1 || 0 || 0 || $SR_{19:3}$ || 1 || $SR_{1:0}$
     PC ¨ 0xFFFF FFFF BFC0 0000
```

**Figure 5.13 Reset Exception Processing**

### Cache Error

Figure 5.14 shows the Cache Error exception process.

```
T:   ErrorEPC ¨ PC
     CacheErr ¨ ER || EC || ED || ET || ES || EE || ED || $0^{25}$
     SR ¨ $SR_{31:3}$ || 1 ||$SR_{1:0}$
     if $SR_{22}$ = 1 then   /*What is the BEV bit setting*/
       PC ¨ 0xFFFF FFFF BFC0 0200 + 0x100 /*Access boot-PROM area*/
     else
       PC ¨ 0xFFFF FFFF A000 0000 + 0x100 /*Access main memory area*/
     endif
```

**Figure 5.14 Cache Error Exception Processing**

### Soft Reset and NMI

Figure 5.15 shows the Soft Reset and NMI exception process.

```
T:   ErrorEPC ¨ PC
     SR ¨ $SR_{31:23}$ || 1 || 0 || 1 || $SR_{19:3}$ || 1 || $SR_{1:0}$
     PC ¨ 0xFFFF FFFF BFC0 0000
```

**Figure 5.15 Soft Reset and NMI Exception Processing**

### General Exceptions

Figure 5.16 shows the process used for exceptions, other than Reset, Soft Reset, NMI, and Cache Error.

```
T:    Cause ¨ BD || 0 || CE || $0^{12}$ || $Cause_{15:8}$ || ExcCode || $0^2$
if $SR_1$ = 0 then   / * System is in User or Supervisor mode with no current exception */
                  EPC ¨ PC
     endif
     SR ¨ $SR_{31:2}$ || 1 || $SR_0$
     if $SR_{22}$ = 1 then
       PC ¨ 0xFFFF FFFF BFC0 0200 + vector /*access to uncached space*/
     else
       PC ¨ 0xFFFF FFFF 8000 0000 + vector /*access to cached space*/
     endif
```

**Figure 5.16 General Exception Processing**

### Exception Vector Locations

The Reset, Soft Reset, and NMI exceptions are always vectored to location 0xFFFF_FFFF_BFC0_0000. Addresses for all other exceptions are a combination of a *vector offset* and a *base address*. The vector associated with a *general exception* is called the *common exception vector;* its base address is determined by the BEV bit of the *Status* register.

Table 5.10 shows the 64-bit-mode vector base address for all exceptions; the 32-bit mode address is the low-order 32 bits (for instance, the base address for NMI in 32-bit mode is 0xBFC0 0000). Table 5.11 shows the vector offset added to the base address to create the exception address. When BEV = 0, the vector base address for the cache error exception changes from *kseg0* (0xFFFF FFFF 8000 0000) to *kseg1* (0xFFFF FFFF A000 0000). This change indicates that the caches are initialized and that the vector can be cached. When BEV = 1, the vector base for the cache error exception is 0xFFFF FFFF BFC0 0200. This is an uncached and unmapped space, allowing the exception to bypass the cache and the TLB.

| BEV Bit | R5000 Processor Vector Base Address |
|---|---|
| 0 | 0xFFFF FFFF 8000 0000 |
| 1 | 0xFFFF FFFF BFC0 0200 |

**Table 5.10 Exception Vector Base Address**

Notes

| Exception | R5000 Processor Vector Offset |
|---|---|
| TLB refill, EXL = 0 | 0x000 |
| XTLB refill, EXL = 0<br>(X = 64-bit TLB) | 0x080 |
| Cache Error | 0x100 |
| Others | 0x180 |
| Reset, Soft Reset, NMI | none |

**Table 5.11 Exception Vector Offsets**

### Priority of Exceptions

Table 5.12 describes exceptions in the order of highest to lowest priority. While more than one exception can occur for a single instruction, only the exception with the highest priority is reported. In generally, the exceptions described in the following sections are first processed by hardware, then serviced by software.

| |
|---|
| Reset *(highest priority)* |
| Soft Reset |
| Nonmaskable Interrupt (NMI) |
| Address error — Instruction fetch |
| TLB refill — Instruction fetch |
| TLB invalid — Instruction fetch |
| Cache error — Instruction fetch |
| Bus error — Instruction fetch |
| Integer overflow, Trap, System Call, Breakpoint, Reserved Instruction, Coprocessor Unusable, or Floating-Point Exception |
| Address error — Data access |
| TLB refill — Data access |
| TLB invalid — Data access |
| TLB modified — Data write |
| Cache error — Data access |
| Bus error — Data access |
| Interrupt *(lowest priority)* |

**Table 5.12 Exception Priority Order**

## Causes, Hardware Processing, and Software Servicing of Exceptions

### Reset Exception

#### Cause

The Reset exception occurs when the ColdReset* signal is asserted and then deasserted. This exception is not maskable.

#### Processing

The CPU provides a special interrupt vector for this exception:

- *location 0xFFFF_FFFF_BFC0_0000 in 64-bit mode.*

The Reset vector resides in unmapped and uncached CPU address space, so the hardware need not initialize the TLB or the cache to process this exception. It also means the processor can fetch and execute instructions while the caches and virtual memory are in an undefined state. The contents of all registers in the CPU are undefined when this exception occurs, except for the following register fields:

- *The Random register is initialized to the value of its upper bound.*
- *The Wired register is initialized to 0.*
- *Some Config register bits are initialized from the boot-time mode stream.*
- *In the Status register, SR is cleared to 0, and ERL and BEV are set to 1. All other bits are undefined.*

See Figure 5.13 for additional information on this process.

### Servicing

The Reset exception is serviced by:

- *initializing all processor registers, coprocessor registers, caches, and the memory system*
- *performing diagnostic tests*
- *bootstrapping the operating system*

## Soft Reset Exception

### Cause

The Soft Reset exception occurs in response to assertion of the Reset* input signal. Execution begins at the Reset vector when the Reset* signal is negated. The Soft Reset exception is not maskable.

### Processing

The Reset vector is used for this exception. The Reset vector is located within uncached and unmapped address space. Hence, the cache and TLB need not be initialized in order to process the exception. Regardless of the cause, when this exception occurs the *SR* bit of the *Status* register is set, distinguishing this exception from a Reset exception. Cache and memory states are undefined when the Soft Reset exception occurs because the Soft Reset can abort cache and bus operations.

The primary purpose of the Soft Reset exception is to reinitialize the processor after a fatal error during normal operation. Unlike an NMI, all cache and bus state machines are reset by this exception. When the Soft Reset exception occurs, all register contents are preserved with the following exceptions:

- *ErrorEPC register, which contains the restart PC.*
- *ERL, BEV, and SR bits of the Status Register, each of which is set to 1.*

See Figure 5.15 for additional information on this process.

### Servicing

The Soft Reset exception is serviced by saving the current processor state for diagnostic purposes, and reinitializing for the Reset exception.

## Non Maskable Interrupt (NMI) Exception

### Cause

The Non Maskable Interrupt exception occurs in response to the falling edge of the NMI signal, or an external write to the Int*[6] bit of the *Interrupt* Register. The NMI interrupt is not maskable and occurs regardless of the settings of the *EXL*, *ERL*, and *IE* bits in the *Status* Register.

Notes

### Processing

The Reset vector is used for this exception. The Reset vector is located within uncached and unmapped address space. Hence, the cache and TLB need not be initialized in order to process the exception. Regardless of the cause, when this exception occurs the *SR* bit of the *Status* register is set, distinguishing this exception from a Reset exception. Because the NMI can occur in the midst of another exception, it is typically not possible to continue program execution after servicing an NMI. An NMI exception is taken only at instruction boundaries. The state of the caches and memory system are preserved.

When the NMI exception occurs, all register contents are preserved with the following exceptions:

- *ErrorEPC register, which contains the restart PC.*
- *ERL, BEV, and SR bits of the Status Register, each of which is set to 1.*

See Figure 5.15 for additional information on this process.

### Servicing

The NMI exception is serviced by saving the current processor state for diagnostic purposes, and reinitializing for the Reset exception.

## Address-Error Exception

### Cause

The Address Error exception occurs when an attempt is made to execute one of the following:

- *load or store a doubleword that is not aligned on a doubleword boundary*
- *load, fetch, 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 or Supervisor mode*
- *reference the supervisor address space from User mode*

This exception is not maskable.

### Processing

The common exception vector is used for this exception. The *AdEL* or *AdES* code in the *Cause* register is set, indicating whether the instruction caused the exception with an instruction reference, load operation, or store operation shown by the *EPC* register and *BD* bit in the *Cause* register.

When this exception occurs, the *BadVAddr* register retains the virtual address that was not properly aligned or that referenced protected address space. The contents of the *VPN* field of the *Context* and *EntryHi* registers are undefined, as are the contents of the *EntryLo* register.

The *EPC* register contains the address of the instruction that caused the exception, unless this instruction is in a branch-delay slot. If it is in a branch-delay slot, the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set as indication.

### Servicing

The process executing at the time is handed a segmentation violation signal. This error is usually fatal to the process incurring the exception.

## TLB Exceptions

Three types of TLB exceptions can occur:

- *TLB Refill occurs when there is no TLB entry that matches an attempted reference to a mapped address space.*
- *TLB Invalid occurs when a virtual address reference matches a TLB entry that is marked invalid.*
- *TLB Modified occurs when a store operation virtual address reference to memory matches a TLB entry which is marked valid but is not dirty (the entry is not writable).*

The following three sections describe these TLB exceptions.

## TLB Refill Exception

### Cause

The TLB refill exception occurs when there is no TLB entry to match a reference to a mapped address space. This exception is not maskable.

### Processing

There are two special exception vectors for this exception; one for references to 32-bit address spaces, and one for references to 64-bit address spaces. The *UX, SX,* and *KX* bits of the *Status* register determine whether the user, supervisor or kernel address spaces referenced are 32-bit or 64-bit spaces. All references use these vectors when the *EXL* bit is set to 0 in the *Status* register. This exception sets the *TLBL* or *TLBS* code in the *ExcCode* field of the *Cause* register. This code indicates whether the instruction, as shown by the *EPC* register and the *BD* bit in the *Cause* register, caused the miss by an instruction reference, load operation, or store operation.

When this exception occurs, the *BadVAddr*, *Context*, *XContext* and *EntryHi* registers hold the virtual address that failed address translation. The *EntryHi* register also contains the ASID from which the translation fault occurred. The *Random* register normally contains a valid location in which to place the replacement TLB entry. The contents of the *EntryLo* register are undefined. The *EPC* register contains the address of the instruction that caused the exception, unless this instruction is in a branch-delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

### Servicing

To service this exception, the contents of the *Context* or *XContext* register are used as a virtual address to fetch memory locations containing the physical page frame and access control bits for a pair of TLB entries. The two entries are placed into the E*ntryLo0/EntryLo1* register; the *EntryHi* and *EntryLo* registers are written into the TLB.

It is possible that the virtual address used to obtain the physical address and access control information is on a page that is not resident in the TLB. This condition is processed by allowing a TLB refill exception in the TLB refill handler. This second exception goes to the common exception vector because the *EXL* bit of the *Status* register is set.

## TLB Invalid Exception

### Cause

The TLB invalid exception occurs when a virtual address reference matches a TLB entry that is marked invalid (TLB valid bit cleared). This exception is not maskable.

### Processing

The common exception vector is used for this exception. The *TLBL* or *TLBS* code in the *ExcCode* field of the *Cause* register is set. This indicates whether the instruction, as shown by the *EPC* register and *BD* bit in the *Cause* register, caused the miss by an instruction reference, load operation, or store operation.

When this exception occurs, the *BadVAddr*, *Context*, *XContext* and *EntryHi* registers contain the virtual address that failed address translation. The *EntryHi* register also contains the ASID from which the translation fault occurred. The *Random* register normally contains a valid location in which to put the replacement TLB entry. The contents of the *EntryLo* register is undefined.

The *EPC* register contains the address of the instruction that caused the exception unless this instruction is in a branch-delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

Notes

### Servicing

A TLB entry is typically marked invalid when one of the following is true:

- *a virtual address does not exist*
- *the virtual address exists, but is not in main memory (a page fault)*
- *a trap is desired on any reference to the page (for example, to maintain a reference bit)*

After servicing the cause of a TLB Invalid exception, the TLB entry is located with TLBP (TLB Probe), and replaced by an entry with that entry's *Valid* bit set.

## TLB Modified Exception

### Cause

The TLB modified exception occurs when a store operation virtual address reference to memory matches a TLB entry that is marked valid but is not dirty and therefore is not writable. This exception is not maskable.

### Processing

The common exception vector is used for this exception, and the *Mod* code in the *Cause* register is set. When this exception occurs, the *BadVAddr*, *Context*, *XContext* and *EntryHi* registers contain the virtual address that failed address translation. The *EntryHi* register also contains the ASID from which the translation fault occurred. The contents of the *EntryLo* register is undefined.

The *EPC* register contains the address of the instruction that caused the exception unless that instruction is in a branch-delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

### Servicing

The kernel uses the failed virtual address or virtual page number to identify the corresponding access control information. The page identified may or may not permit write accesses; if writes are not permitted, a write protection violation occurs.

If write accesses are permitted, the page frame is marked dirty/writable by the kernel in its own data structures. The TLBP instruction places the index of the TLB entry that must be altered into the *Index* register. The *EntryLo* register is loaded with a word containing the physical page frame and access control bits (with the *D* bit set), and the *EntryHi* and *EntryLo* registers are written into the TLB.

## Cache Error Exception

### Cause

The Cache Error exception occurs when either a primary or secondary cache parity error is detected. This exception is maskable by the *DE* bit in the Status Register. When a read response (cached or uncached) returns with bad parity, this exception is also taken.

### Processing

The processor sets the *ERL* bit in the *Status* register, saves the exception restart address in the *ErrorEPC* register, and then transfers the information to one of the following special vectors in uncached space:

- *If BEV = 0, the vector is 0xFFFF FFFF A000 0100.*
- *If BEV = 1, the vector is 0xFFFF FFFF BFC0 0300.*

See Figure 5.14 for additional information on this process.

Notes

### Servicing

All errors should be logged. To correct parity errors the system uses the CACHE instruction to invalidate the cache block, overwrite the old data through a cache miss, and resumes execution with an ERET. Other errors are not correctable and are likely to be fatal to the current process.

## Bus Error Exception

### Cause

A Bus Error exception is raised by board-level circuitry for events such as bus time-out, backplane bus parity errors, and invalid physical-memory addresses or access types. This exception is not maskable. A Bus Error exception occurs when a cache-miss refill, uncached reference, or an unbuffered write occurs synchronously; a Bus Error exception resulting from a buffered write transaction must be reported using the general interrupt mechanism.

### Processing

The common interrupt vector is used for a Bus Error exception. The *IBE* or *DBE* code in the *ExcCode* field of the *Cause* register is set, signifying whether the instruction (as indicated by the *EPC* register and *BD* bit in the *Cause* register) caused the exception by an instruction reference, load operation, or store operation.

The *EPC* register contains the address of the instruction that caused the exception, unless it is in a branch-delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

### Servicing

The physical address at which the fault occurred can be computed from information available in the CP0 registers:

- ◆ *If the IBE code in the Cause register is set (indicating an instruction fetch reference), the virtual address is contained in the EPC register.*
- ◆ *If the DBE code is set (indicating a load or store reference), the instruction that caused the exception is located at the virtual address contained in the EPC register (or 4+ the contents of the EPC register if the BD bit of the Cause register is set).*

The virtual address of the load and store reference can then be obtained by interpreting the instruction. The physical address can be obtained by using the TLBP instruction and reading the *EntryLo* register to compute the physical page number. The process executing at the time of this exception is handed a bus error signal, which is usually fatal.

## Integer Overflow Exception

### Cause

An Integer Overflow exception occurs when an ADD, ADDI, SUB, DADD, DADDI or DSUB instruction results in a 2’s complement overflow. This exception is not maskable.

### Processing

The common exception vector is used for this exception, and the *OV* code in the *Cause* register is set. The *EPC* register contains the address of the instruction that caused the exception unless the instruction is in a branch-delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

### Servicing

The process executing at the time of the exception is handed a floating-point exception/integer overflow signal. This error is usually fatal to the current process.

Notes

## Trap Exception

### Cause

The Trap exception occurs when a TGE, TGEU, TLT, TLTU, TEQ, TNE, TGEI, TGEUI, TLTI, TLTUI, TEQI, or TNEI instruction results in a TRUE condition. This exception is not maskable.

### Processing

The common exception vector is used for this exception, and the *Tr* code in the *Cause* register is set. The *EPC* register contains the address of the instruction causing the exception unless the instruction is in a branch-delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

### Servicing

The process executing at the time of a Trap exception is handed a floating-point exception/integer overflow signal. This error is usually fatal.

## System Call Exception

### Cause

A System Call exception occurs during an attempt to execute the SYSCALL instruction. This exception is not maskable.

### Processing

The common exception vector is used for this exception, and the *Sys* code in the *Cause* register is set. The *EPC* register contains the address of the SYSCALL instruction unless it is in a branch-delay slot, in which case the *EPC* register contains the address of the preceding branch instruction. If the SYSCALL instruction is in a branch-delay slot, the *BD* bit of the *Status* register is set; otherwise this bit is cleared.

### Servicing

When this exception occurs, control is transferred to the applicable system routine. To resume execution, the *EPC* register must be altered so that the SYSCALL instruction does not re-execute; this is accomplished by adding a value of 4 to the *EPC* register (*EPC* register + 4) before returning. If a SYSCALL instruction is in a branch-delay slot, a more complicated algorithm, beyond the scope of this description, may be required.

## Breakpoint Exception

### Cause

A Breakpoint exception occurs when an attempt is made to execute the BREAK instruction. This exception is not maskable.

### Processing

The common exception vector is used for this exception, and the *BP* code in the *Cause* register is set. The *EPC* register contains the address of the BREAK instruction unless it is in a branch-delay slot, in which case the *EPC* register contains the address of the preceding branch instruction. If the BREAK instruction is in a branch-delay slot, the *BD* bit of the *Status* register is set, otherwise the bit is cleared.

### Servicing

When the Breakpoint exception occurs, control is transferred to the applicable system routine. Additional distinctions can be made by analyzing the unused bits of the BREAK instruction (bits 25:6), and loading the contents of the instruction whose address the *EPC* register contains. A value of 4 must be added to the contents of the *EPC* register *(EPC* register + 4) to locate the instruction if it resides in a branch-delay slot.

Notes

To resume execution, the *EPC* register must be altered so that the BREAK instruction does not re-execute; this is accomplished by adding a value of 4 to the *EPC* register (*EPC* register + 4) before returning. If a BREAK instruction is in a branch-delay slot, interpretation of the branch instruction is required to resume execution.

## Reserved Instruction Exception

### Cause

The Reserved Instruction exception occurs when one of the following conditions occurs:

- *an attempt is made to execute an instruction with an undefined major opcode (bits 31:26).*
- *an attempt is made to execute a SPECIAL instruction with an undefined minor opcode (bits 5:0).*
- *an attempt is made to execute a REGIMM instruction with an undefined minor opcode (bits 20:16).*
- *an attempt is made to execute 64-bit operations in 32-bit mode when in User or Supervisor modes.*

64-bit operations are always valid in Kernel mode regardless of the value of the *KX* bit in the *Status* register. This exception is not maskable.

### Processing

The common exception vector is used for this exception, and the *RI* code in the *Cause* register is set. The *EPC* register contains the address of the reserved instruction unless it is in a branch-delay slot, in which case the *EPC* register contains the address of the preceding branch instruction.

### Servicing

No instructions in the MIPS ISA are currently interpreted. The process executing at the time of this exception is handed an illegal instruction/reserved operand fault signal. This error is usually fatal.

## Coprocessor Unusable Exception

### Cause

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

- *a corresponding coprocessor unit that has not been marked usable, or*
- *CP0 instructions, when the unit has not been marked usable and the process executes in either User or Supervisor mode.*

This exception is not maskable.

### Processing

The common exception vector is used for this exception, and the *CPU* code in the *Cause* register is set. The contents of the *Coprocessor Usage Error* field of the coprocessor *Control* register indicate which of the four coprocessors was referenced. The *EPC* register contains the address of the unusable coprocessor instruction unless it is in a branch-delay slot, in which case the *EPC* register contains the address of the preceding branch instruction.

### Servicing

The coprocessor unit to which an attempted reference was made is identified by the Coprocessor Usage Error field, which results in one of the following situations:

- *If the process is entitled access to the coprocessor, the coprocessor is marked usable and the corresponding user state is restored to the coprocessor.*
- *If the process is entitled access to the coprocessor, but the coprocessor does not exist or has failed, interpretation of the coprocessor instruction is possible.*
- *If the BD bit is set in the Cause register, the branch instruction must be interpreted; then the coprocessor instruction can be emulated and execution resumed with the EPC register advanced past*

Notes

*the coprocessor instruction.*

- *If the process is not entitled access to the coprocessor, the process executing at the time is handed an illegal instruction/privileged instruction fault signal. This error is usually fatal.*

### Floating-Point Exception

#### Cause

The Floating-Point exception is used by the floating-point coprocessor. This exception is not maskable.

#### Processing

The common exception vector is used for this exception, and the *FPE* code in the *Cause* register is set. The contents of the *Floating-Point Control/Status* register indicate the cause of this exception.

#### Servicing

This exception is cleared by clearing the appropriate bit in the *Floating-Point Control/Status* register. For an unimplemented instruction exception, the kernel should emulate the instruction; for other exceptions, the kernel should pass the exception to the user program that caused the exception.

### Interrupt Exception

#### Cause

The Interrupt exception occurs when one of the eight interrupt conditions is asserted. The significance of these interrupts is dependent upon the specific system implementation. Each of the eight interrupts can be masked by clearing the corresponding bit in the *Int-Mask* field of the *Status* register, and all of the eight interrupts can be masked at once by clearing the *IE* bit of the *Status* register.

#### Processing

The common exception vector is used for this exception, and the *Int* code in the *Cause* register is set. The *IP* field of the *Cause* register indicates current interrupt requests. It is possible that more than one of the bits can be simultaneously set (or even *no* bits may be set) if the interrupt is asserted and then deasserted before this register is read.

#### Servicing

If the interrupt is caused by one of the two software-generated exceptions (*SW1* or *SW0*), the interrupt condition is cleared by setting the corresponding *Cause* register bit to 0. If the interrupt is hardware-generated, the interrupt condition is cleared by correcting the condition causing the interrupt pin to be asserted.

Due to the on-chip write buffer, a store to an external device may not occur until after other instructions in the pipeline finish. Hence, the user must ensure that the store will occur before the *return from exception* instruction (ERET) is executed. Otherwise the interrupt may be serviced again even though there is no actual interrupt pending.

## Exception Handling and Servicing Flowcharts

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

- *general exceptions and their exception handler*
- *TLB/XTLB miss exception and their exception handler*
- *cache error exception and its handler*
- *reset, soft reset and NMI exceptions, and a guideline to their handler.*

Generally speaking, the exceptions are handled by hardware (HW); the exceptions are then serviced by software (SW).

## Notes

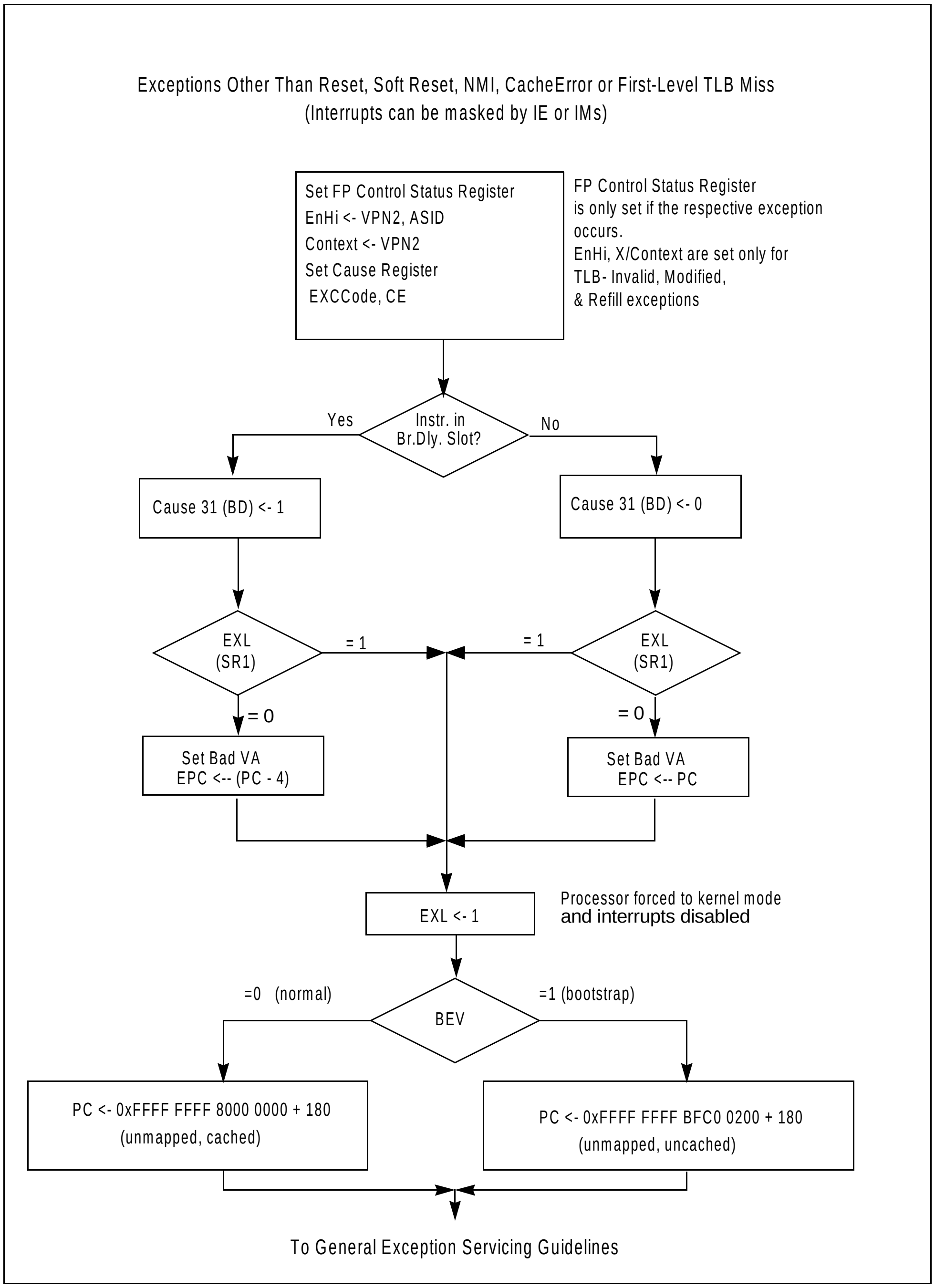

Exceptions Other Than Reset, Soft Reset, NMI, CacheError or First-Level TLB Miss
(Interrupts can be masked by IE or IMs)
Set FP Control Status Register
EnHi <- VPN2, ASID
Context <- VPN2
Set Cause Register
EXCCode, CE
FP Control Status Register is only set if the respective exception occurs.
EnHi, X/Context are set only for TLB- Invalid, Modified, & Refill exceptions
Instr. in Br.Dly. Slot?
Yes
No
Cause 31 (BD) <- 1
Cause 31 (BD) <- 0
EXL (SR1)
= 1
= 1
EXL (SR1)
= 0
= 0
Set Bad VA
EPC <-- (PC - 4)
Set Bad VA
EPC <-- PC
EXL <- 1
Processor forced to kernel mode and interrupts disabled
=0 (normal)
BEV
=1 (bootstrap)
PC <- 0xFFFF FFFF 8000 0000 + 180
(unmapped, cached)
PC <- 0xFFFF FFFF BFC0 0200 + 180
(unmapped, uncached)
To General Exception Servicing Guidelines

**Figure 5.17 General Exception Handler (HW)**

## Notes

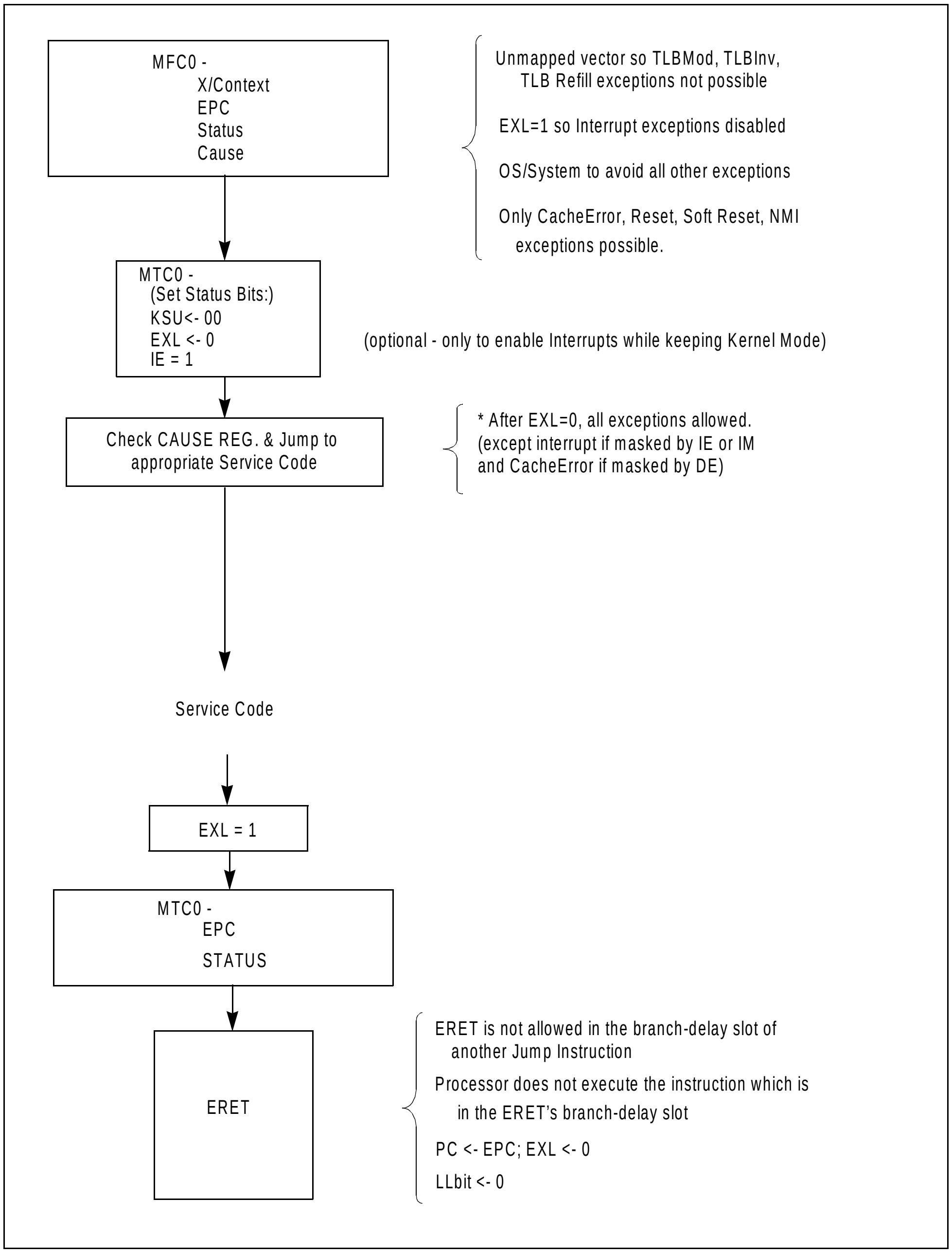

MFC0 -
X/Context
EPC
Status
Cause
Unmapped vector so TLBMod, TLBInv, TLB Refill exceptions not possible
EXL=1 so Interrupt exceptions disabled
OS/System to avoid all other exceptions
Only CacheError, Reset, Soft Reset, NMI exceptions possible.
MTC0 -
(Set Status Bits:)
KSU<- 00
EXL <- 0
IE = 1
(optional - only to enable Interrupts while keeping Kernel Mode)
Check CAUSE REG. & Jump to appropriate Service Code
* After EXL=0, all exceptions allowed. (except interrupt if masked by IE or IM and CacheError if masked by DE)
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 5.18 General Exception Servicing Guidelines (SW)**

## Notes

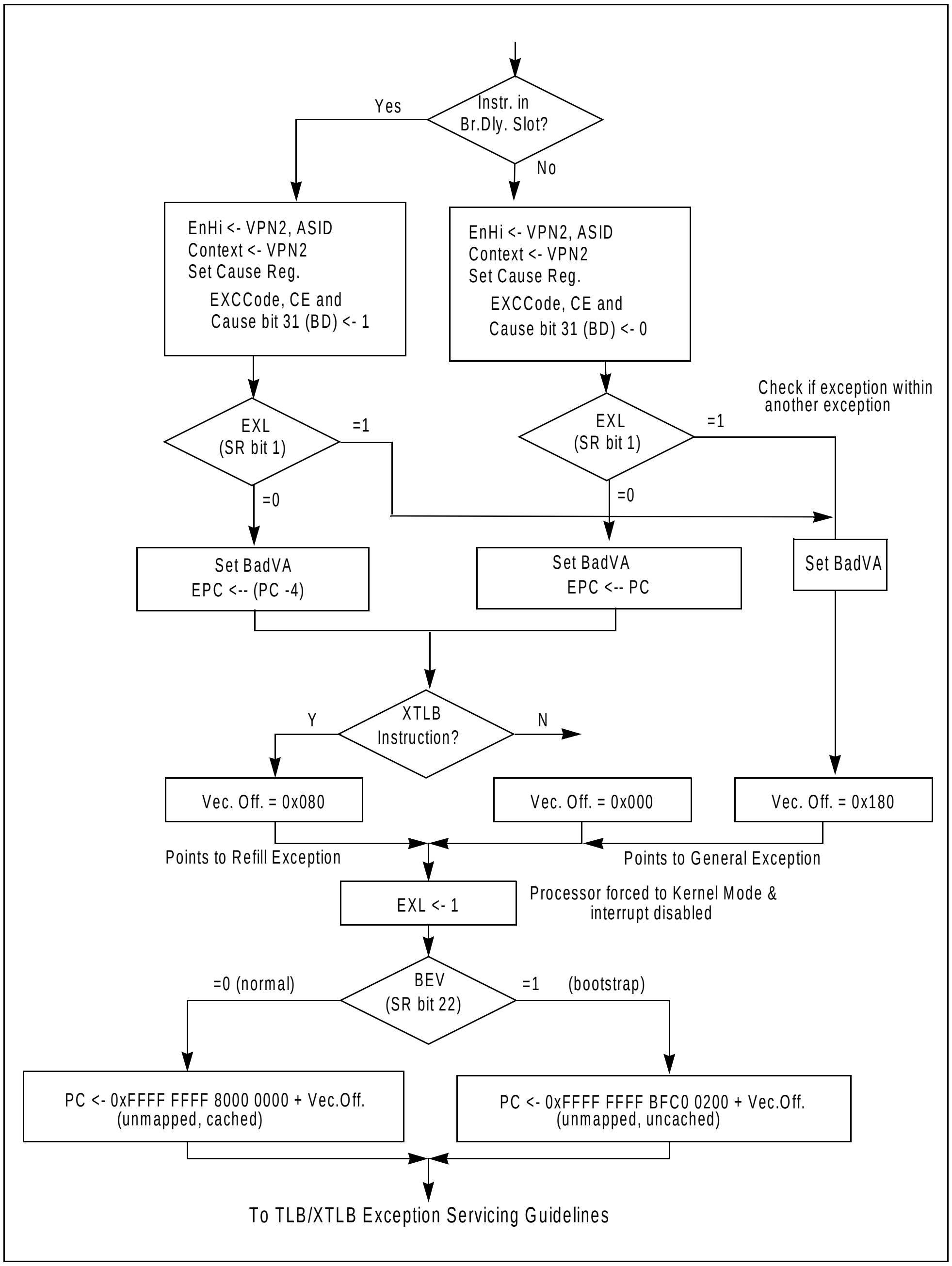


**Figure 5.19 TLB/XTLB Miss Exception Handler (HW)**

## Notes

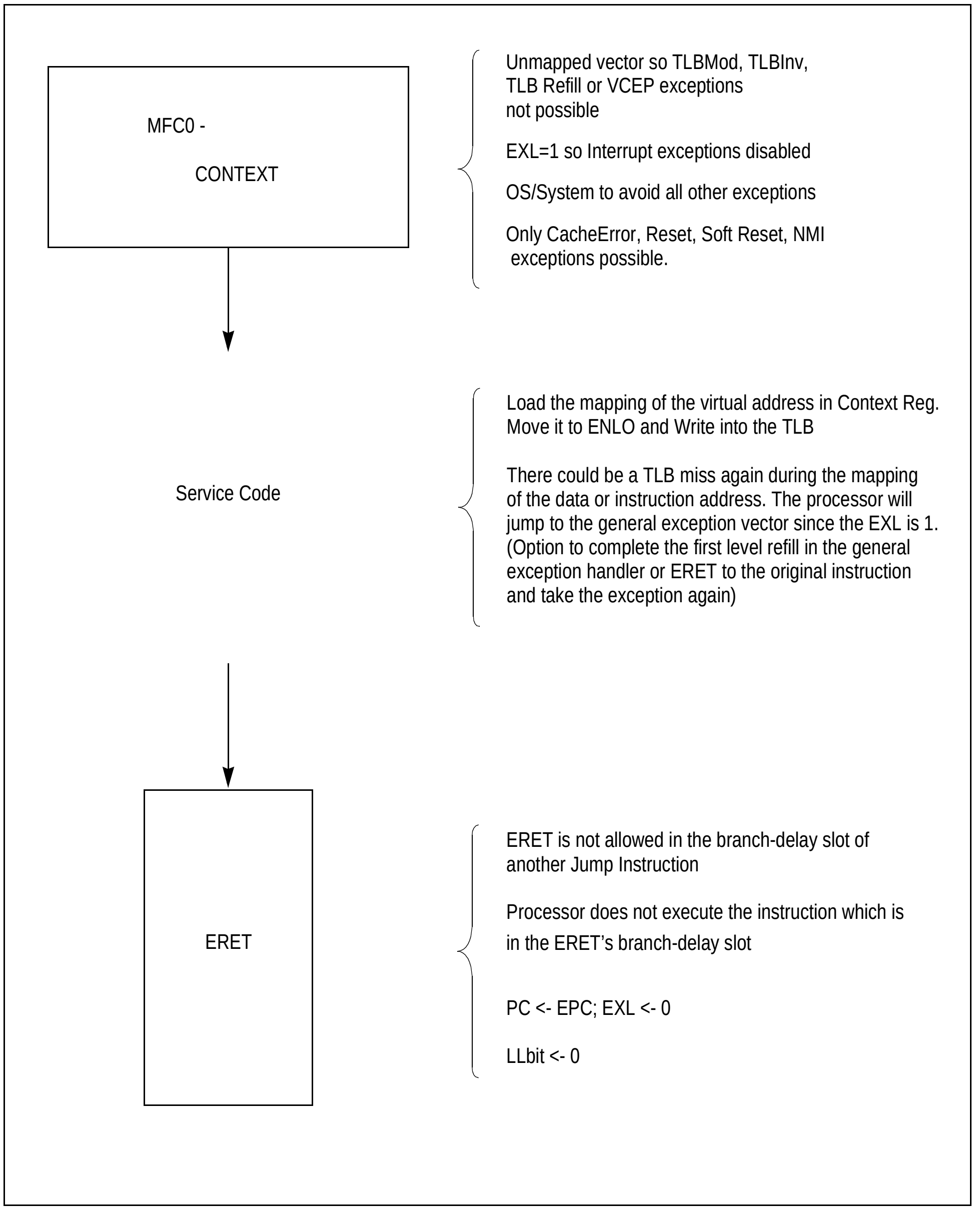

MFC0 -
CONTEXT
Unmapped vector so TLBMod, TLBInv, TLB Refill or VCEP exceptions not possible
EXL=1 so Interrupt exceptions disabled
OS/System to avoid all other exceptions
Only CacheError, Reset, Soft Reset, NMI exceptions possible.
Service Code
Load the mapping of the virtual address in Context Reg.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

**Figure 5.20 TLB/XTLB Exception Servicing Guidelines (SW)**

## Notes

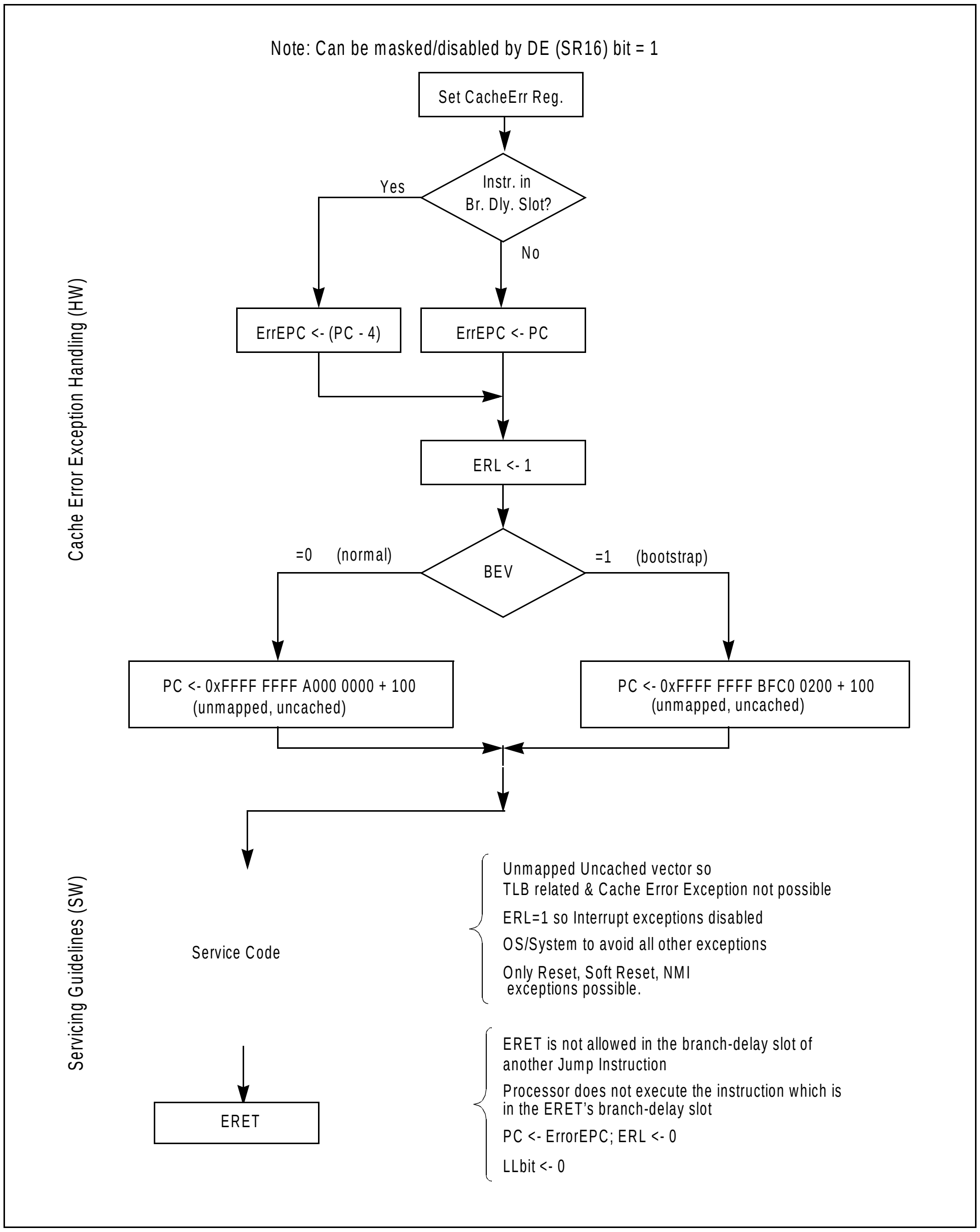

Note: Can be masked/disabled by DE (SR16) bit = 1
Set CacheErr Reg.
Instr. in Br. Dly. Slot?
Yes
No
ErrEPC <- (PC - 4)
ErrEPC <- PC
ERL <- 1
BEV
=0 (normal)
=1 (bootstrap)
PC <- 0xFFFF FFFF A000 0000 + 100 (unmapped, uncached)
PC <- 0xFFFF FFFF BFC0 0200 + 100 (unmapped, uncached)
Cache Error Exception Handling (HW)
Servicing Guidelines (SW)
Service Code
Unmapped Uncached vector so TLB related & Cache Error Exception not possible
ERL=1 so Interrupt exceptions disabled
OS/System to avoid all other exceptions
Only Reset, Soft Reset, NMI exceptions possible.
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 <- ErrorEPC; ERL <- 0
LLbit <- 0

**Figure 5.21 Cache Error Exception Handling (HW) and Servicing Guidelines**

## Notes

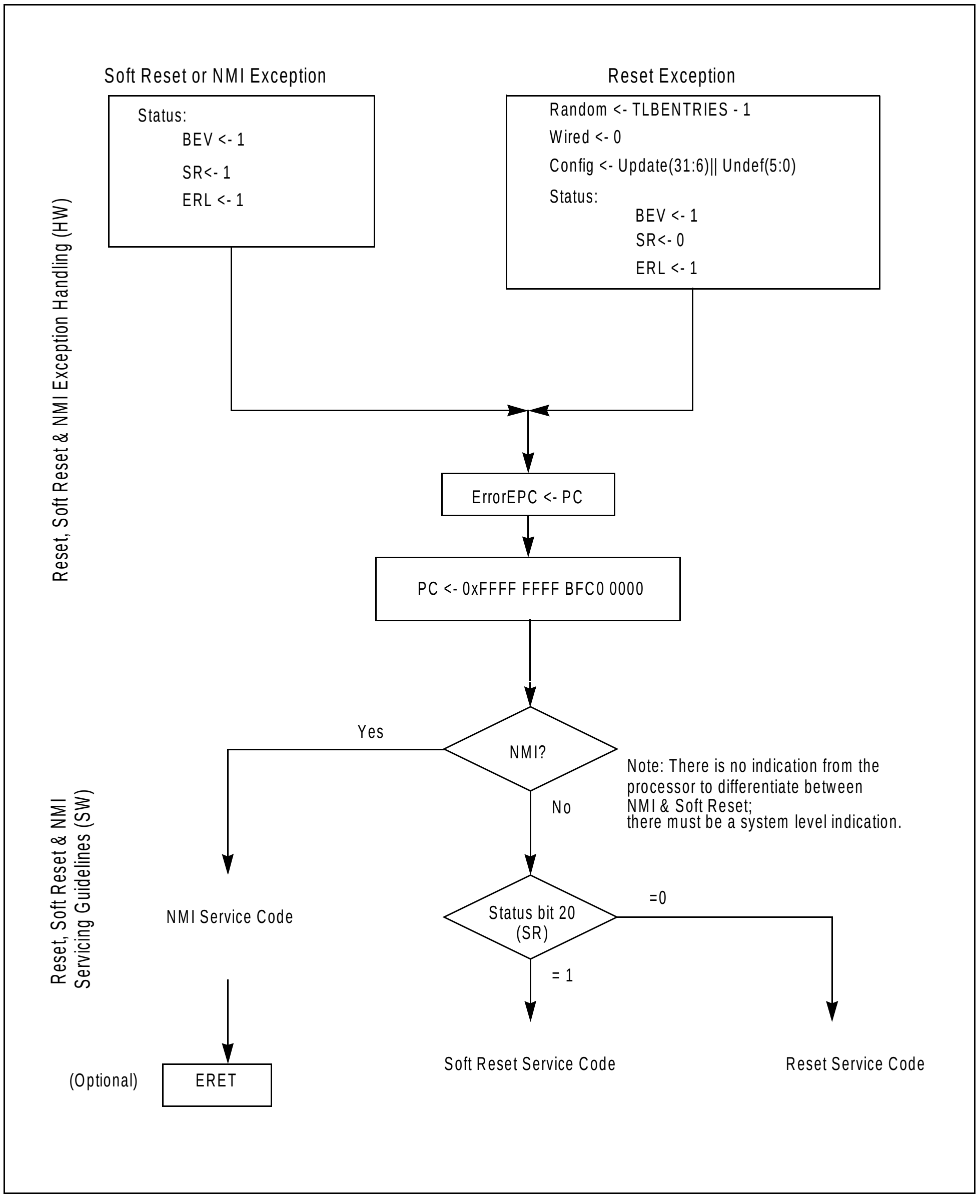

Reset, Soft Reset & NMI Exception Handling (HW)
Soft Reset or NMI Exception
Status:
BEV <- 1
SR<- 1
ERL <- 1
Reset Exception
Random <- TLBENTRIES - 1
Wired <- 0
Config <- Update(31:6)|| Undef(5:0)
Status:
BEV <- 1
SR<- 0
ERL <- 1
ErrorEPC <- PC
PC <- 0xFFFF FFFF BFC0 0000
Reset, Soft Reset & NMI Servicing Guidelines (SW)
NMI?
Yes
No
Note: There is no indication from the processor to differentiate between NMI & Soft Reset; there must be a system level indication.
NMI Service Code
Status bit 20 (SR)
=0
= 1
(Optional)
ERET
Soft Reset Service Code
Reset Service Code

Figure 5.22 Reset, Soft Reset & NMI Exception Handling

# Floating-Point (FPU) Exceptions

Notes

## Introduction

A floating-point exception occurs whenever the FPU cannot handle either the operands or the results of a floating-point operation in its normal way. The FPU responds by generating an exception to initiate a software trap or by setting a status flag.

## Exception Types

The FP *Control/Status* register described in Chapter 3 contains an *Enable* bit for each exception type. Exception *Enable* bits determine whether an exception will cause the FPU to initiate a trap or set a status flag:

- *If a trap is taken, the FPU remains in the state found at the beginning of the operation, and a software exception handling routine executes.*
- *If no trap is taken, an appropriate value is written into the FPU destination register and execution continues.*

The FPU supports the five IEEE Standard 754 exceptions:

- *Inexact (I)*
- *Underflow (U)*
- *Overflow (O)*
- *Division by Zero (Z)*
- *Invalid Operation (V)*

*Cause* bits, *Enables*, and *Flag* bits (status flags) are used. The FPU adds a sixth exception type, *Unimplemented Operation (E)*, to use when the FPU cannot implement the standard MIPS floating-point architecture, including cases in which the FPU cannot determine the correct exception behavior. This exception indicates the use of a software implementation. The Unimplemented Operation exception has no *Enable* or *Flag* bit; whenever this exception occurs, an unimplemented exception trap is taken (if the FPU interrupt input to the CPU is enabled).

Figure 6.1 illustrates the Control/Status register bits that support exceptions.

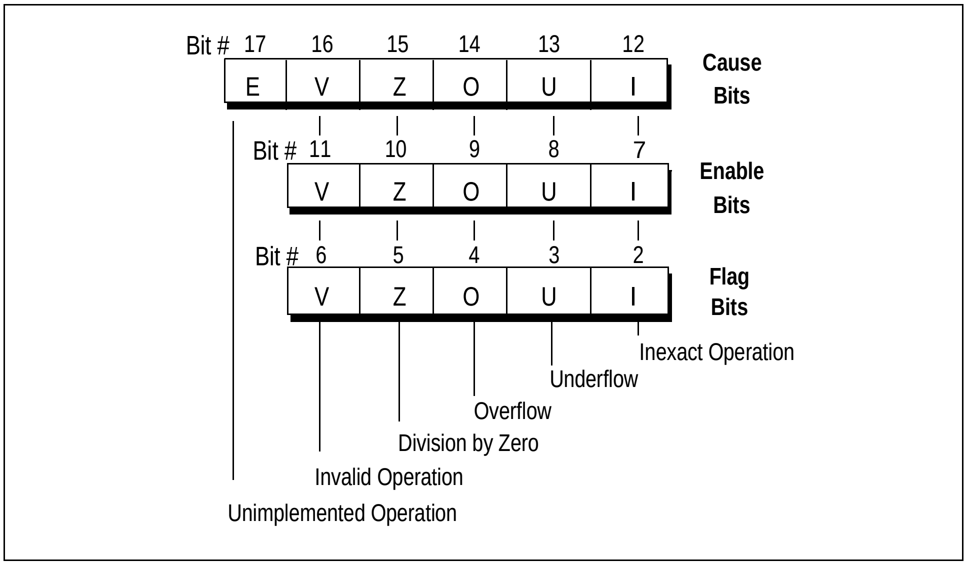


**Figure 6.1 Control/Status Register Exception/Flag/Trap/Enable Bits**

Notes

Each of the five IEEE Standard 754 exceptions (V, Z, O, U, I) is associated with a trap under user control, and is enabled by setting one of the five *Enable* bits. When an exception occurs, the corresponding *Cause* bit is set. If the corresponding *Enable* bit is not set, the *Flag* bit is also set. If the corresponding *Enable* bit is set, the *Flag* bit is not set and the FPU generates an interrupt to the CPU. Subsequent exception processing allows a trap to be taken.

## Exception Trap Processing

When a floating-point exception trap is taken, the *Cause* register indicates the floating-point coprocessor is the cause of the exception trap. The Floating-Point Exception (FPE) code is used, and the *Cause* bits of the floating-point *Control/Status* register indicate the reason for the floating-point exception. These bits are, in effect, an extension of the system coprocessor *Cause* register.

## Trap Handlers for IEEE Standard 754 Exceptions

The IEEE Standard 754 strongly recommends that users be allowed to specify a trap handler for any of the five standard exceptions that can compute; the trap handler can either compute or specify a substitute result to be placed in the destination register of the operation.

By retrieving an instruction using the processor *Exception Program Counter (EPC*) register, the trap handler determines:

- *exceptions occurring during the operation*
- *the operation being performed*
- *the destination format*

On Overflow or Underflow exceptions (except for conversions), and on Inexact exceptions, the trap handler gains access to the correctly rounded result by examining source registers and simulating the operation in software. On Overflow or Underflow exceptions encountered on floating-point conversions, and on Invalid Operation and Divide-by-Zero exceptions, the trap handler gains access to the operand values by examining the source registers of the instruction.

The IEEE Standard 754 recommends that, if enabled, the overflow and underflow traps take precedence over a separate inexact trap. This prioritization is accomplished in software; hardware sets the bits for both.

## Flags

A *Flag* bit is provided for each IEEE exception. This *Flag* bit is set to 1 on the assertion of its corresponding exception, with no corresponding exception trap signaled. The *Flag* bit is reset (cleared to 0) by writing a new value into the *Status* register. Flags can be saved and restored by software either individually or as a group.

When no exception trap is signaled, floating-point coprocessor takes a default action, providing a substitute value for the exception-causing result of the floating-point operation. The particular default action taken depends upon the type of exception. Table 6.1 lists the default action taken by the FPU for each of the IEEE exceptions.

Notes

| Field | Description | Rounding Mode | Default action |
|---|---|---|---|
| I | Inexact exception | Any | Supply a rounded result |
| U | Underflow exception | RN | Modify underflow values to 0 with the sign of the intermediate result |
| | | RZ | Modify underflow values to 0 with the sign of the intermediate result |
| | | RP | Modify positive underflows to the format's smallest positive finite number; modify negative underflows to -0 |
| | | RM | Modify negative underflows to the format's smallest negative finite number; modify positive underflows to 0 |
| O | Overflow exception | RN | Modify overflow values to ∞ with the sign of the intermediate result |
| | | RZ | Modify overflow values to the format's largest finite number with the sign of the intermediate result |
| | | RP | Modify negative overflows to the format's most negative finite number; modify positive overflows to + ∞ |
| | | RM | Modify positive overflows to the format's largest finite number; modify negative overflows to – ∞ |
| Z | Division by zero | Any | Supply a properly signed ∞ |
| V | Invalid operation | Any | Supply a quiet Not a Number (NaN) |

**Table 6.1 Default FPU Exception Actions**

Table 6.2 lists the exception-causing situations and contrasts the behavior of the FPU with the requirements of the IEEE Standard 754.

| FPA Internal Result | IEEE Standard 754 | Trap Enable | Trap Disable | Notes |
|---|---|---|---|---|
| Inexact result | I | I | I | Loss of accuracy |
| Exponent overflow | O,I[1] | O,I | O,I | Normalized exponent > $E_{max}$ |
| Division by zero | Z | Z | Z | Zero is (exponent = $E_{min}$-1, mantissa = 0) |
| Overflow on convert | V | E | E | Source out of integer range |
| Signaling NaN source | V | V | V | |
| Invalid operation | V | V | V | 0/0, etc. |
| Exponent underflow | U | E | E | Normalized exponent < $E_{min}$ |
| Denormalized or QNaN | None | E | E | Denormalized is (exponent = $E_{min}$-1 and mantissa <> 0) |

[1] The IEEE Standard 754 specifies an inexact exception on overflow only if the overflow trap is disabled.

**Table 6.2 FPU Exception-Causing Conditions**

## FPU Exceptions

The following sections describe the conditions that cause the FPU to generate each of its exceptions, and the FPU response to each exception-causing condition.

### Inexact Exception (I)

The FPU generates the Inexact exception if one of the following occurs:

- *the rounded result of an operation is not exact.*

Notes

- *the rounded result of an operation overflows.*
- *the rounded result of an operation underflows and both the Underflow and Inexact Enable bits are not set and the FS bit is set.*

The FPU usually examines the operands of floating-point operations before execution actually begins, to determine (based on the exponent values of the operands) if the operation can *possibly* cause an exception. If there is a possibility of an instruction causing an exception trap, the FPU uses a coprocessor stall to execute the instruction.

It is impossible, however, for the FPU to predetermine if an instruction will produce an inexact result. If Inexact exception traps are enabled, the FPU uses the coprocessor stall mechanism to execute all floating-point operations that require more than one cycle. Since this mode of execution can impact performance, Inexact exception traps should be enabled only when necessary.

The enabling of Inexact exception traps have the following results:

- *Trap Enabled: The result register is not modified and the source registers are preserved.*
- *Trap Disabled: The rounded or overflowed result is delivered to the destination register if no other software trap occurs.*

### Invalid Operation Exception (V)

The Invalid Operation exception is signaled if one or both of the operands are invalid for an implemented operation. When the exception occurs without a trap, the MIPS ISA defines the result as a quiet Not a Number (NaN). The invalid operations are:

- *Addition or subtraction: magnitude subtraction of infinities, such as: ( + ∞ ) + ( – ∞ ) or ( – ∞ ) – ( – ∞ ).*
- *Multiplication: 0 times ∞, with any signs.*
- *Division: 0/0, or ∞/∞, with any signs.*
- *Comparison of predicates involving* **<** *or* **>** *without* **?***, when the operands are unordered.*
- *Comparison or a Convert From Floating-point Operation on a signaling NaN.*
- *Any arithmetic operation on a signaling NaN. A move (MOV) operation is not considered to be an arithmetic operation, but absolute value (ABS) and negate (NEG) are considered to be arithmetic operations and cause this exception if one or both operands is a signaling NaN.*
- *Square root: √x, where x is less than zero.*

Software can simulate the Invalid Operation exception for other operations that are invalid for the given source operands. Examples of these operations include IEEE Standard 754-specified functions implemented in software, such as Remainder: *x* REM *y*, where *y* is 0 or *x* is infinite; conversion of a floating-point number to a decimal format whose value causes an overflow, is infinity, or is NaN; and transcendental functions, such as ln (–5) or cos–1(3).

The enabling of traps have the following results:

- *Trap Enabled: The original operand values are undisturbed.*
- *Trap Disabled: A quiet NaN is delivered to the destination register if no other software trap occurs.*

### Division-by-Zero Exception (Z)

The Division-by-Zero exception is signaled on an implemented divide operation if the divisor is zero and the dividend is a finite nonzero number. Software can simulate this exception for other operations that produce a signed infinity, such as ln(0), sec(p/2), csc(0), or 0–1.

The enabling of traps have the following results:

- *Trap Enabled: The result register is not modified, and the source registers are preserved.*
- *Trap Disabled: The result, when no trap occurs, is a correctly signed infinity.*

Notes

### Overflow Exception (O)

The Overflow exception is signaled when the magnitude of the rounded floating-point result, with an unbounded exponent range, is larger than the largest finite number of the destination format. (This exception also sets the Inexact exception and *Flag* bits.)

The enabling of traps have the following results:

- *Trap Enabled: The result register is not modified, and the source registers are preserved.*
- *Trap Disabled: The result, when no trap occurs, is determined by the rounding mode and the sign of the intermediate result.*

### Underflow Exception (U)

Two related events contribute to the Underflow exception:

- *creation of a tiny nonzero result between* $\pm 2^{Emin}$ *which can cause some later exception because it is so tiny*
- *extraordinary loss of accuracy during the approximation of such tiny numbers by denormalized numbers.*

IEEE Standard 754 allows a variety of ways to detect these events, but requires they be detected the same way for all operations. Tininess can be detected by one of the following methods:

- *after rounding (when a nonzero result, computed as though the exponent range were unbounded, would lie strictly between* $\pm 2^{Emin}$*)*
- *before rounding (when a nonzero result, computed as though the exponent range and the precision were unbounded, would lie strictly between* $\pm 2^{Emin}$*).*

The MIPS architecture requires that tininess be detected after rounding. Loss of accuracy can be detected by one of the following methods:

- *denormalization loss (when the delivered result differs from what would have been computed if the exponent range were unbounded)*
- *inexact result (when the delivered result differs from what would have been computed if the exponent range and precision were both unbounded).*

The MIPS architecture requires that loss of accuracy be detected as an inexact result.

The enabling of traps have the following results:

- *Trap Enabled: If Underflow or Inexact traps are enabled, or if the FS bit is not set, then an Unimplemented exception (E) is generated, and the result register is not modified.*
- *Trap Disabled: If Underflow and Inexact traps are not enabled and the FS bit is set, the result is determined by the rounding mode and the sign of the intermediate result (as listed in Table 6.1).*

### Unimplemented Instruction Exception (E)

Any attempt to execute an instruction with an operation code or format code that has been reserved for future definition sets the *Unimplemented* bit in the *Cause* field in the FPU *Control/Status* register and traps. The operand and destination registers remain undisturbed and the instruction is emulated in software. Any of the IEEE Standard 754 exceptions can arise from the emulated operation, and these exceptions in turn are simulated.

The Unimplemented Instruction exception can also be signaled when unusual operands or result conditions are detected that the implemented hardware cannot handle properly. These include:

- *Denormalized operand, except for Compare instruction*
- *Quiet Not a Number operand, except for Compare instruction*
- *Denormalized result or Underflow, when either Underflow or Inexact Enable bits are set or the FS bit is not set.*
- *Reserved opcodes*
- *Unimplemented formats*

Notes

- *Operations which are invalid for their format (for instance, CVT.S.S)*

Denormalized and NaN operands are only trapped if the instruction is a convert or computational operation. Moves do not trap if their operands are either denormalized or NaNs.

The use of this exception for such conditions is optional; most of these conditions are newly developed and are not expected to be widely used in early implementations. Loopholes are provided in the architecture so that these conditions can be implemented with assistance provided by software, maintaining full compatibility with the IEEE Standard 754.

The enabling of traps have the following results:

- *Trap Enabled: The original operand values are undisturbed.*
- *Trap Disabled: This trap cannot be disabled.*

## Saving and Restoring State

Sixteen or thirty-two doubleword coprocessor load or store operations save or restore the coprocessor floating-point register state in memory. The remainder of control and status information can be saved or restored through Move To/From Coprocessor Control Register instructions, and saving and restoring the processor registers. Normally, the *Control*/*Status* register is saved first and restored last.

When the coprocessor *Control/Status* register (*FCR31*) is read, and the coprocessor is executing one or more floating-point instructions, the instruction(s) in progress are either completed or reported as exceptions. The architecture requires that no more than one of these pending instructions can cause an exception. If the pending instruction cannot be completed, this instruction is placed in the *Exception* register, if present. Information indicating the type of exception is placed in the *Control/Status* register. When state is restored, state information in the status word indicates that exceptions are pending.

Writing a zero value to the *Cause* field of *Control/Status* register clears all pending exceptions, permitting normal processing to restart after the floating-point register state is restored.

The *Cause* field of the *Control/Status* register holds the results of only one instruction; the FPU examines source operands before an operation is initiated to determine if this instruction can possibly cause an exception. If an exception is possible, the FPU executes the instruction in stall mode to ensure that no more than one instruction (that might cause an exception) is executed at a time.

# Memory Management Unit

Notes

## Introduction

The processor provides a full-featured memory management unit (MMU) which uses an on-chip translation lookaside buffer (TLB) to translate virtual addresses into physical addresses.

This chapter describes the processor virtual and physical address spaces, the virtual-to-physical address translation, the operation of the TLB in making these translations, and those System Control Coprocessor (CP0) registers that provide the software interface to the TLB.

## Address Spaces

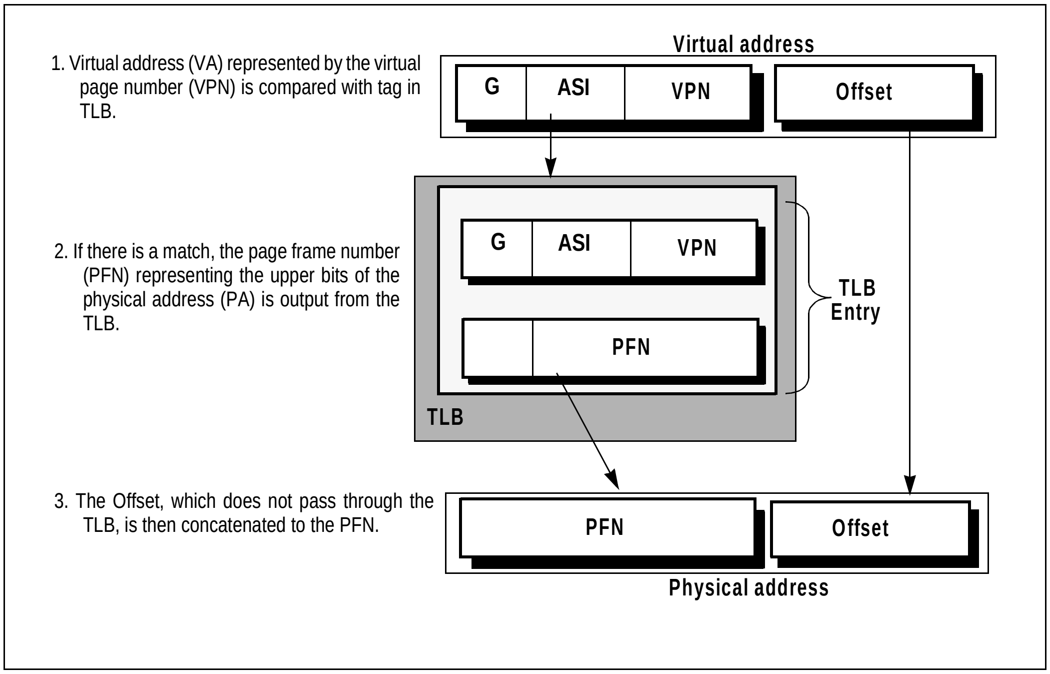
This section describes the virtual and physical address spaces and the manner in which virtual addresses are translated into physical addresses in the TLB.

### Virtual Address Space

The processor virtual address can be either 32 or 64 bits wide, depending on whether the processor is operating in 32-bit or 64-bit mode.

- *In 32-bit mode (extended address bit = 0), addresses are 32 bits wide. The maximum user process size is 2 gigabytes ($2^{31}$).*
- *In 64-bit mode (extended address bit = 1), addresses are 64 bits wide. The maximum user process size is 1 terabyte ($2^{40}$).*

Figure 7.1 shows the translation of a virtual address into a physical address.



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

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

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

**Figure 7.1 Overview of a Virtual-to-Physical Address Translation**

As shown in Figures 11 and 12, the virtual address is extended with an 8-bit address space identifier (ASID), which reduces the frequency of TLB flushing when switching contexts. This 8-bit ASID is in the CP0 *EntryHi* register. The *Global* bit (*G*) is in the *EntryLo0* and *EntryLo1* registers.

Notes

### Physical Address Space

Using a 36-bit address, the processor supports a physical address space of 64 gigabytes. The following section describes the translation of a virtual address to a physical address.

### 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 virtual page number (VPN) of the address is the same as the VPN field of the entry, and either:

- *the Global (G) bit of the TLB entry is set, or*
- *the ASID field of the virtual address is the same as the ASID field of the TLB entry.*

This match is referred to as a *TLB hit*. If there is no match, 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. If there is a virtual address match in the TLB, the physical address is output from the TLB and concatenated with the *Offset*, which represents an address within the page frame space. The *Offset* does not pass through the TLB.

Virtual-to-physical translation is described in greater detail throughout the remainder of this chapter. The next two sections describe the 32-bit and 64-bit address translations.

### 32-bit Mode Virtual Address Translation

Figure 7.2 shows the virtual-to-physical-address translation of a 32-bit mode address:

- *The top portion of Figure 7.2 shows a virtual address with a 12-bit, or 4-Kbyte, page size, labelled Offset. The remaining 20 bits of the address represent the VPN, and index the 1M-entry page table.*
- *The bottom portion of Figure 7.2 shows a virtual address with a 24-bit, or 16-Mbyte, page size, labelled Offset. The remaining 8 bits of the address represent the VPN, and index the 256-entry page table.*

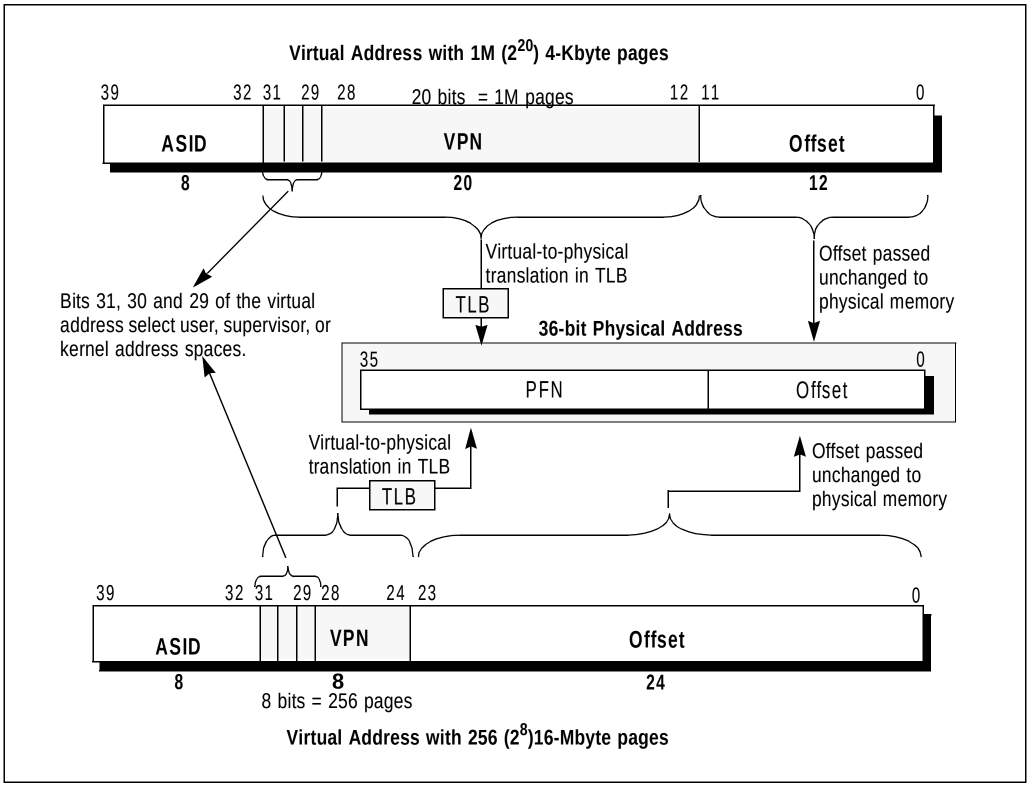

Virtual Address with 1M ($2^{20}$) 4-Kbyte pages
39
32
31
29
28
20 bits = 1M pages
12
11
0
ASID
VPN
Offset
8
20
12
Virtual-to-physical translation in TLB
Offset passed unchanged to physical memory
Bits 31, 30 and 29 of the virtual address select user, supervisor, or kernel address spaces.
TLB
36-bit Physical Address
35
0
PFN
Offset
Virtual-to-physical translation in TLB
TLB
Offset passed unchanged to physical memory
39
32
31
29
28
24
23
0
ASID
VPN
Offset
8
8
24
8 bits = 256 pages
Virtual Address with 256 ($2^8$)16-Mbyte pages

Figure 7.2 32-bit Mode Virtual Address Translation

Notes

### 64-bit Mode Virtual Address Translation

Figure 7.3 shows the virtual-to-physical-address translation of a 64-bit mode address. This figure illustrates the two extremes in the range of possible page sizes: a 4-Kbyte page (12 bits) and a 16-Mbyte page (24 bits):

- *The top portion of Figure 7.3 shows a virtual address with a 12-bit, or 4-Kbyte, page size, labelled Offset. The remaining 28 bits of the address represent the VPN, and index the 256M-entry page table.*
- *The bottom portion of Figure 7.3 shows a virtual address with a 24-bit, or 16-Mbyte, page size, labelled Offset. The remaining 16 bits of the address represent the VPN, and index the 64K-entry page table.*

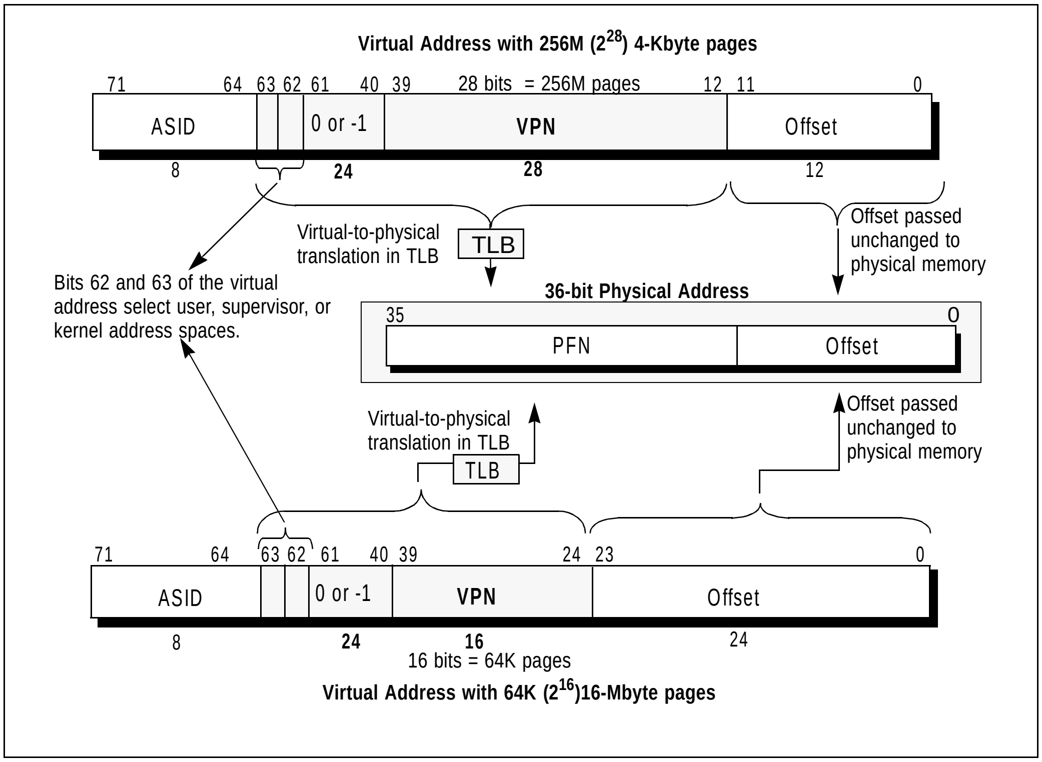

Virtual Address with 256M ($2^{28}$) 4-Kbyte pages
71
64
63
62
61
40
39
28 bits = 256M pages
12
11
0
ASID
0 or -1
VPN
Offset
8
24
28
12
Virtual-to-physical translation in TLB
TLB
Offset passed unchanged to physical memory
Bits 62 and 63 of the virtual address select user, supervisor, or kernel address spaces.
36-bit Physical Address
35
0
PFN
Offset
Virtual-to-physical translation in TLB
TLB
Offset passed unchanged to physical memory
71
64
63
62
61
40
39
24
23
0
ASID
0 or -1
VPN
Offset
8
24
16
24
16 bits = 64K pages
Virtual Address with 64K ($2^{16}$)16-Mbyte pages

Figure 7.3 64-bit Mode Virtual Address Translation

## Operating Modes

The processor has three operating modes—user, supervisor, and kernel—that function in both 32- and 64-bit operations. These modes are described in the next three sections.

### User Mode Operations

Figure 7.4 shows User mode virtual address space. In User mode, a single, uniform virtual address space—labelled User segment—is available; its size is:

- *2 Gbytes ($2^{31}$ bytes) in 32-bit mode. UX = 0 (useg)*
- *1 Tbyte ($2^{40}$ bytes) in 64-bit mode. UX = 1 (xuseg)*

Notes

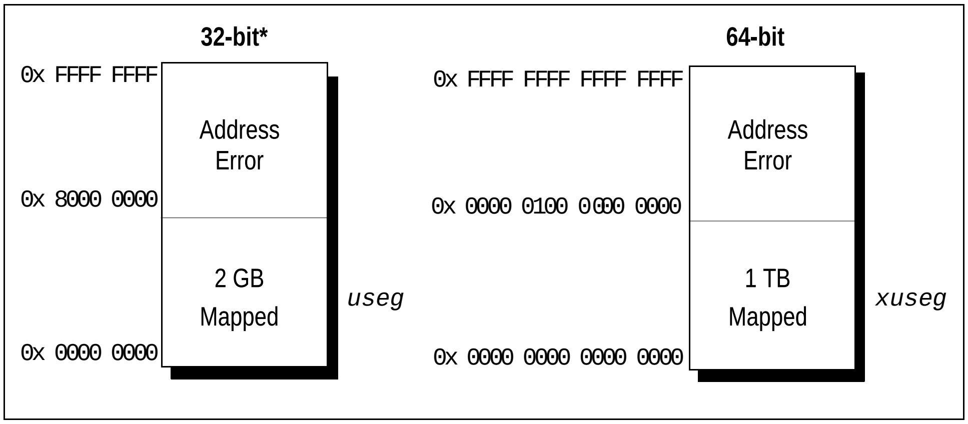


**Figure 7.4 User Mode Virtual Address Space**

The User segment starts at address 0 and the current active user process resides in either useg (in 32-bit mode) or xuseg (in 64-bit mode). The TLB identically maps all references to useg/xuseg from all modes, and controls cache accessibility.

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

- *KSU bits = $10_2$*
- *EXL = 0*
- *ERL = 0*

In conjunction with these bits, the *UX* bit in the *Status* register selects between 32- or 64-bit User mode addressing as follows:

- *when UX = 0, 32-bit useg space is selected.*
- *when UX = 1, 64-bit xuseg space is selected.*

Table 7.1 lists the characteristics of the two user mode segments, *useg* and *xuseg*.

| Address Bit Values | Status Register Bit Values | | | | Segment Name | Address Range | Segment Size |
|---|---|---|---|---|---|---|---|
| | KSU | EXL | ERL | UX | | | |
| 32-bit<br>A(31) = 0 | $10_2$ | 0 | 0 | 0 | useg | 0x0000 0000<br>through<br>0x7FFF FFFF | 2 Gbyte<br>($2^{31}$ bytes) |
| 64-bit<br>A(63:40) = 0 | $10_2$ | 0 | 0 | 1 | xuseg | 0x0000 0000 0000 0000<br>through<br>0x0000 00FF FFFF FFFF | 1 Tbyte<br>($2^{40}$ bytes) |

**Table 7.1 32-bit and 64-bit User Mode Segments**

### 32-bit User Mode (useg)

In User mode, when *UX* = 0 in the *Status* register, User mode addressing is compatible with the 32-bit addressing model shown in Figure 7.4, and a 2-Gbyte user address space is available, labelled *useg*. The system maps all references to *useg* through the TLB, and bit settings within the TLB entry for the page determine the cacheability of a reference. All valid User mode virtual addresses have their most-significant bit cleared to 0; any attempt to reference an address with the most-significant bit set while in User mode causes an Address Error exception.

Notes

### 64-bit User Mode (*xuseg*)

In User mode, when *UX* = 1 in the *Status* register, User mode addressing is extended to 64-bits. In 64-bit User mode, the processor provides a single, uniform address space of $2^{40}$ bytes, labelled *xuseg*. All valid User mode virtual addresses have bits 63:40 equal to 0; an attempt to reference an address with bits 63:40 not equal to 0 causes an Address Error exception.

## Supervisor Mode Operations

Supervisor mode is designed for layered operating systems in which a true kernel runs in Kernel mode, and the rest of the operating system runs in Supervisor mode. The processor operates in Supervisor mode when the *Status* register contains the following bit-values:

- *KSU* = $01_2$
- *EXL = 0*
- *ERL = 0*

In conjunction with these bits, the *SX* bit in the *Status* register selects between 32- or 64-bit Supervisor mode addressing:

- *when SX = 0, 32-bit supervisor space is selected and TLB misses are handled by the 32-bit TLB refill exception handler*
- *when SX = 1, 64-bit supervisor space is selected and TLB misses are handled by the 64-bit XTLB refill exception handler. Figure 7.5 shows Supervisor mode address mapping. Table 7.2 lists the characteristics of the supervisor mode segments; descriptions of the address spaces follow.*

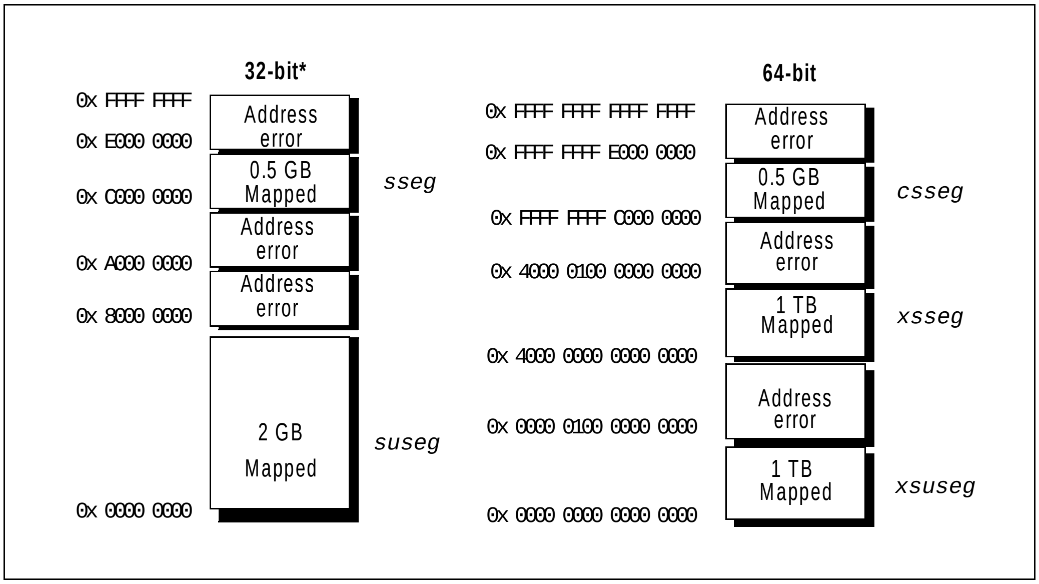


**Figure 7.5 Supervisor Mode Address Space**

| Address Bit Values | Status Register Bit Values | | | | Segment Name | Address Range | Segment Size |
|---|---|---|---|---|---|---|---|
| | KSU | EXL | ERL | SX | | | |
| 32-bit<br>A(31) = 0 | $01_2$ | 0 | 0 | 0 | suseg | 0x0000 0000<br>through<br>0x7FFF FFFF | 2 Gbytes<br>$2^{31}$ bytes) |
| 32-bit<br>A(31:29) = $110_2$ | $01_2$ | 0 | 0 | 0 | ssseg | 0xC000 0000<br>through<br>0xDFFF FFFF | 512 Mbytes<br>($2^{29}$ bytes) |
| 64-bit<br>A(63:62) = $00_2$ | $01_2$ | 0 | 0 | 1 | xsuseg | 0x0000 0000 0000 0000<br>through<br>0x0000 00FF FFFF FFFF | 1 Tbyte<br>($2^{40}$ bytes) |
| 64-bit<br>A(63:62) = $01_2$ | $01_2$ | 0 | 0 | 1 | xsseg | 0x4000 0000 0000 0000<br>through<br>0x4000 00FF FFFF FFFF | 1 Tbyte<br>($2^{40}$ bytes) |
| 64-bit<br>A(63:62) = $11_2$ | $01_2$ | 0 | 0 | 1 | csseg | 0xFFFF FFFF C000 0000<br>through<br>0xFFFF FFFF DFFF FFFF | 512 Mbytes<br>($2^{29}$ bytes) |

Table 7.2 32-bit and 64-bit Supervisor Mode Segments

### 32-bit Supervisor Mode, User Space (*suseg*)

In Supervisor mode, when *SX* = 0 in the *Status* register and the most-significant bit of the 32-bit virtual address is set to 0, the *suseg* virtual address space is selected; it covers the full $2^{31}$ bytes (2 Gbytes) of the current user address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

This mapped space starts at virtual address 0x0000 0000 and runs through 0x7FFF FFFF.

### 32-bit Supervisor Mode, Supervisor Space (*sseg*)

In Supervisor mode, when *SX* = 0 in the *Status* register and the three most-significant bits of the 32-bit virtual address are $110_2$, the *sseg* virtual address space is selected; it covers $2^{29}$-bytes (512 Mbytes) of the current supervisor address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

This mapped space begins at virtual address 0xC000 0000 and runs through 0xDFFF FFFF.

### 64-bit Supervisor Mode, User Space (*xsuseg*)

In Supervisor mode, when *SX* = 1 in the *Status* register and bits 63:62 of the virtual address are set to $00_2$, the *xsuseg* virtual address space is selected; it covers the full $2^{40}$ bytes (1 Tbyte) of the current user address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

This mapped space starts at virtual address 0x0000 0000 0000 0000 and runs through 0x0000 00FF FFFF FFFF.

### 64-bit Supervisor Mode, Current Supervisor Space (*xsseg*)

In Supervisor mode, when *SX* = 1 in the *Status* register and bits 63:62 of the virtual address are set to $01_2$, the *xsseg* current supervisor virtual address space is selected. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

This mapped space begins at virtual address 0x4000 0000 0000 0000 and runs through 0x4000 00FF FFFF FFFF.

### 64-bit Supervisor Mode, Separate Supervisor Space (*csseg*)

In Supervisor mode, when *SX* = 1 in the *Status* register and bits 63:62 of the virtual address are set to $11_2$, the *csseg* separate supervisor virtual address space is selected. Addressing of the *csseg* is compatible with addressing *sseg* in 32-bit mode. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

This mapped space begins at virtual address 0xFFFF FFFF C000 0000 and runs through 0xFFFF FFFF DFFF FFFF.

### Kernel Mode Operations

The processor operates in Kernel mode when the *Status* register contains one of the following values:

- *KSU* = $00_2$
- *EXL = 1*
- *ERL = 1*

In conjunction with these bits, the *KX* bit in the *Status* register selects between 32- or 64-bit Kernel mode addressing:

- *when KX = 0, 32-bit kernel space is selected.*
- *when KX = 1, 64-bit kernel space is selected.*

The processor enters Kernel mode whenever an exception is detected and it remains in Kernel mode until an Exception Return (ERET) instruction is executed. The ERET instruction restores the processor to the mode existing prior to the exception.

Kernel mode virtual address space is divided into regions differentiated by the high-order bits of the virtual address, as shown in Figure 7.6. Table 7.3 lists the characteristics of the 32-bit kernel mode segments, and Table 7.4 lists the characteristics of the 64-bit kernel mode segments.

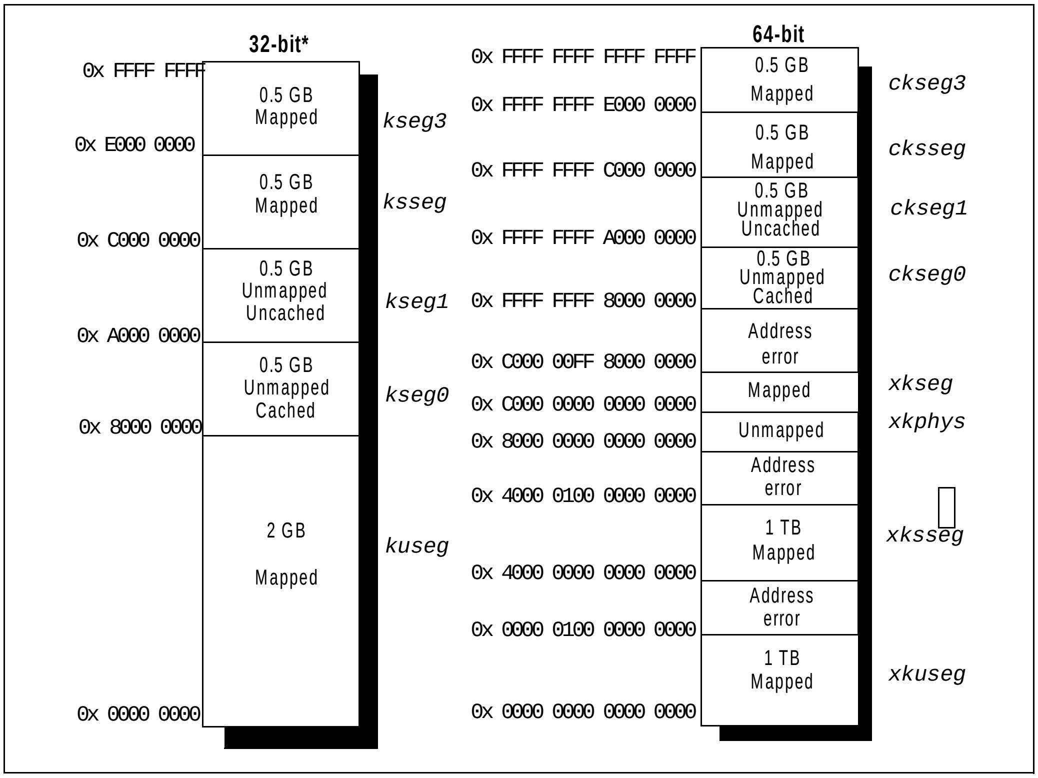


**Figure 7.6 Kernel Mode Address Space**

**Notes**

| Address Bit Values | Status Register Is One Of These Values | | | | Segment Name | Address Range | Segment Size |
|---|---|---|---|---|---|---|---|
| | KSU | EXL | ERL | KX | | | |
| A(31) = 0 | KSU = $00_2$<br>or<br>EXL = 1<br>or<br>ERL =1 | | | 0 | kuseg | 0x0000 0000<br>through<br>0x7FFF FFFF | 2 Gbytes<br>($2^{31}$ bytes) |
| A(31:29) = $100_2$ | | | | 0 | kseg0 | 0x8000 0000<br>through<br>0x9FFF FFFF | 512 Mbytes<br>($2^{29}$ bytes) |
| A(31:29) = $101_2$ | | | | 0 | kseg1 | 0xA000 0000<br>through<br>0xBFFF FFFF | 512 Mbytes<br>($2^{29}$ bytes) |
| A(31:29) = $110_2$ | | | | 0 | ksseg | 0xC000 0000<br>through<br>0xDFFF FFFF | 512 Mbytes<br>($2^{29}$ bytes) |
| A(31:29) = $111_2$ | | | | 0 | kseg3 | 0xE000 0000<br>through<br>0xFFFF FFFF | 512 Mbytes<br>($2^{29}$ bytes) |

**Table 7.3 32-Bit Kernel Mode Segments**

### 32-bit Kernel Mode, User Space (*kuseg*)

In Kernel mode, when *KX* = 0 in the *Status* register, and the most-significant bit of the virtual address, A31, is cleared, the 32-bit *kuseg* virtual address space is selected; it covers the full $2^{31}$ bytes (2 Gbytes) of the current user address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

### 32-bit Kernel Mode, Kernel Space 0 (*kseg0*)

In Kernel mode, when *KX* = 0 in the *Status* register and the most-significant three bits of the virtual address are $100_2$, 32-bit *kseg0* virtual address space is selected; it is the $2^{29}$-byte (512-Mbyte) kernel physical space. References to *kseg0* are not mapped through the TLB; the physical address selected is defined by subtracting 0x8000 0000 from the virtual address. The *K0* field of the *Config* register, described in this chapter, controls cacheability and coherency.

### 32-bit Kernel Mode, Kernel Space 1 (*kseg1*)

In Kernel mode, when *KX* = 0 in the *Status* register and the most-significant three bits of the 32-bit virtual address are $101_2$, 32-bit *kseg1* virtual address space is selected; it is the $2^{29}$-byte (512-Mbyte) kernel physical space. References to *kseg1* are not mapped through the TLB; the physical address selected is defined by subtracting 0xA000 0000 from the virtual address. Caches are disabled for accesses to these addresses, and physical memory (or memory-mapped I/O device registers) are accessed directly.

### 32-bit Kernel Mode, Supervisor Space (*ksseg*)

In Kernel mode, when *KX* = 0 in the *Status* register and the most-significant three bits of the 32-bit virtual address are $110_2$, the *ksseg* virtual address space is selected; it is the current $2^{29}$-byte (512-Mbyte) supervisor virtual space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

### 32-bit Kernel Mode, Kernel Space 3 (*kseg3*)

In Kernel mode, when *KX* = 0 in the *Status* register and the most-significant three bits of the 32-bit virtual address are $111_2$, the *kseg3* virtual address space is selected; it is the current $2^{29}$-byte (512-Mbyte) kernel virtual space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

## Notes

| Address Bit Values | Status Register Is One Of These Values | | | | Segment Name | Address Range | Segment Size |
|---|---|---|---|---|---|---|---|
| | KSU | EXL | ERL | KX | | | |
| A(63:62) = $00_2$ | KSU = $00_2$<br>or<br>EXL = 1<br>or<br>ERL =1 | | | 1 | xksuseg | 0x0000 0000 0000 0000<br>through<br>0x0000 00FF FFFF FFFF | 1 Tbyte<br>($2^{40}$ bytes) |
| A(63:62) = $01_2$ | | | | 1 | xksseg | 0x4000 0000 0000 0000<br>through<br>0x4000 00FF FFFF FFFF | 1 Tbyte<br>($2^{40}$ bytes) |
| A(63:62) = $10_2$ | | | | 1 | xkphys | 0x8000 0000 0000 0000<br>through<br>0xBFFF FFFF FFFF FFFF | 8 $2^{36}$-byte spaces |
| A(63:62) = $11_2$ | | | | 1 | xkseg | 0xC000 0000 0000 0000<br>through<br>0xC000 00FF 7FFF FFFF | ($2^{40}$–$2^{31}$) bytes |
| A(63:62) = $11_2$<br>A(61:31) = -1 | | | | 1 | ckseg0 | 0xFFFF FFFF 8000 0000<br>through<br>0xFFFF FFFF 9FFF FFFF | 512 Mbytes<br>($2^{29}$ bytes) |
| A(63:62) = $11_2$<br>A(61:31) = -1 | | | | 1 | ckseg1 | 0xFFFF FFFF A000 0000<br>through<br>0xFFFF FFFF BFFF FFFF | 512 Mbytes<br>($2^{29}$ bytes) |
| A(63:62) = $11_2$<br>A(61:31) = -1 | | | | 1 | cksseg | 0xFFFF FFFF C000 0000<br>through<br>0xFFFF FFFF DFFF FFFF | 512 Mbytes<br>($2^{29}$ bytes) |
| A(63:62) = $11_2$<br>A(61:31) = -1 | | | | 1 | ckseg3 | 0xFFFF FFFF E000 0000<br>through<br>0xFFFF FFFF FFFF FFFF | 512 Mbytes<br>($2^{29}$ bytes) |

**Table 7.4 64-Bit Kernel Mode Segments**

### 64-bit Kernel Mode, User Space (*xkuseg*)

In Kernel mode, when *KX* = 1 in the *Status* register and bits 63:62 of the 64-bit virtual address are $00_2$, the *xkuseg* virtual address space is selected; it covers the current user address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address. When ERL = 1 in the *Status* register, the user address region becomes a $2^{31}$-byte unmapped (that is, mapped directly to physical addresses) uncached address space.

### 64-bit Kernel Mode, Current Supervisor Space (*xksseg*)

In Kernel mode, when *KX* = 1 in the *Status* register and bits 63:62 of the 64-bit virtual address are $01_2$, the *xksseg* virtual address space is selected; it is the current supervisor virtual space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

Notes

### 64-bit Kernel Mode, Physical Spaces (*xkphys*)

In Kernel mode, when *KX* = 1 in the *Status* register and bits 63:62 of the 64-bit virtual address are $10_2$, the *xkphys* virtual address space is selected; it is a set of eight $2^{36}$-byte kernel physical spaces. Accesses with address bits 58:36 not equal to 0 cause an address error. References to this space are not mapped; the physical address selected is taken from bits 35:0 of the virtual address. Bits 61:59 of the virtual address specify the cacheability and coherency attributes, as shown in Table 7.5.

| Value (61:59) | Cacheability and Coherency Attributes | Starting Address |
|---|---|---|
| 0 | Cacheable, noncoherent, write-through, no write allocate | 0x8000 0000 0000 0000 |
| 1 | Cacheable, noncoherent, write-through, write allocate | 0x8800 0000 0000 0000 |
| 2 | Uncached | 0x9000 0000 0000 0000 |
| 3 | Cacheable, noncoherent | 0x9800 0000 0000 0000 |
| 4-7 | Reserved | 0xA000 0000 0000 0000 |

**Table 7.5 Cacheability and Coherency Attributes**

### 64-bit Kernel Mode, Kernel Space (*xkseg*)

In Kernel mode, when *KX* = 1 in the *Status* register and bits 63:62 of the 64-bit virtual address are $11_2$, the address space selected is one of the following:

- *kernel virtual space, xkseg, the current kernel virtual space; the virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address*
- *one of the four 32-bit kernel compatibility spaces, as described in the next section.*

### 64-bit Kernel Mode, Compatibility Spaces

In Kernel mode, when *KX* = 1 in the *Status* register, bits 63:62 of the 64-bit virtual address are $11_2$, and bits 61:31 of the virtual address equal –1. The lower two bytes of address, as shown in figure 15, select one of the following 512-Mbyte compatibility spaces.

- *ckseg0. This 64-bit virtual address space is an unmapped region, compatible with the 32-bit address model kseg0. The K0 field of the Config register controls cacheability and coherency.*
- *ckseg1. This 64-bit virtual address space is an unmapped and uncached region, compatible with the 32-bit address model kseg1.*
- *cksseg. This 64-bit virtual address space is the current supervisor virtual space, compatible with the 32-bit address model ksseg.*
- *ckseg3. This 64-bit virtual address space is kernel virtual space, compatible with the 32-bit address model kseg3.*

## System Control Coprocessor

The System Control Coprocessor (CP0) is implemented as an integral part of the CPU, and supports memory management, address translation, exception handling, and other privileged operations. CP0 contains the registers shown in Figure 7.7 plus a 48-entry TLB. The sections that follow describe how the processor uses the memory management-related registers.

Each CP0 register has a unique number that identifies it; this number is referred to as the *register number*. For instance, the *Page Mask* register is register number 5.

Notes

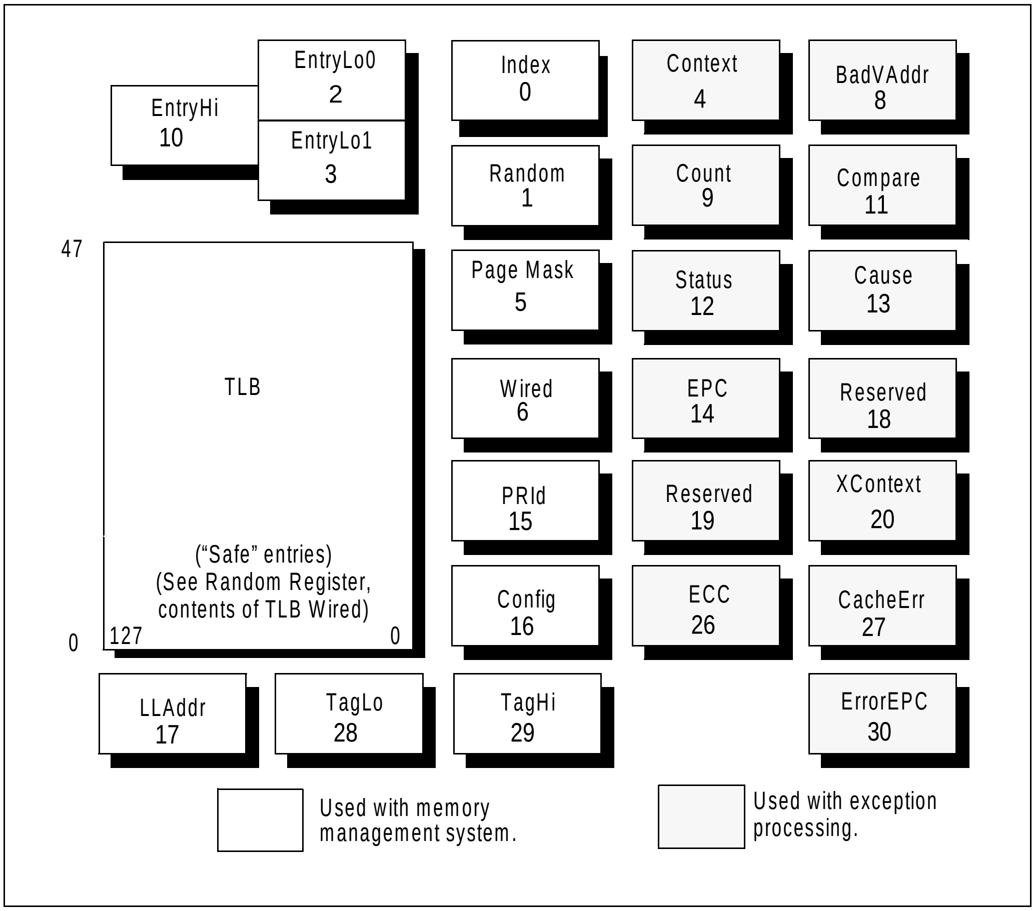


Figure 7.7 CP0 Registers and the TLB

## Translation Lookaside Buffer (TLB)

Mapped virtual addresses are translated into physical addresses using an on-chip TLB.[1] The TLB is a fully associative memory that holds 48 entries, which provide mapping to 48 odd/even page pairs (96 pages). When address mapping is indicated, each TLB entry is checked simultaneously for a match with the virtual address that is extended with an ASID stored in the *EntryHi* register. The page size can be configured, on a per-entry basis, at 4Kbytes, 16Kbytes, 64Kbytes, 256Kbytes, 1Mbytes, 4Mbytes, or 16Mbytes

## Format of a TLB Entry

Figure 7.8 shows the TLB entry formats for both 32- and 64-bit modes. Each field of an entry has a corresponding field in the *EntryHi*, *EntryLo0*, *EntryLo1*, or *PageMask* registers. Figure 7.9 and Figure 7.10 show the *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers. The formats of these registers are nearly the same as the TLB-entry formats. The one exception is the *Global* field (*G* bit), which is used in the TLB, but is reserved in the *EntryHi* register.

[1] There are virtual-to-physical address translations that occur outside of the TLB. For example, addresses in *the kseg0* and *kseg1* spaces are unmapped translations. In these spaces the physical address is 0x000_0000_0 || VA[28:0].

## Notes

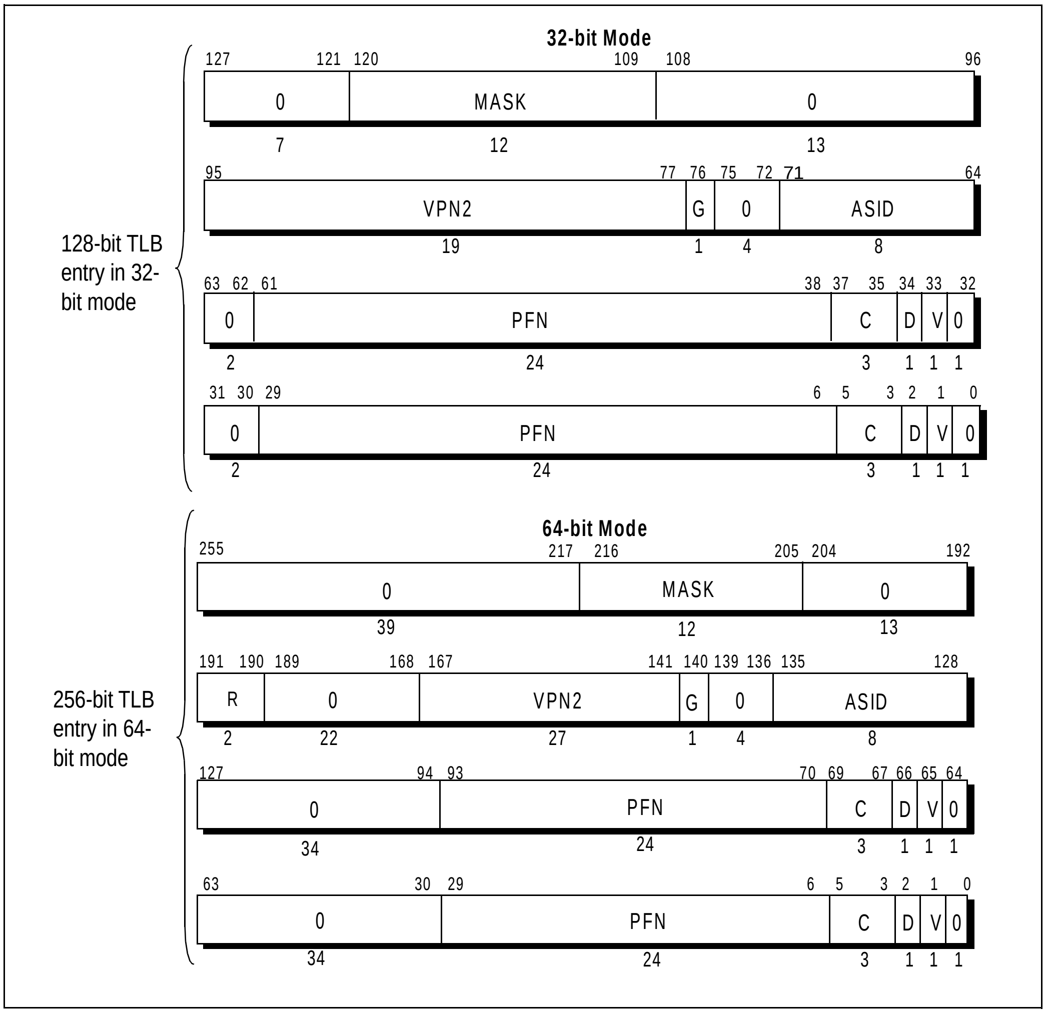
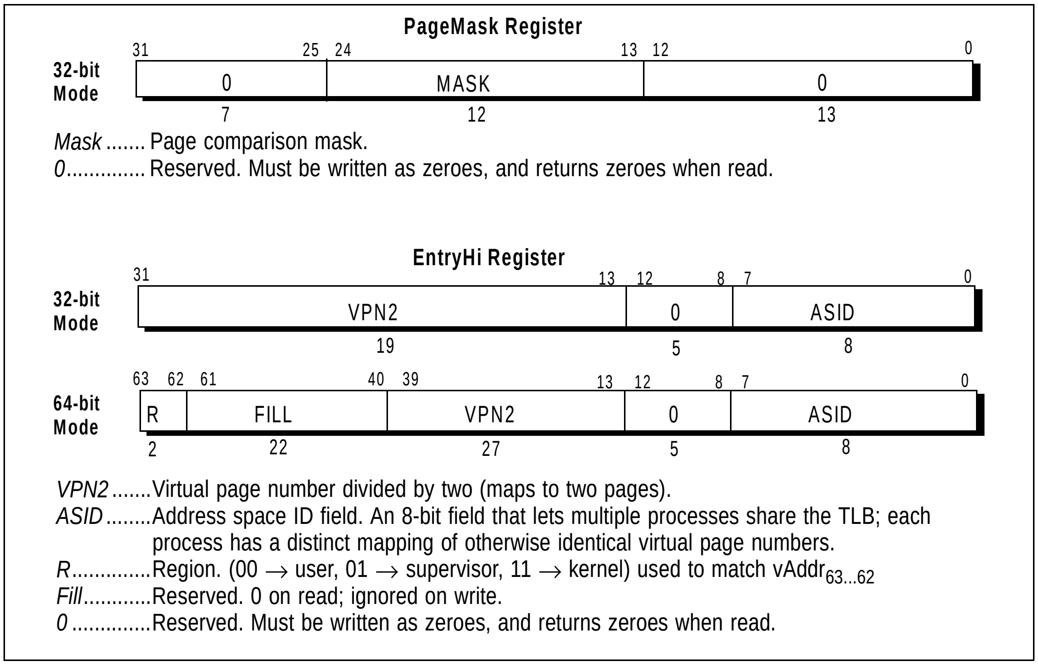


**Figure 7.8 Format of a TLB Entry**

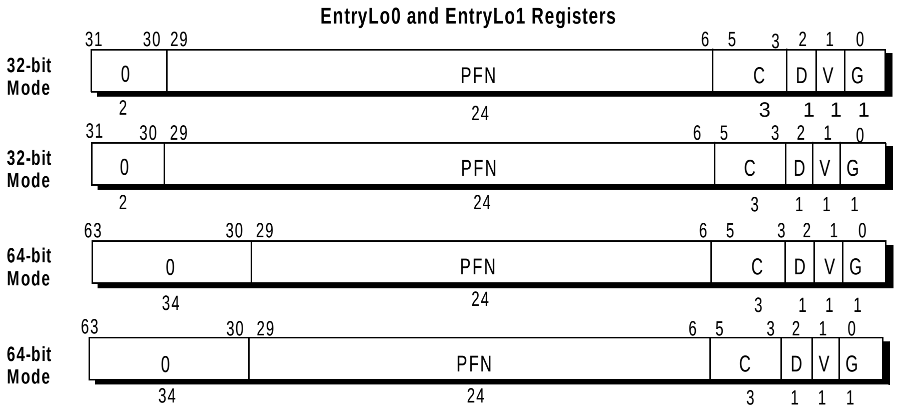


*Mask* ....... Page comparison mask.
*0*.............. Reserved. Must be written as zeroes, and returns zeroes when read.

*VPN2*.......Virtual page number divided by two (maps to two pages).
*ASID*........Address space ID field. An 8-bit field that lets multiple processes share the TLB; each process has a distinct mapping of otherwise identical virtual page numbers.
*R*..............Region. (00 → user, 01 → supervisor, 11 → kernel) used to match $vAddr_{63...62}$
*Fill*............Reserved. 0 on read; ignored on write.
*0* ..............Reserved. Must be written as zeroes, and returns zeroes when read.

**Figure 7.9 Fields of the PageMask and EntryHi Registers**

Notes



*PFN*......... Page frame number; the upper bits of the physical address.
*C* ............ Specifies the TLB page coherency attribute; see Table 7.6.
*D* ............ Dirty. If this bit is set, the page is marked as dirty and, therefore, writable. This bit is actually a write-protect bit that software can use to prevent alteration of data.
*V* ............ Valid. If this bit is set, it indicates that the TLB entry is valid; otherwise, a TLBL or TLBS miss occurs.
*G* ............ Global. If this bit is set in both Lo0 and Lo1, then the processor ignores the ASID during TLB lookup.
*0*............. Reserved. Must be written as zeroes, and returns zeroes when read.

**Figure 7.10 Fields of the EntryLo0 and EntryLo1 Registers**

The TLB page coherency attribute (*C*) bits specify whether references to the page should be cached; if cached, the algorithm selects between several coherency attributes. Table 7.6 shows the coherency attributes selected by the *C* bits.

| *C*(5:3) Value | Page Coherency Attribute |
|---|---|
| 0 | Cacheable, noncoherent, write-through, no write allocate |
| 1 | Cacheable, noncoherent, write-through, write allocate |
| 2 | Uncached |
| 3 | Cacheable, noncoherent, write-back |
| 4 - 7 | Reserved |

**Table 7.6 TLB Page Coherency (C) Bit Values**

## CP0 Registers

The following sections describe the CP0 registers that are assigned specifically as a software interface with memory management (each register is followed by its register number in parentheses).

- *Index register (CP0 register number 0)*
- *Random register (1)*
- *EntryLo0 (2) and EntryLo1 (3) registers*
- *PageMask register (5)*
- *Wired register (6)*
- *EntryHi register (10)*
- *PRId register (15)*
- *Config register (16)*
- *LLAddr register (17)*
- *TagLo (28) and TagHi (29) registers*

### Index Register (0)

The *Index* register is a 32-bit, read/write register containing six bits to index an entry in the TLB. The high-order bit of the register shows the success or failure of a TLB Probe (TLBP) instruction. The *Index* register also specifies the TLB entry affected by TLB Read (TLBR) or TLB Write Index (TLBWI) instructions.

Figure 7.11 shows the format of the *Index* register; Table 7.7 describes the *Index* register fields.

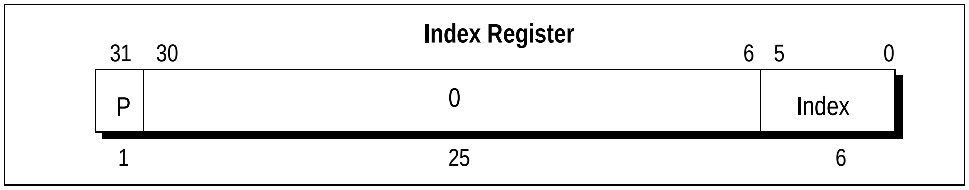


**Figure 7.11 Index Register**

| Field | Description |
|---|---|
| P | Probe failure. Set to 1 when the previous TLBProbe (TLBP) instruction was unsuccessful. |
| Index | Index to the TLB entry affected by the TLBRead and TLBWrite instructions |
| 0 | Reserved. Must be written as zeroes, and returns zeroes when read. |

**Table 7.7 Index Register Field Descriptions**

### Random Register (1)

The *Random* register is a read-only register of which six bits index an entry in the TLB. This register decrements as each instruction executes, and its values range between an upper and a lower bound, as follows:

- *A lower bound is set by the number of TLB entries reserved for exclusive use by the operating system (the contents of the Wired register).*
- *An upper bound is set at one less than the total number of TLB entries (47 maximum).*

The *Random* register specifies the entry in the TLB that is affected by the TLB Write Random instruction. The register does not need to be read for this purpose; however, the register is readable to verify proper operation of the processor. To simplify testing, the *Random* register is set to the value of the upper bound upon system reset. This register is also set to the upper bound when the *Wired* register is written.

Figure 7.12 shows the format of the *Random* register. Table 7.8 describes the *Random* register fields.

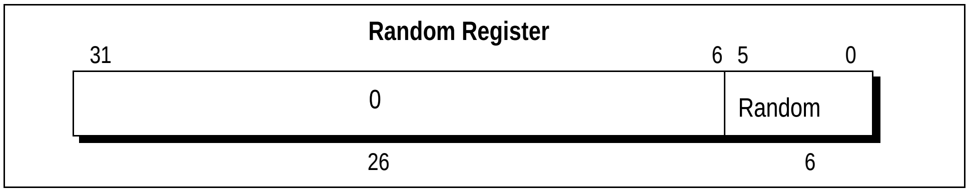


**Figure 7.12 Random Register**

| Field | Description |
|---|---|
| Random | TLB Random index |
| 0 | Reserved. Must be written as zeroes, and returns zeroes when read. |

**Table 7.8 Random Register Field Descriptions**

### EntryLo0 (2), and EntryLo1 (3) Registers

The *EntryLo* register consists of two registers that have identical formats:

- *EntryLo0 is used for even virtual pages.*

Notes

- *EntryLo1 is used for odd virtual pages.*

The *EntryLo0* and *EntryLo1* registers are read/write registers. They hold the physical page frame number (PFN) of the TLB entry for even and odd pages, respectively, when performing TLB read and write operations. Figure 7.10 shows the format of these registers.

### PageMask Register (5)

The *PageMask* register is a read/write register used for reading from or writing to the TLB; it holds a comparison mask that sets the variable page size for each TLB entry. TLB read and write operations use this register as either a source or a destination; when virtual addresses are presented for translation into physical address, the corresponding bits in the TLB identify which virtual address bits among bits 24:13 are used in the comparison. When the *Mask* field is not one of the values shown in Table 7.9, the operation of the TLB is undefined.

| Page Size | Bit | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 |
| 4 Kbytes | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 16 Kbytes | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 64 Kbytes | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 256 Kbytes | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 Mbyte | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 4 Mbytes | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 16 Mbytes | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**Table 7.9 Mask Field Values for Page Sizes**

### Wired Register (6)

The *Wired* register is a read/write register that specifies the boundary between the *wired* and *random* entries of the TLB as shown in Figure 7.13. Wired entries are fixed, nonreplaceable entries, which cannot be overwritten by a TLB write operation. Random entries can be overwritten.

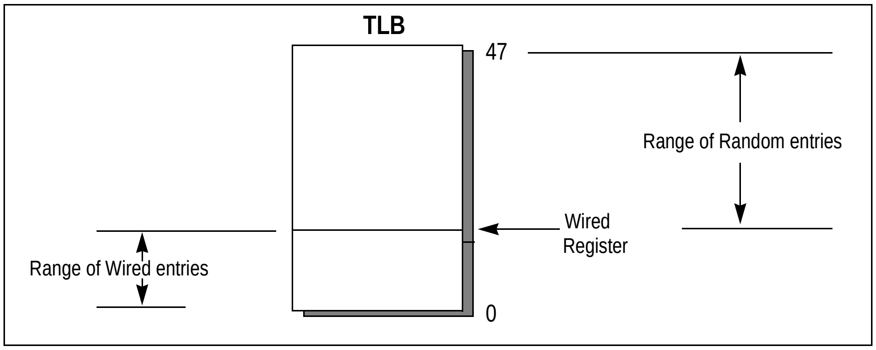


**Figure 7.13 Wired Register Boundary**

The *Wired* register is set to 0 upon system reset. Writing this register also sets the *Random* register to the value of its upper bound (see *Random* register, above). Figure 7.14 shows the format of the *Wired* register; Table 7.9 describes the register fields.

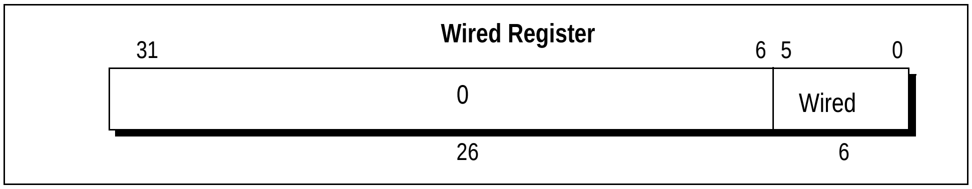


**Figure 7.14 Wired Register**

| Field | Description |
|---|---|
| Wired | TLB Wired boundary |
| 0 | Reserved. Must be written as zeroes, and returns zeroes when read. |

Table 7.10 Wired Register Field Descriptions

### EntryHi Register (CP0 Register 10)

The *EntryHi* register holds the high-order bits of a TLB entry for TLB read and write operations. The *EntryHi* register is accessed by the TLB Probe, TLB Write Random, TLB Write Indexed, and TLB Read Indexed instructions. When either a TLB refill, TLB invalid, or TLB modified exception occurs, the *EntryHi* register is loaded with the virtual page number (VPN2) and the ASID of the virtual address that did not have a matching TLB entry.

### Processor Revision Identifier (PRId) Register (15)

The 32-bit, read-only *Processor Revision Identifier* (*PRId*) register contains information identifying the implementation and revision level of the CPU and CP0. Figure 7.15 shows the format of the *PRId* register; Table 7.11 describes the *PRId* register fields.

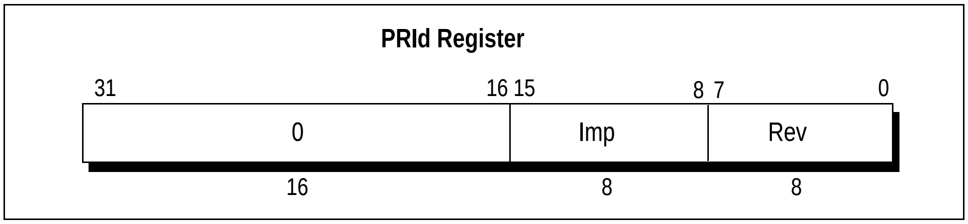


Figure 7.15 Processor Revision Identifier Register Format

| Field | Description |
|---|---|
| Imp | Implementation number<br>Imp=0x23 |
| Rev | Revision number |
| 0 | Reserved. Must be written as zeroes, and returns zeroes when read. |

Table 7.11 PRId Register Fields

The low-order byte (bits 7:0) of the *PRId* register is interpreted as a revision number, and the high-order byte (bits 15:8) is interpreted as an implementation number. The implementation number of the R5000 processor is 0x23. The content of the high-order halfword (bits 31:16) of the register are reserved.

The revision number is stored as a value in the form *y.x*, where *y* is a major revision number in bits 7:4 and *x* is a minor revision number in bits 3:0. The revision number can distinguish some chip revisions, however there is no guarantee that changes to the chip will necessarily be reflected in the *PRId* register, or that changes to the revision number necessarily reflect real chip changes. For this reason, these values are not listed and software should not rely on the revision number in the *PRId* register to characterize the chip.

### Config Register (16)

The *Config* register specifies various configuration options. Some configuration options, as defined by *Config* bits 31:13 and 11:3, are set by the hardware during Reset and are included in the *Config* register as read-only status bits. Other configuration options are read/write (as indicated by *Config* register bits 12 and 2:0) and controlled by software; on Reset, these fields are undefined. Certain configurations have restrictions. The *Config* register should be initialized by software before caches are used. Caches should be reinitialized after any change is made.

Notes

Figure 7.16 shows the format of the *Config* register; Table 7.12 describes the *Config* register fields. Refer to Chapter 8 for more information on the SE, SC and SS fields.

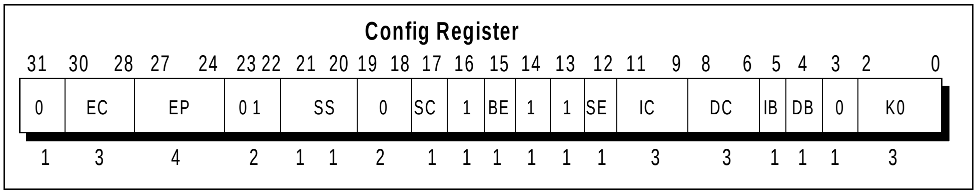


**Figure 7.16 Config Register Format**

| Field | Description |
|---|---|
| EC | System clock ratio:<br>0 → processor clock frequency divided by 2<br>1 → processor clock frequency divided by 3<br>2 → processor clock frequency divided by 4<br>3 → processor clock frequency divided by 5<br>4 → processor clock frequency divided by 6<br>5 → processor clock frequency divided by 7<br>6 → processor clock frequency divided by 8<br>7 → Reserved |
| EP | Transmit data pattern (pattern for write-back data):<br>0 → DDoubleword every cycle<br>1 → DDxDDx2 Doublewords every 3 cycles<br>2 → DDxxDDxx2 Doublewords every 4 cycles<br>3 → DxDxDxDx2 Doublewords every 4 cycles<br>4 → DDxxxDDxxx2 Doublewords every 5 cycles<br>5 → DDxxxxDDxxxx2 Doublewords every 6 cycles<br>6 → DxxDxxDxxDxx2 Doublewords every 6 cycles<br>7 → DDxxxxxxDDxxxxxx2 Doublewords every 8 cycles<br>8 → DxxxDxxxDxxxDxxx2 Doublewords every 8 cycles |
| SS | Secondary Cache Size<br>00 → 512 KByte<br>01 → 1 MByte<br>10 → 2 MByte<br>11 → None |
| BE | Big Endian Mode:<br>0 → Little Endian<br>1 → Big Endian |
| SE | Secondary Cache Enable<br>0 → Disabled<br>1 → Enabled |
| SC | Secondary Cache Present<br>0 → Present<br>1 → Not Present |
| IC | Primary I-cache Size (I-cache size = $2^{12+IC}$ bytes). In the R5000 processor, this is set to 32 Kbytes (IC=3). |
| DC | Primary D-cache Size (D-cache size = $2^{12+DC}$ bytes). In the R5000 processor, this is set to 32 Kbytes (DC=3). |
| IB | Primary I-cache line size. In the R5000 processor, this is set to 32 bytes (IB=1).<br>0 → 16 bytes<br>1 → 32 bytes |

**Table 7.12 Config Register Fields (Part 1 of 2)**

Notes

| Field | Description |
|---|---|
| DB | Primary D-cache line size. In the R5000 processor, this is set to 32 bytes (DB=1).<br>0 → 16 bytes<br>1 → 32 bytes |
| K0 | *kseg0* coherency algorithm (see *EntryLo0* and *EntryLo1* registers and the *C* field of Table 7.6) |

Table 7.12 Config Register Fields (Part 2 of 2)

### Load Linked Address (LLAddr) Register (17)

The read/write *Load Linked Address* (*LLAddr*) register contains the physical address read by the most recent Load Linked instruction. This register is for diagnostic purposes only, and serves no function during normal operation. Figure 7.17 shows the format of the *LLAddr* register; *PAddr* represents bits of the physical address, PA(35:4).

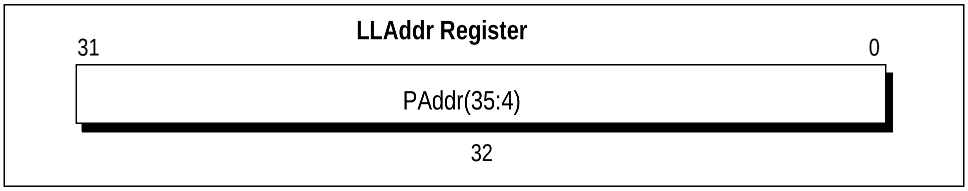


Figure 7.17 LLAddr Register Format

### Cache Tag Registers [TagLo (28) and TagHi (29)]

The *TagLo* and *TagHi* registers are 32-bit read/write registers that hold either the primary cache tag and parity, or the secondary cache tag and ECC during cache initialization, cache diagnostics, or cache error processing. The *Tag* registers are written by the CACHE and MTC0 instructions.

The *P* and *ECC* fields of these registers are ignored on Index Store Tag operations. Parity and ECC are computed by the store operation. Avoid using Instruction Index Store Tag operations except during primary-cache initialization, because the IType field determines the instruction type and problems occur if this specified incorrectly.

Figure 7.18 shows the format of these registers for primary cache operations. Figure 7.19 shows the format of these registers for secondary cache operations. Table 7.13 lists the field definitions of these registers.

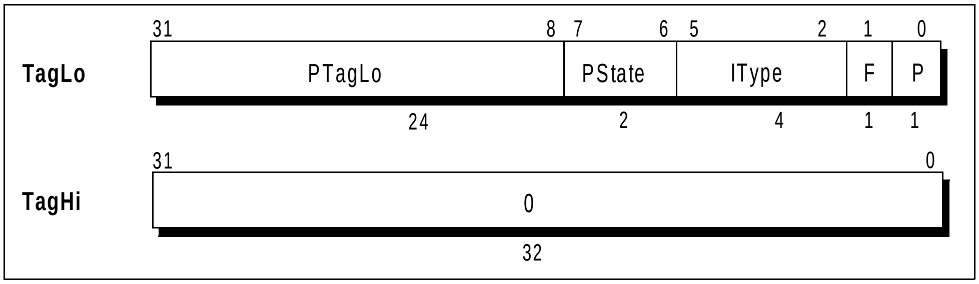


Figure 7.18 TagLo and TagHi Register (P-cache) Formats

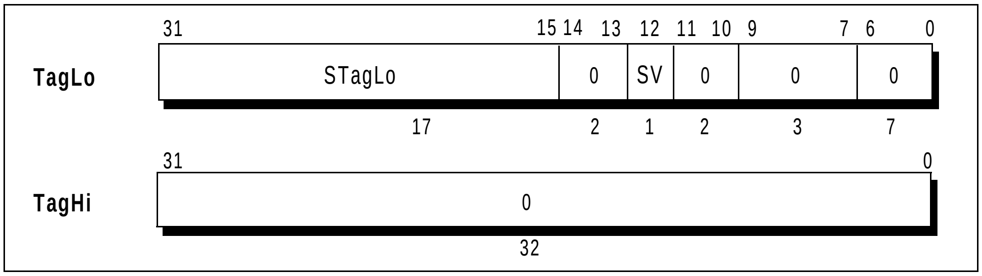


Figure 7.19 TagLo and TagHi Register (S-cache) Formats

Notes

| Field | Description |
|---|---|
| PTagLo | Specifies the physical address bits 35:12. |
| PState | Specifies the primary cache state. |
| IType | Instruction-type bits: (28) MS instruction, (25) LS instruction.<br>Specifies the even-word instruction type as integer or floating-point. |
| F | The FIFO bit, used to implement FIFO refill of the cache. |
| P | Specifies the primary tag even parity bit. |
| STagLo | Specifies the physical address bits 35:19. |
| SV | Specifies the Valid bit for secondary cache. |
| 0 | Reserved. Must be written as zeroes, and returns zeroes when read. |

**Table 7.13 Cache Tag Register Fields**

## Virtual-to-Physical Address Translation Process

During virtual-to-physical address translation, the CPU compares the 8-bit ASID (if the Global bit, *G*, is not set) of the virtual address to the ASID of the TLB entry to see if there is a match. One of the following comparisons are also made:

- *In 32-bit mode, the highest 7-to-19 bits (depending upon the page size) of the virtual address are compared to the contents of the TLB virtual page number.*
- *In 64-bit mode, the highest 15-to-27 bits (depending upon the page size) of the virtual address are compared to the contents of the TLB virtual page number.*

If a TLB entry matches, the physical address and access control bits (*C*, *D*, and *V*) are retrieved from the matching TLB entry. While the *V* bit of the entry must be set for a valid translation to take place, it is not involved in the determination of a matching TLB entry. Figure 7.20 illustrates the TLB address translation process.

Notes

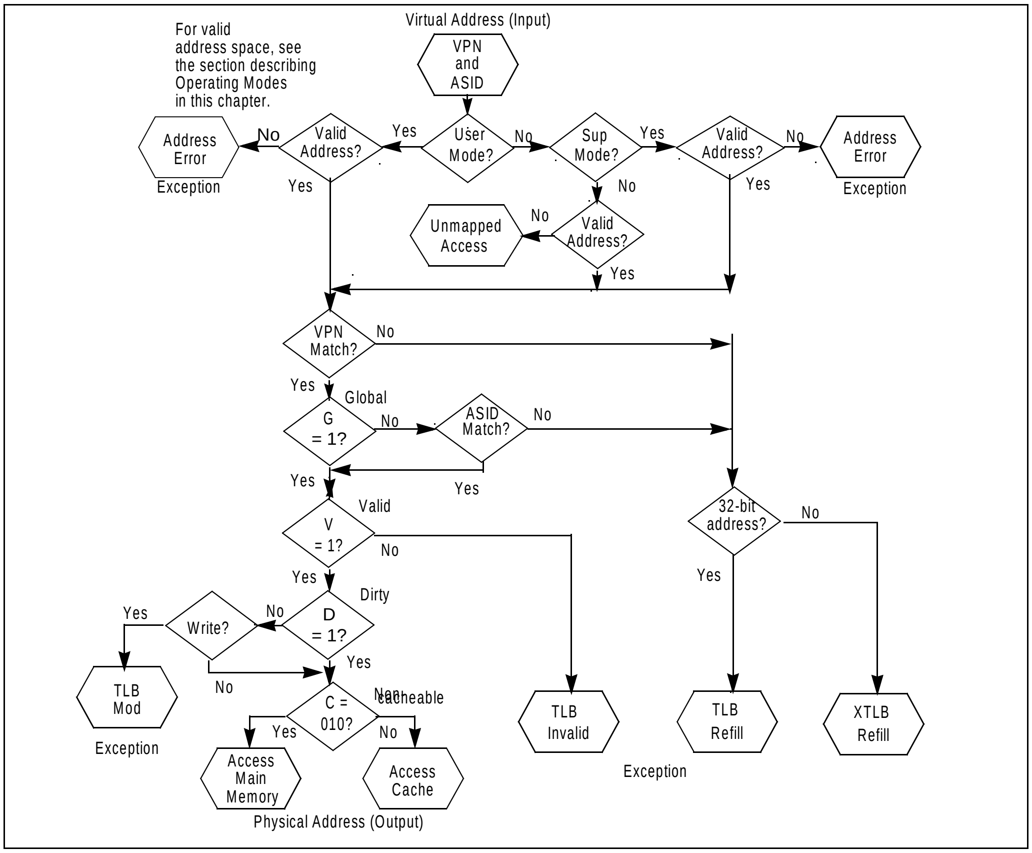


Figure 7.20 TLB Address Translation

## TLB Hits and Misses

If there is a virtual address match, or hit, in the TLB, the physical page number is extracted from the TLB and concatenated with the offset to form the physical address. If there is no TLB entry that matches the virtual address, a TLB miss exception occurs and software refills the TLB from the page table resident in memory. Software can write over a selected TLB entry or use a hardware mechanism to write into a random entry.

## Multiple TLB Matches

The processor does not provide any detection or shutdown mechanism for multiple matches in the TLB. The result of this condition is undefined, and software is expected to never allow this to occur.

## Invalid TLB Accesses

If the access control bits (*D* and *V*) indicate that the access is not valid, a TLB modification or TLB invalid exception occurs. If the *C* bits equal $010_2$, the physical address that is retrieved accesses main memory, bypassing the cache.

## TLB Instructions

Table 7.14 lists the instructions that the CPU provides for working with the TLB.

| Op Code | Description of Instruction |
|---|---|
| TLBP | Translation Lookaside Buffer Probe |
| TLBR | Translation Lookaside Buffer Read |
| TLBWI | Translation Lookaside Buffer Write Index |
| TLBWR | Translation Lookaside Buffer Write Random |

Table 7.14 TLB Instructions

# Cache

Notes

## Introduction

This chapter describes the on-chip primary cache, the individual operations of the primary cache, and the organization and operations of the on-chip secondary cache controller.

Figure 8.1 shows the R5000 system memory hierarchy. In the logical memory hierarchy, caches lie between the CPU and main memory. They are designed to make the speedup of memory accesses transparent to the user. Each functional block in Figure 8.1 has the capacity to hold more data than the block above it. At the same time, each functional block takes longer to access than any block above it. To improve, the speed of access to stored instructions and data, the processor has two on-chip primary caches, one for instruction and one for data, plus an on-chip secondary-cache controller.

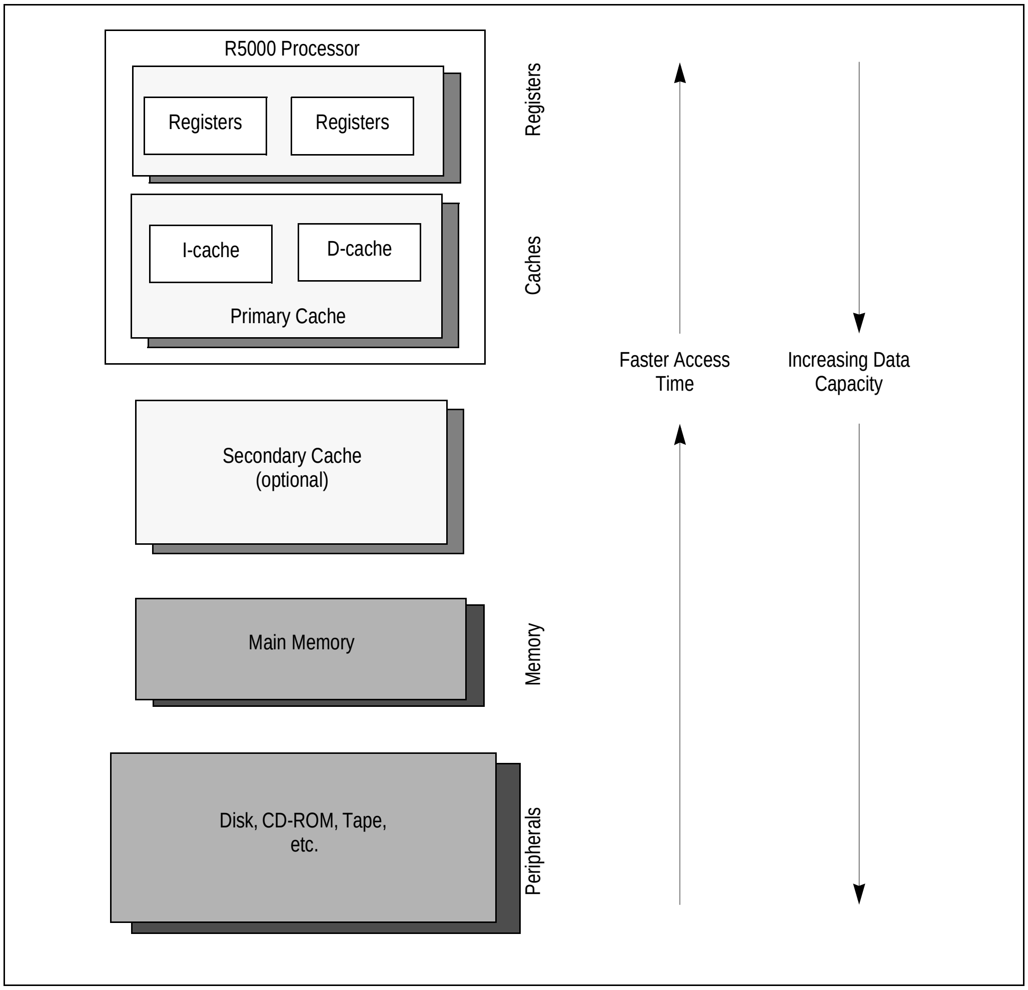


**Figure 8.1 Logical Hierarchy of Memory**

Caches provide fast, temporary data storage, and they make the speedup of memory accesses transparent to the user. In general, the processor attempts to access the next-required instruction or datum in the primary cache. A successful access is called a primary-cache hit. If the instruction/data is not present in the primary cache, it is retrieved as a cache line from secondary cache (if installed) or memory and is written into the primary cache. For a data cache miss, the processor can restart the pipeline after the first double-word (the one at the miss address) is retrieved and continues the cache line refill in parallel.

Notes

It is possible for the same data to be in three places simultaneously: main memory, secondary cache, and primary cache. This data is kept consistent through the use of either a write-back or a write-through protocol. For a write-back cache, the modified data is not written to memory until the cache line is replaced. In a write-through cache, the data is written to memory when the cached data is modified (with a possible delay due to the write buffer).

## Primary Caches

This section describes the organization of on-chip primary caches and the optional secondary cache.

### Cache Line Size

A *cache line* is the smallest unit of information that can be fetched from memory to be filled into the cache. A primary cache line is 8 words (32 bytes) in length and is represented by a single tag. Upon a cache miss in the primary cache, the missing cache line is loaded from memory into the primary cache.

### Cache Organization and Accessibility

This section describes the organization of the primary cache, including the manner in which it is mapped, the addressing used to index the cache, and composition of the cache lines. The primary instruction and data caches are indexed with a virtual address (VA).

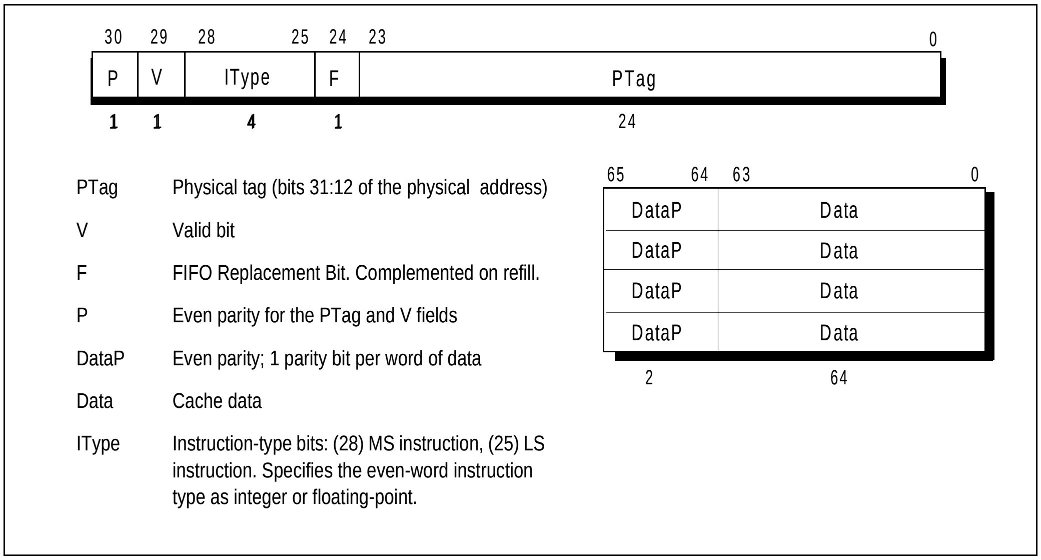
### Organization of the Primary Instruction Cache (I-Cache)

Each line of primary I-cache data (although it is actually an instruction, it is referred to as data to distinguish it from its tag) has an associated 31-bit tag that contains a 24-bit physical address, a single valid bit, a reserved bit, a single parity bit and the FIFO replacement bit. Word parity is used on I-cache data. Instruction-type bits determine whether the even-word instruction is an integer instruction or a floating-point instruction. A four-bit IType field is created to be written with the tag. The four bits correspond to each of the four instructions associated with this tag, with the most-significant bit corresponding to the most-significant instruction.

The primary I-cache has the following characteristics:

- *two-way set associative.*
- *indexed with a virtual address.*
- *checked with a physical tag.*
- *organized with 8-word (32-byte) cache line.*

Figure 8.2 shows the format of a primary I-cache line.



| | |
|---|---|
| PTag | Physical tag (bits 31:12 of the physical address) |
| V | Valid bit |
| F | FIFO Replacement Bit. Complemented on refill. |
| P | Even parity for the PTag and V fields |
| DataP | Even parity; 1 parity bit per word of data |
| Data | Cache data |
| IType | Instruction-type bits: (28) MS instruction, (25) LS instruction. Specifies the even-word instruction type as integer or floating-point. |

**Figure 8.2 Primary I-Cache Line Format**

Notes

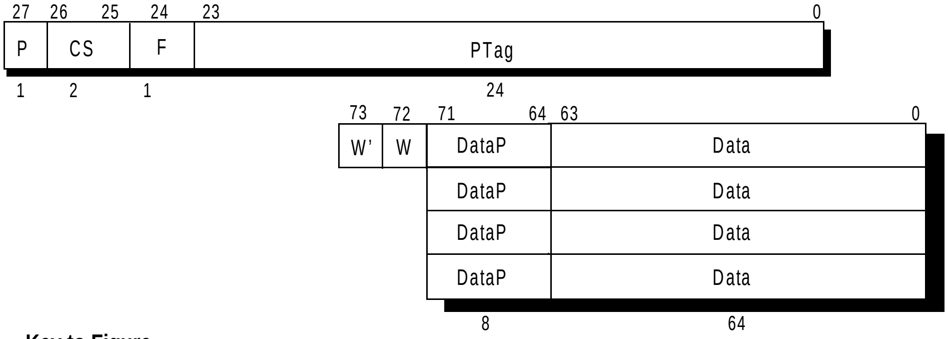
## Organization of the Primary Data Cache (D-Cache)

Each line of primary D-cache data has an associated 28-bit tag that contains a 24-bit physical address, 2-bit cache line state, a write-back bit, a parity bit for the physical address and cache state fields, a parity bit for the write-back bit, and the FIFO replacement bit.

The primary D-cache has the following characteristics:

- *write-back or write-through on a per-page basis.*
- *two-way set associative.*
- *indexed with a virtual address.*
- *checked with a physical tag.*
- *organized with 8-word (32-byte) cache line.*

Figure 8.3 shows the format of a primary D-cache line. In the R5000, the *W* (write-back) bit, not the cache state, indicates whether or not the primary cache contains modified data that must be written back to memory. *There is no hardware support for cache coherency. The only cache states used are Dirty Exclusive and Invalid.*



**Key to Figure:**

| | |
|---|---|
| F | FIFO Replacement Bit |
| W' | Even parity for the write-back bit |
| W | Write-back bit (set if cache line has been written) |
| P | Even parity for the PTag and CS fields |
| CS | Primary cache state:<br>0 = Invalid, 1 = Shared,<br>2 = Clean Exclusive, 3 = Dirty Exclusive |
| PTag | Physical tag (bits 35:12 of the physical address) |
| DataP | Even parity for the data; 1-bit per byte |
| Data | Cache data |

**Figure 8.3 8-Word Primary Data-Cache Line Format**

## Accessing the Primary Caches

Figure 8.4 shows the virtual address (VA) index into the primary caches. Each instruction and data cache size is 32 Kbytes.

Notes

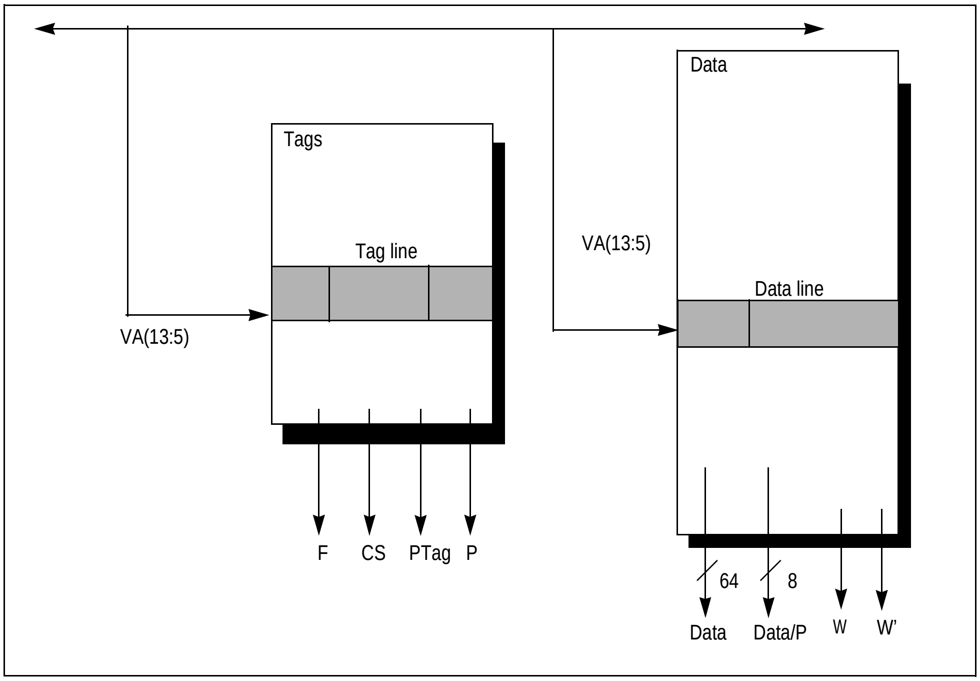


Figure 8.4 Primary Cache Data and Tag Organization

## Secondary Cache Controller

### Organization

Figure 8.5 shows a block diagram of how a secondary cache might be configured to the R5000 processor. Figure 8.6 shows a timing diagram of hit and miss read followed by write cycles. Figure 8.7 shows a timing diagram of basic read and write cycles.

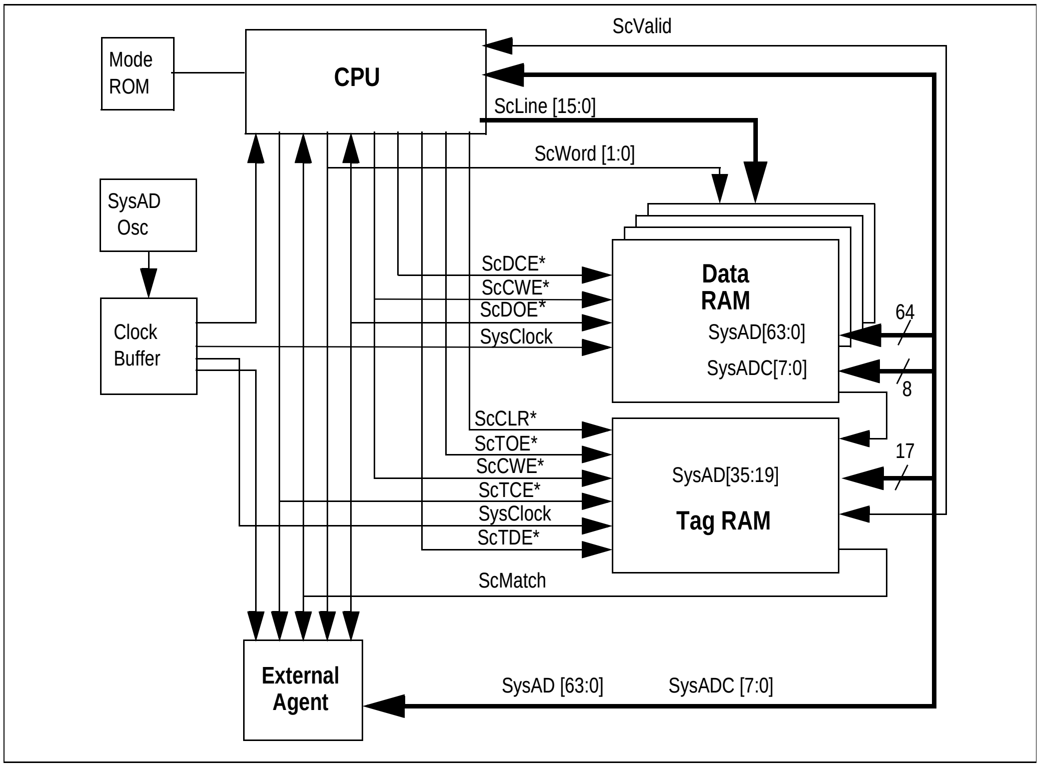


Figure 8.5 Secondary Cache Block Diagram

Notes

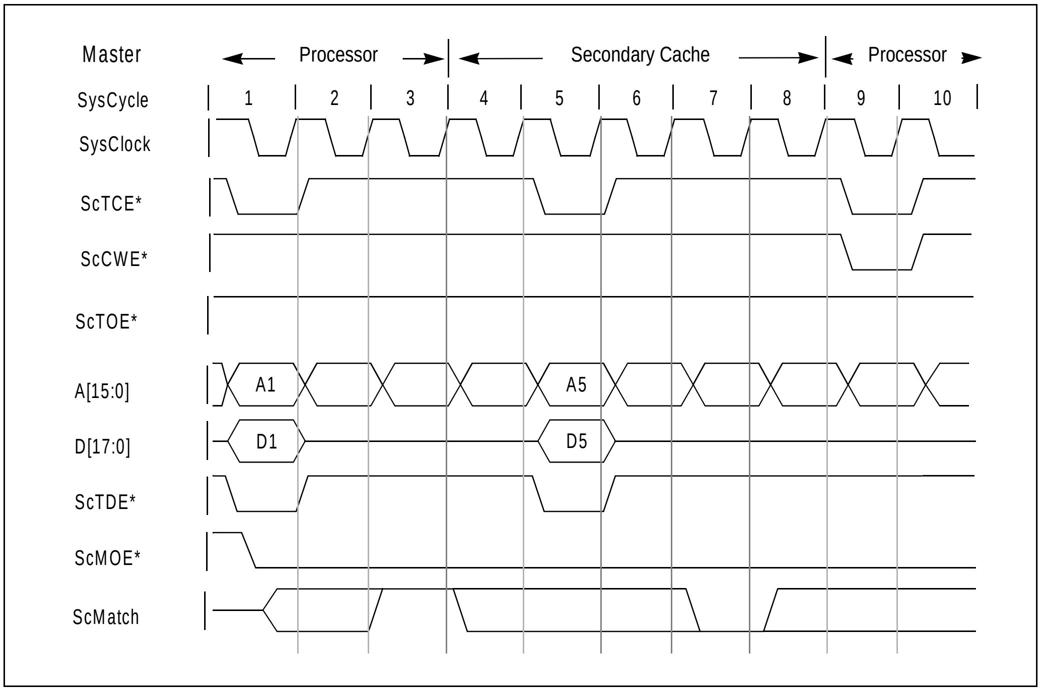


**Figure 8.6 Tag RAM Hit and Miss Read-Followed-By-Write Cycles**

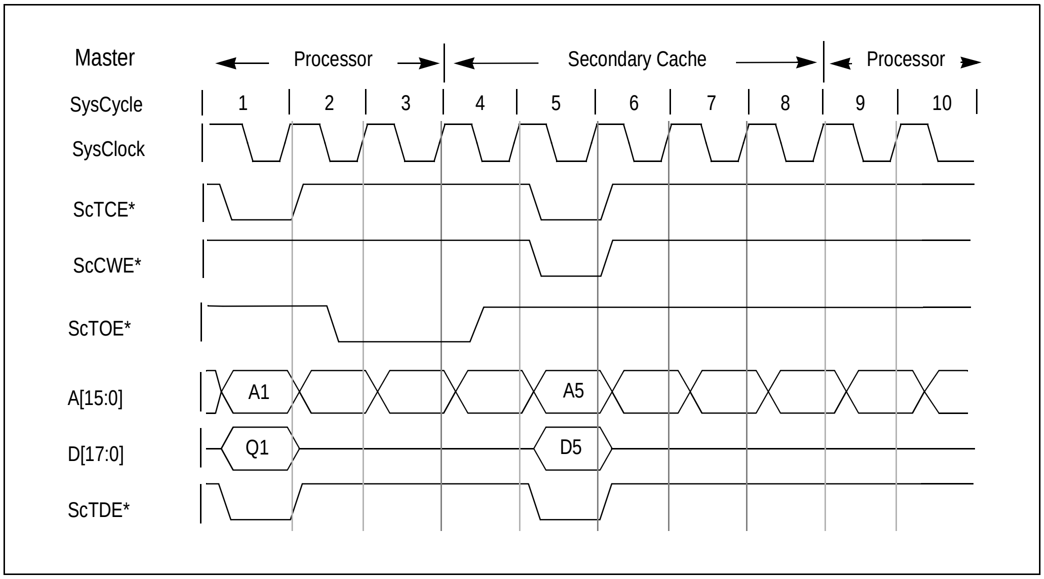


**Figure 8.7 Tag RAM Read and Write Cycles**

## Interface Block Diagram

The data RAMs are pipelined synchronous SRAMs with registered -inputs and outputs. The chip enable and write enable signals are pipelined. The output enable signal is asynchronous. Figure 8.8 shows a block diagram of the data RAMs.

Notes

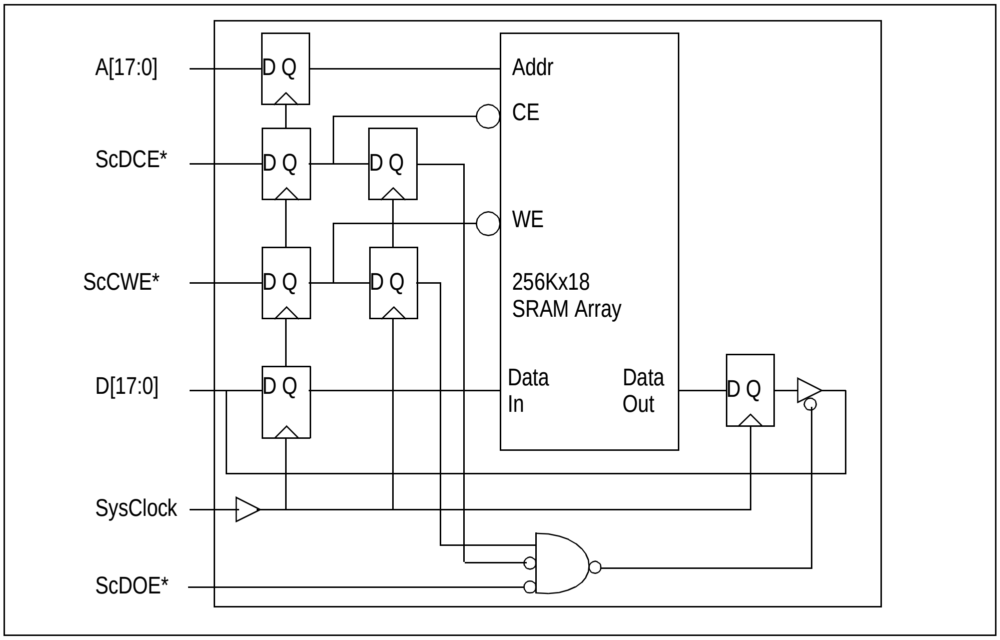


**Figure 8.8 Data RAM Block Diagram**

As illustrated in the above block diagrams, the RAMs synchronously enable their outputs two cycles after a read operation is issued, and synchronously disable their outputs two cycles after the end of a read operation. The tag RAM has the same architecture as the data RAM with the addition of a load enable signal for the data input register and a registered comparator output of the data input register and the RAM array. The tag RAM may optionally support a flash clear of the valid bit column.

Figure 8.9 shows a block diagram of the tag RAM.

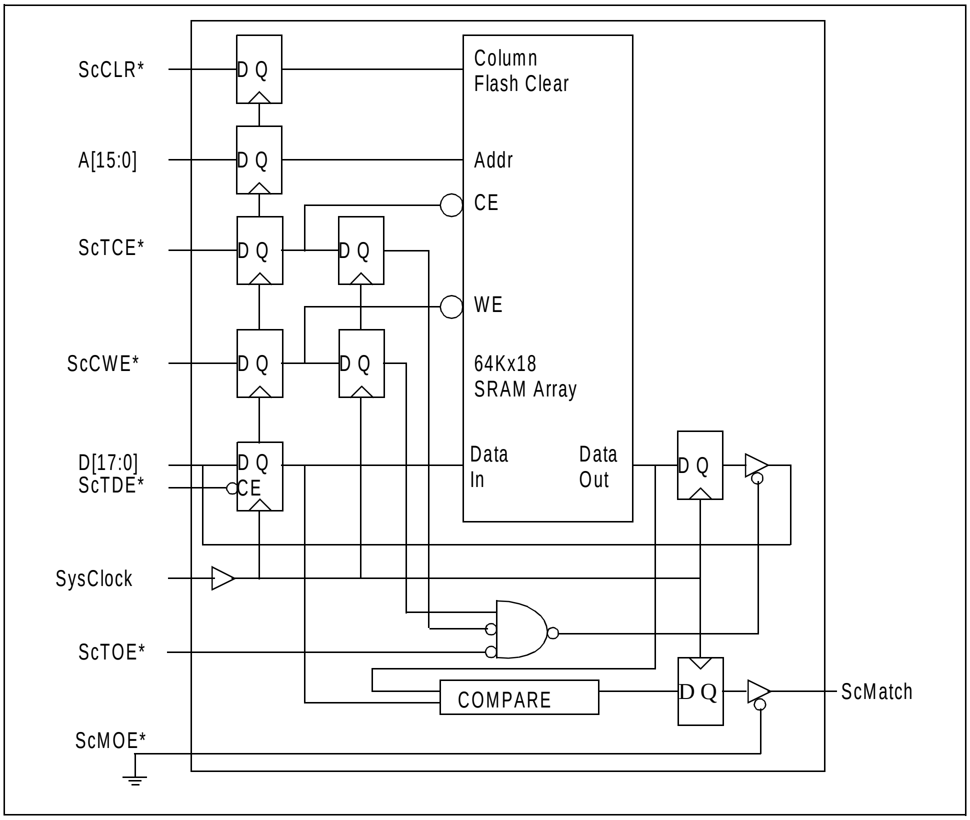


**Figure 8.9 Tag RAM Block Diagram**

Notes

### Secondary Cache Operations

The CACHE instruction defines two operations for the secondary cache: *index load tag* and *index store tag*. The following orientation of the index bits determine the type of operation:

- *Index Bits [17:16] equal to 11b (3h) specifies the secondary cache.*
- *Index bits [20:18] equal to 001b (1h) specifies the index load tag. The index load tag reads the secondary cache for the specified index and places it into the TagLo CP0 register.*
- *Index bits [20:18] equal to 010b (2h) specify the index store tag. The index store tag writes the secondary cache for the specified index from the physical address generated by the CACHE instruction.*
- *Index bits [20:18] equal to 000b (0h) generates a valid clear sequence to flush the entire cache in one operation.*
- *Index bits [20:18] equal to 101b (5h) generates a cache page invalidate instruction to flush 128 lines of the cache in one operation with the tag value from the TagLo CP0 register. The index for the cache page invalidate must be page aligned.*
- *Interrupts are deferred until a cache page invalidate instruction completes (up to 512 processor clocks for a SysClock ratio of 4).*
- *TagLo[12] is the valid bit and TagLo[31:15] is the tag for all secondary cache operations.*

### Secondary Cache Mode Configuration

The secondary cache configuration is specified by the processor ROM mode serial bit [12]. The state of this bit is indicated by the Secondary Cache (SC) bit in the CP0 configuration register (bit 17). If bit [17] is zero, a secondary cache is present in the system.

If no secondary cache is present and the mode ROM is configured for no secondary cache, the ScMatch and ScDOE* signals become don’t-care inputs and must be terminated to valid logic levels, and the processor drives all secondary cache signals to their inactive state. If the secondary cache is present and enabled, then the SysADC signals must implement valid parity during block read responses.

The doublewords transferred on SysAD during secondary-cache block read transactions are in *subblock order* (see Chapter 11 for a description of subblock order). The doublewords transferred on SysAD during secondary cache block write transactions are in sequential order.

The size of the secondary cache is indicated by the processor mode ROM serial bits [17:16], and are encoded as follows:

- *[17:16] = 00 - 512 Kbyte*
- *[17:16] = 01 - 1 Mbyte*
- *[17:16] = 10 - 2 Mbyte*
- *[17:16] = 11 - Reserved*

The state of these bits appear as configuration register bits [21:20].

### Secondary Cache Software Enable

The secondary cache may be enabled or disabled by software control via CP0 configuration register bit 12 (SE). When the SE bit is set (1) the secondary cache is enabled. When the SE bit is cleared (0) the secondary cache is disabled. The SE bit is cleared at reset. When the secondary cache is enabled by setting the SE bit, the state of the cache is undefined and software must explicitly invalidate the entire secondary cache before using it.

## Cache-Line States

The terms below are used to describe the *state* of a cache line:

- ***Exclusive****: a cache line that is present in exactly one cache in the system is exclusive. This is always the case for the R5000. All cache lines are in an exclusive state.*
- ***Dirty****: a cache line containing data that has changed since it was loaded from memory.*

Notes

- ***Clean**: a cache line containing data that has not changed since it was loaded from memory.*
- ***Shared**: a cache line present in more than one cache in the system. The R5000 does not provide for hardware cache coherency. This state will never happen in normal operations.*

The R5000 supports the four data-cache states shown in Table 8.1. Under normal operations, however, the only cache-line states that will occur in the data cache are the *Dirty Exclusive* and *Invalid* states. Each primary instruction cache line is either valid or invalid.

Although valid data is in the Dirty Exclusive state, it may still be consistent with memory. One must look at the *dirty bit*, W, to determine if the cache line is to be written back to memory when it is replaced.

| Cache Line State | Description |
|---|---|
| Invalid | A cache line that does not contain valid information must be marked invalid, and cannot be used. A cache line in any other state than invalid is assumed to contain valid information. |
| Shared | A cache line that is present in more than one cache in the system is shared. This state will not occur for normal operations. |
| Clean Exclusive | A clean exclusive cache line contains valid information and this cache line is not present in any other cache. The cache line is consistent with memory and is not owned by the processor (see “Cache-Line Ownership” on page 8-8). This state will not occur for normal operations. |
| Dirty Exclusive | A dirty exclusive cache line contains valid information and is not present in any other cache. The cache line may or may not be consistent with memory and is owned by the processor (see “Cache-Line Ownership” on page 8-8). Use the W bit to determine if the line must be written back on replacement. |

**Table 8.1 Cache States**

## Cache-Line Ownership

The processor is the owner of a cache line when it is in the *dirty exclusive* state, and is responsible for the contents of that line. There can only be one owner for each cache line. The ownership of a cache line is set and maintained through the rules described below.

- *A processor assumes ownership of the cache line if the state of the primary cache line is dirty exclusive.*
- *A processor that owns a cache line is responsible for writing the cache line back to memory if the line is replaced during the execution of a Write-back or Write-back Invalidate cache instruction if the line is in a write-back page. The cache instruction is explained further in the IDT MIPS Microprocessor Family Software Reference Manual.*
- *Memory always owns clean cache lines.*
- *The processor gives up ownership of a cache line when the state of the cache line changes to invalid.*

Therefore, based on these rules and that any valid data cache line is in the Dirty Exclusive state (under normal operating conditions), the processor is considered to be the owner of the cache line.

## Cache Write Policy

The R5000 manages its primary data cache by using either a write-back or a write-through policy on a per-page basis. Four policies are supported:

- *Uncached*
- *Writeback*
- *Write-Through With Write Allocation*
- *Write-Through With No Write Allocation.*

Uncached writes do not modify the cache and are written directly to main memory.

A write-back policy does not write data to main memory until the cache line is replaced or the CACHE instruction flushes and possibly invalidates the dirty cache line. If a write hits in cache, the data is stored in cache and the cache line is marked dirty: no main memory access occurs. If a write misses in cache, the replaced cache line is written to memory, if dirty. The new cache line is read, the data is written to cache, and the cache line is marked dirty.

A write-through policy writes the data to main memory, immediately. If a write hits in cache, the data is stored in cache and written to memory. The cache line is *not* marked dirty, since cache and main memory have the same value. If the write misses in the cache and the policy is write-through with write allocation, the replaced cache line is written to memory, if dirty; the new cache line is read, the data is stored in cache and written to memory. If the policy is write-through with no write allocate, the data is written to memory and does not modify the cache.

| Cache Write Policy | Cache Hit | Cache Miss |
|---|---|---|
| Uncached | N/A | Write to main memory |
| Write-back | Store in cache<br>Cache line marked dirty | Flush cache line (if dirty)<br>Read new cache line<br>Store in cache<br>Cache line marked dirty |
| Write-Through<br>Write Allocate | Store in cache<br>Write to main memory | Flush cache line (if dirty)<br>Read new cache line<br>Store in cache<br>Write to main memory |
| Write-Through<br>No Write Allocate | Store in cache<br>Write to main memory | Write to main memory |

**Table 8.2 CPU Cache Write Policy**

## Cache-State Transitions

Figure 8.10 shows the cache-state transitions for the primary cache. When an external agent supplies a cache line, it need not return the initial state of the cache line for normal operations (refer to Chapter 10 for a definition of an external agent). This is because the only read request the R5000 should issue are for non-coherent data, and the lower three bits for the data identifier are reserved. The initial state will automatically be set to dirty exclusive by the R5000. Otherwise, the processor changes the state of the cache line during one of the following events:

- *A store to a dirty exclusive line remains in a dirty exclusive state.*
- *The state is changed to invalid for:*
  - *for a Cache invalidate operation*
  - *if the line is replaced*

.

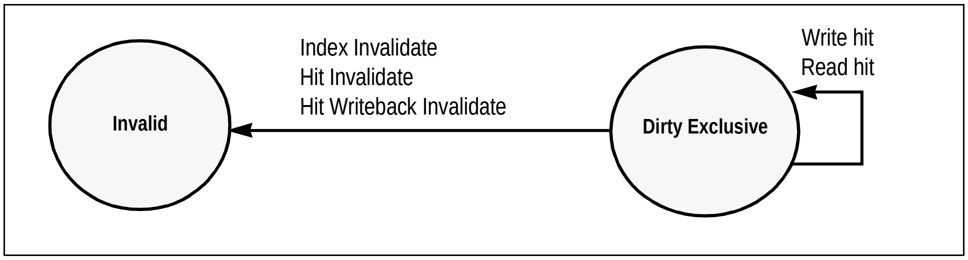


**Figure 8.10 Primary Data Cache State Diagram**

## Cache Coherency

Systems using more than one master must have a mechanism to maintain data consistency throughout the system. This mechanism is called a cache coherency protocol. The R5000 does not provide any hardware cache coherency. Cache coherency must be handled with software.

Notes

### Cache Coherency Attributes

The TLB contains 3 bits per entry that provide two possible cache-coherency attribute types, *uncached* and *noncoherent*. These attribute types can be used to control cache coherency on a per-page basis. Table 8.3 lists the two attribute types and summarizes the behavior of the processor on load misses and store misses for each type.

| Attribute Type | Load Miss | Store Miss |
|---|---|---|
| Uncached | Main-memory read | Main-memory write |
| Noncoherent | Noncoherent read | Noncoherent read (write-allocate page)<br>Main-memory write (no write-allocate page) |

**Table 8.3 Coherency Attributes and Processor Behavior**

### Uncached Attribute

Lines within an *uncached* page are never in a cache. When a virtual address has the uncached coherency attribute, the processor issues a doubleword, partial-doubleword, word, or partial-word read or write request directly to main memory (bypassing the cache) for any load or store to a location within that page.

### Noncoherent Attribute

Lines with a *noncoherent* attribute type can reside in a cache. A load miss causes the processor to issue a noncoherent block read request to a location within the cached page. For a store miss to a write-allocate page, the processor issues a noncoherent block read request to a location within the cached page and then does the write-through. If the virtual address has the no write-allocate attribute, a store miss will generate a write to the memory as in the uncached case.

## Multiprocessor Synchronization Support

In a multiprocessor system, it is essential that two or more processors working on a common task can execute without corrupting each other’s subtasks. *Synchronization*, an operation that guarantees an orderly access to shared memory, must be implemented for a properly functioning multiprocessor system. Two of the more widely used methods are discussed in this section: *test-and-set*, and *counter*. Even though the R5000 does not support symmetric multi-processing (SMP), these are useful for multi-master and heterogeneous multi-processing.

### Test-and-Set

Test-and-set uses a variable called the *semaphore*, which protects data from being simultaneously modified by more than one processor. In other words, a processor can lock out other processors from accessing shared data when the processor is in a *critical section*, a part of program in which no more than a fixed number of processors is allowed to execute. In the case of test-and-set, only one processor can enter the critical section.

Figure 8.11 illustrates a test-and-set synchronization procedure that uses a semaphore; when the semaphore is cleared to 0, the shared data is unlocked, and when the semaphore is set to 1, the shared data is locked.

Notes

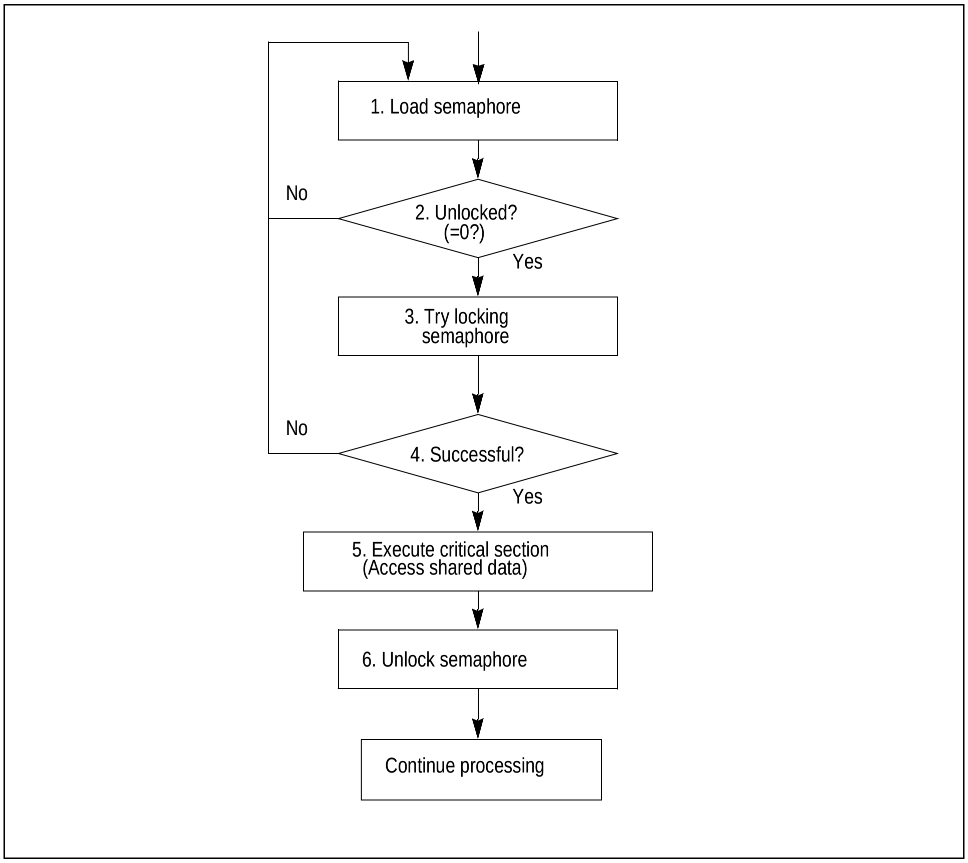


**Figure 8.11 Synchronization with Test-and-Set**

The processor begins by loading the semaphore and checking to see if it is unlocked (0) in steps 1 and 2. If the semaphore is not 0, the processor loops back to step 1. If the semaphore is 0, indicating the shared data is not locked, the processor next tries to lock out any other access to the shared data (step 3). If not successful, the processor loops back to step 1, and reloads the semaphore.

If the processor is successful at setting the semaphore (step 4), it executes the critical section of code (step 5) and gains access to the shared data, completes its task, unlocks the semaphore (step 6), and continues processing.

### Counter

Another common synchronization technique uses a *counter.* A counter is a designated memory location that can be incremented or decremented. In the test-and-set method, only one processor at a time is permitted to enter the critical section. Using a counter, up to *N* processors are allowed to concurrently execute the critical section. All processors after the *N*th processor must wait until one of the *N* processors exits the critical section and a space becomes available.

The counter works by not allowing more than one processor to modify it at any given time. Conceptually, the counter can be viewed as a variable that counts the number of limited resources (for example, the number of processes, or software licenses, etc.). Figure 8.12 shows this process.

Notes

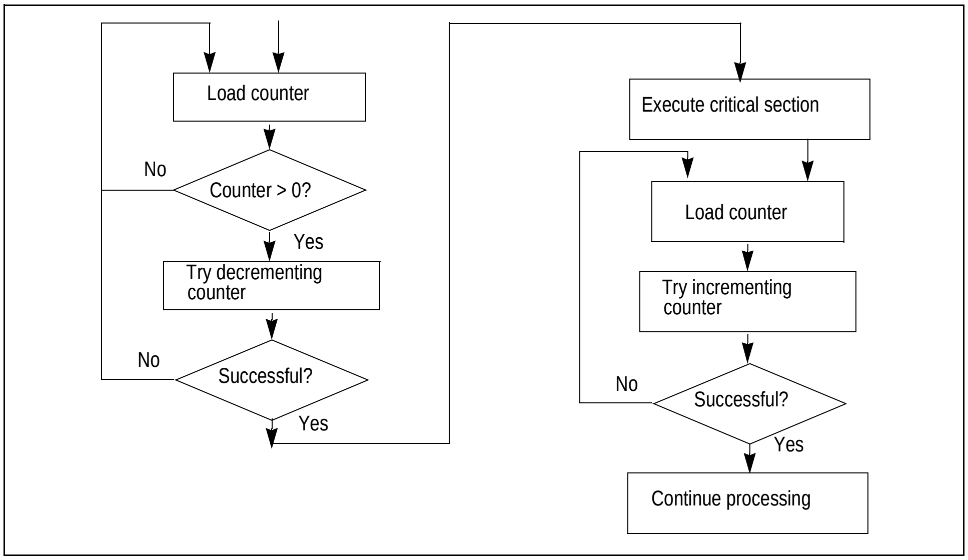


**Figure 8.12 Synchronization Using a Counter**

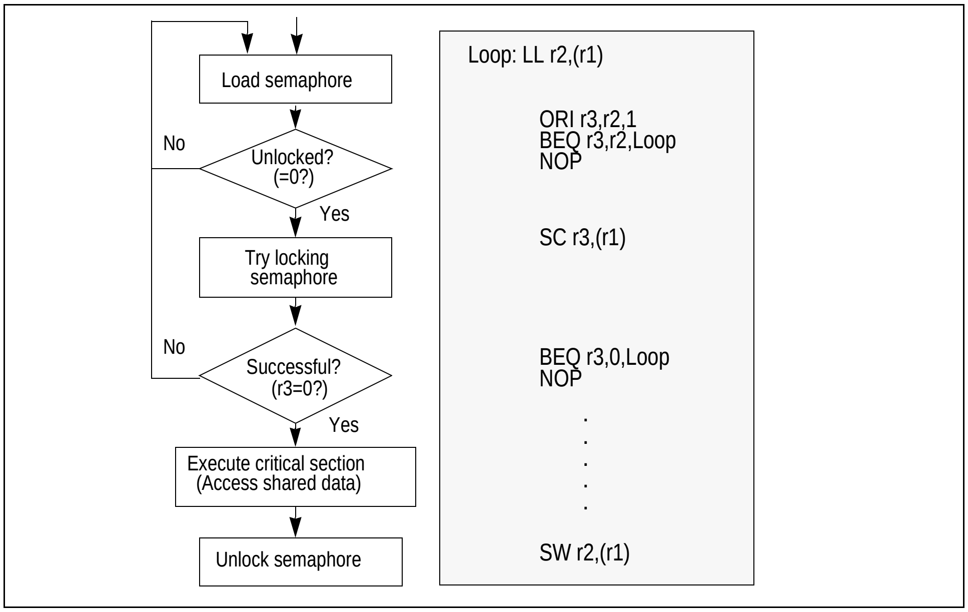
## Load Linked and Store Conditional

The R5000 instructions *Load Linked* (LL) and *Store Conditional* (SC) provide support for processor synchronization. These two instructions work very much like their simpler counterparts, load and store. The LL instruction, in addition to doing a simple load, has the side effect of setting a bit called the *link bit*. This link bit forms a breakable link between the LL instruction and the subsequent SC instruction. The SC performs a simple store if the link bit is set when the store executes. If the link bit is not set, then the store fails to execute. The success or failure of the SC is indicated in the target register of the store. The link is broken upon completion of an ERET (return from exception) instruction.

The most important features of LL and SC are that:

- *they provide a mechanism for generating all of the common synchronization primitives including test-and-set, counters, sequencers, etc., with no additional overhead*
- *when they operate, bus traffic is generated only if the state of the cache line changes; lock words stay in the cache until some other processor takes ownership of that cache line*

Figure 8.13 shows how to implement test-and-set using LL and SC instructions.

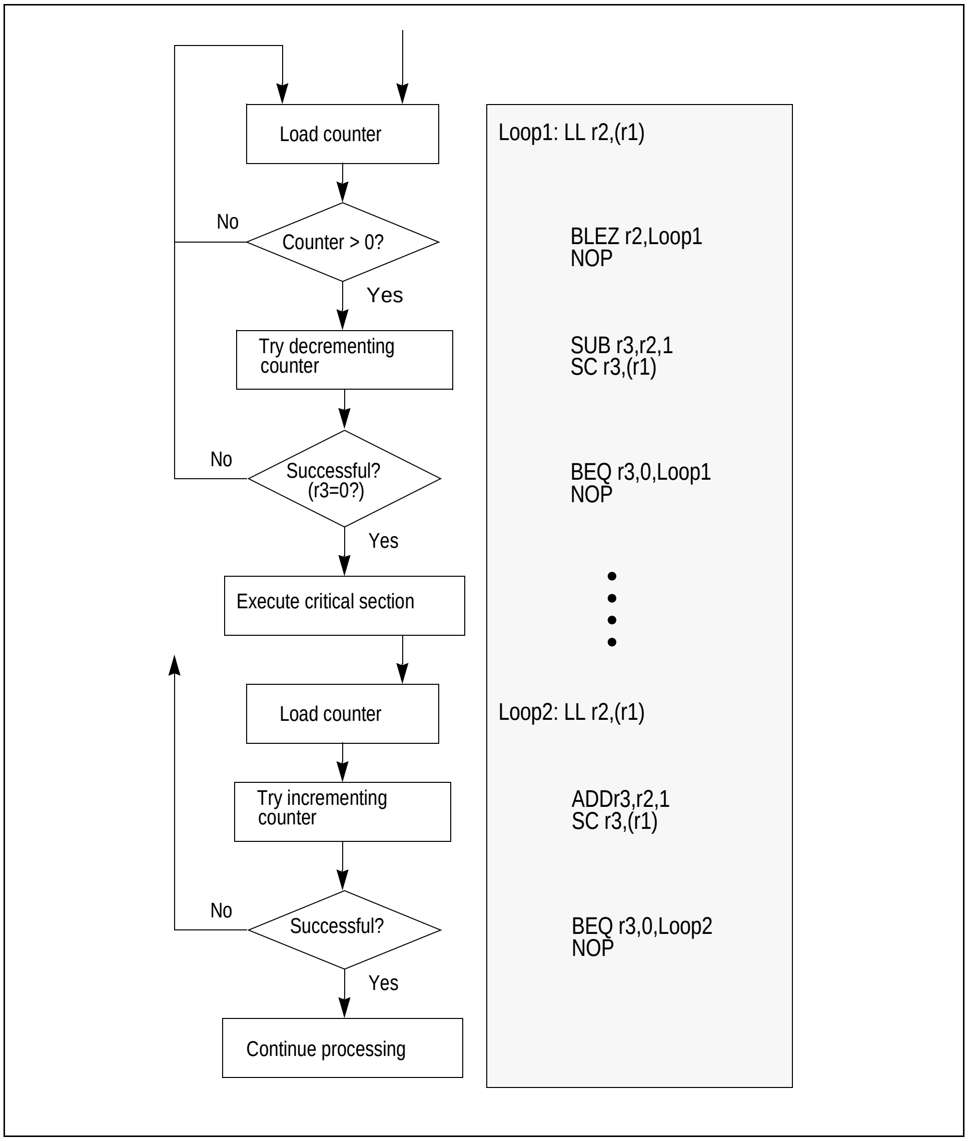


```
Loop: LL r2,(r1)

          ORI r3,r2,1
          BEQ r3,r2,Loop
          NOP

          SC r3,(r1)

          BEQ r3,0,Loop
          NOP
            .
            .
            .
            .
            .

          SW r2,(r1)
```

**Figure 8.13 Test-and-Set using LL and SC**

## Notes

Figure 8.14 shows synchronization using a counter.



```
Loop1: LL r2,(r1)

            BLEZ r2,Loop1
            NOP

            SUB r3,r2,1
            SC r3,(r1)

            BEQ r3,0,Loop1
            NOP

                •
                •
                •
                •

Loop2: LL r2,(r1)

            ADDr3,r2,1
            SC r3,(r1)

            BEQ r3,0,Loop2
            NOP
```

**Figure 8.14 Counter Using LL and SC**

## Notes

# Signal Descriptions

Notes

## Introduction

This chapter describes the signals used by and in conjunction with the R5000 processor. The signals include the System Interface, Clock Interface, Secondary Cache Interface, Interrupt Interface, Joint Test Action Group (JTAG) Interface, and Initialization Interface signals.

Figure 9.1 illustrates the functional groupings of the processor signals. Active-low signals have a trailing asterisk—for instance, the low-active Read Ready signal is **RdRdy***.

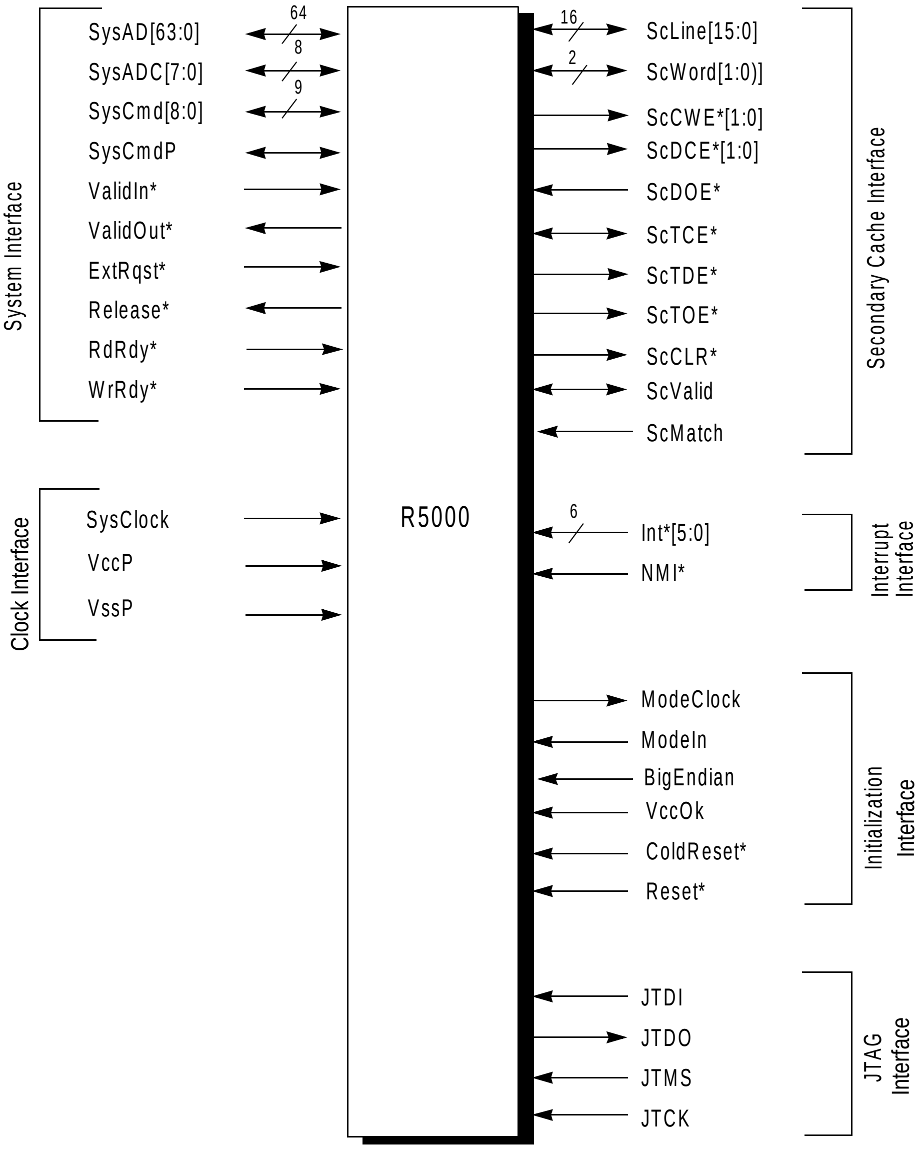


**Figure 9.1 R5000 Processor Signals**

Notes

## System Interface Signals

System interface signals provide the connection between the R5000 processor and the other components in the system. Table 9.1 lists the system interface signals.

| Name | Definition | Direction | Description |
|---|---|---|---|
| ExtRqst* | External request | Input | An external agent asserts **ExtRqst*** to request use of the System interface. The processor grants the request by asserting **Release***. |
| Release* | Release interface | Output | In response to the assertion of **ExtRqst***, the processor asserts **Release***, signalling to the requesting device that the System interface is available. |
| RdRdy* | Read ready | Input | The external agent asserts **RdRdy*** to indicate that it can accept processor read requests in either secondary or no-secondary cache mode. |
| SysAD(63:0) | System address/data bus | Input/Output | A 64-bit address and data bus for communication between the processor, the secondary cache, and an external agent. |
| SysADC(7:0) | System address/ data check bus | Input/Output | An 8-bit bus containing parity for the **SysAD** bus. **SysADC** is valid on data cycles only. |
| SysCmd(8:0) | System command/ data identifier | Input/Output | A 9-bit bus for command and data identifier transmission between the processor and an external agent. |
| SysCmdP | System command/ data identifier bus parity | Input/Output | Always zero when driven by the processor. Never checked by the processor. This signal is defined to maintain R4000 compatibility. |
| ValidIn* | Valid input | Input | The external agent asserts **ValidIn*** when it is driving a valid address or data on the **SysAD** bus and a valid command or data identifier on the **SysCmd** bus. |
| ValidOut* | Valid output | Output | The processor asserts **ValidOut*** when it is driving a valid address or data on the **SysAD** bus and a valid command or data identifier on the **SysCmd** bus to the external agent. |
| WrRdy* | Write ready | Input | The external agent asserts **WrRdy*** when it can accept a processor write request. |

**Table 9.1 System Interface Signals**

## Clock Interface Signals

The Clock interface signals make up the interface for clocking. Table 9.2 lists the Clock interface signals.

| Name | Definition | Direction | Description |
|---|---|---|---|
| SysClock | System Clock | Input | System clock input that establishes the system interface operating frequency and phase. |
| VccP | **Quiet Vcc for PLL** | Input | Quiet Vcc for the internal phase locked loop. |
| VssP | Quiet Vss for PLL | Input | Quiet Vss for the internal phase locked loop. |

**Table 9.2 Secondary Cache Interface Signals**

## Secondary Cache Interface Signals

Secondary Cache interface signals constitute the interface between the R5000 processor and secondary cache. Table 9.3 lists the Secondary Cache interface signals in alphabetical order.

Notes

| Name | Definition | Direction | Description |
|---|---|---|---|
| ScCLR* | Secondary Cache Block Clear | Output | Clears all valid bits in those Tag RAMs which support this function. |
| ScCWE*(1:0) | Secondary Cache Write Enable | Output | Asserted during writes to the secondary cache. Two signals are provided to minimize loading from the cache RAMs. |
| ScDCE*(1:0) | Data RAM Chip Enable | Output | Chip Enable for Secondary Cache Data RAM. Two signals are provided to minimize loading from the cache RAMs. |
| ScDOE* | Data RAM Output Enable | Input | Asserted by the external agent to enable data onto the **SysAD** bus |
| ScLine (15:0) | Secondary Cache Line Index | Output | Cache line index for secondary cache |
| ScMatch | Secondary cache Tag Match | Input | Asserted by Tag RAM on Secondary cache tag match |
| ScTCE* | Secondary cache Tag RAM Chip Enable | Output | Chip enable for secondary cache tag RAM. |
| ScTDE* | Secondary cache Tag RAM Data Enable | Output | Data Enable for Secondary Cache Tag RAM. |
| ScTOE* | Secondary cache Tag RAM Output Enable | Output | Tag RAM Output enable for Secondary Cache Tag RAM |
| ScWord (1:0) | Secondary cache Word Index | Input/Output | Determines correct double-word of Secondary cache Index |
| ScValid | Secondary cache Valid | Input/Output | Always driven by the CPU except during a CACHE Probe operation, where it is driven by the Tag RAM. |

**Table 9.3 Secondary Cache Interface Signals**

## Interrupt Interface Signals

The Interrupt interface signals make up the interface used by external agents to interrupt the R5000 processor. Table 9.4 lists the Interrupt interface signals.

| Name | Definition | Direction | Description |
|---|---|---|---|
| Int*(5:0) | Interrupt | Input | General processor interrupts, bit-wise ORed with bits 5:0 of the interrupt register. |
| NMI* | Nonmaskable interrupt | Input | Nonmaskable interrupt, ORed with bit 6 of the interrupt register. |

**Table 9.4 Interrupt Interface Signals**

## JTAG Interface Signals

The JTAG interface signals make up the boundary scan interface. Table 9.5 lists the JTAG interface signals. *While JTAG signals are shown on the pinout, the JTAG interface is not supported by the R5000.*

| Name | Definition | Direction | Description |
|---|---|---|---|
| JTDI | JTAG data in | Input | Data is serially scanned in through this pin. |
| JTCK | TAG clock input | Input | The processor accepts a serial clock on **JTCK**. On the rising edge of **JTCK**, both **JTDI** and **JTMS** are sampled. |
| JTDO | JTAG data out | Output | Data is serially scanned out through this pin on the falling edge of **JTCK**. |
| JTMS | JTAG command | Input | JTAG command signal, indicating the incoming serial data is command data. |

**Table 9.5 JTAG Interface Signals**

## Notes

# Initialization Interface Signals

The Initialization interface signals make up the interface by which an external agent initializes the processor operating parameters. Table 9.6 lists the Initialization interface signals.

| Name | Definition | Direction | Description |
|---|---|---|---|
| BigEndian | Endian Mode Select | Input | Allows the system to change the processor addressing mode without rewriting the mode ROM. If endianness is to be specified via the BigEndian pin, program mode ROM bit 8 to zero. If endianness is to be specified by the mode ROM, ground the BigEndian pin. |
| ColdReset* | Cold reset | Input | This signal must be asserted for a power on reset or a cold reset. **ColdReset*** must be deasserted synchronously with **SysClock**. |
| ModeClock | Boot mode clock | Output | Serial boot-mode data clock output; runs at the system clock frequency divided by 256: (**SysClock**/256). |
| ModeIn | Boot mode data in | Input | Serial boot-mode data input. |
| Reset* | Reset | Input | This signal must be asserted for any reset sequence. It can be asserted synchronously or asynchronously for a cold reset, or synchronously to initiate a warm reset. **Reset*** must be deasserted synchronously with **SysClock**. |
| VccOk | Vcc is OK | Input | When asserted, this signal indicates to the processor that the Vcc Min power supply has been above 3.135 volts for more than 100 milliseconds and will remain stable. The assertion of **VccOk** initiates the initialization sequence. |

**Table 9.6 Initialization Interface Signals**

# System Interface Transactions

Notes

## Introduction

The System interface allows the processor to access external resources that are needed to satisfy cache misses and uncached operations, while permitting an external agent access to some of the processor internal resources. This chapter describes the system interface from the point of view of both the processor and the external agent.

The system interface of the R5000 processor is a modification of the R4600 system interface. The clock portion of the system interface has been simplified and many of the external clock signals have been deleted.

The R5000 processor supports up to a 100 MHz pipelined SysAD bus. R5000 also implements a unified, write-through secondary cache controller which has the same 32-byte line size as the primary caches. Secondary cache index and control signals are supplied by the processor. Secondary cache sizes of 512 KByte, 1 MByte, and 2 MByte are supported.

## Terminology

The following terms are used in this chapter:

An *external agent* is any logic device connected to the processor over the system interface that allows the processor to issue requests.

A *system event* is an event that occurs within the processor and requires access to external system resources.

*Sequence* refers to the precise series of requests that a processor generates to service a system event.

*Protocol* refers to the cycle-by-cycle signal transitions that occur on the system interface pins to assert a processor or external request.

*Syntax* refers to the precise definition of bit patterns on encoded buses, such as the command bus.

## Processor Requests

When a system event occurs, the processor issues either a single request or a series of requests—called *processor requests*—through the System interface, to access an external resource and service the event. For this to work, the processor System interface must be connected to an external agent that is compatible with the System interface protocol, and can coordinate access to system resources.

An external agent requesting access to a processor internal resource generates an *external request*. This access request passes through the System interface. System events and request cycles are shown in Figure 10.1

Notes

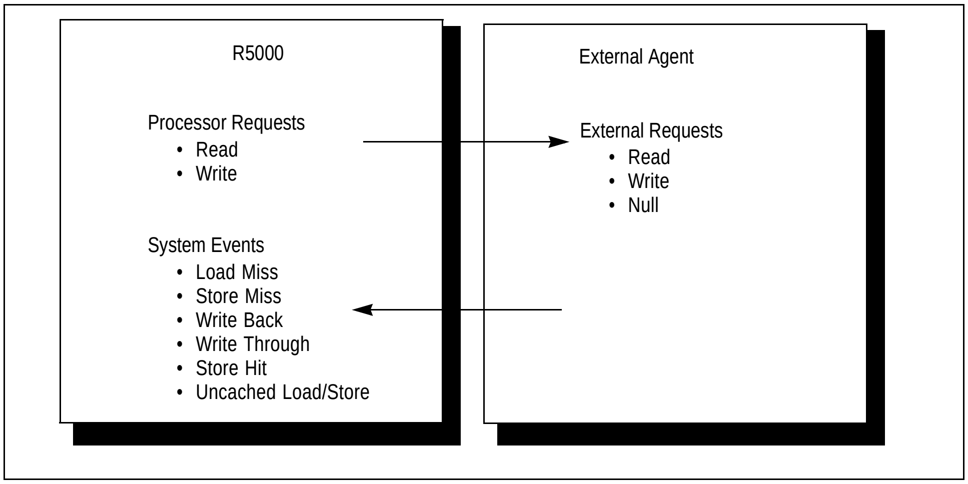


**Figure 10.1 Requests and System Events**

### Rules for Processor Requests

A processor request is a request or a series of requests, through the System interface, to access some external resource. As shown in Figure 10.2, processor requests include read and write.

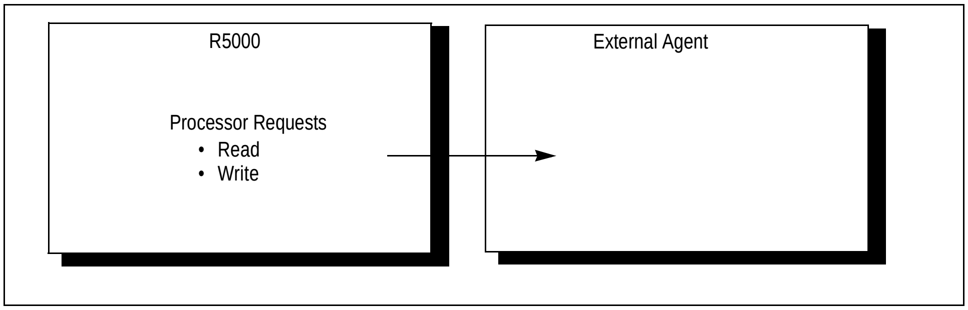


**Figure 10.2 Processor Requests to External Agent**

A *read request* asks for a partial word, word, partial doubleword, doubleword, or larger block of data either from main memory or from another system resource. A *write request* provides a partial word, word, partial doubleword, doubleword, or larger block of data to be written either to main memory or to another system resource.

The processor is only allowed to have one request pending at any time. For example, the processor issues a read request and waits for a read response before issuing any subsequent requests. The processor submits a write request only if there are no read requests pending.

The processor has the input signals RdRdy* and WrRdy* to allow an external agent to manage the flow of processor requests. RdRdy* controls the flow of processor read requests, while WrRdy* controls the flow of processor write requests. The processor request cycle sequence is shown in Figure 10.3.

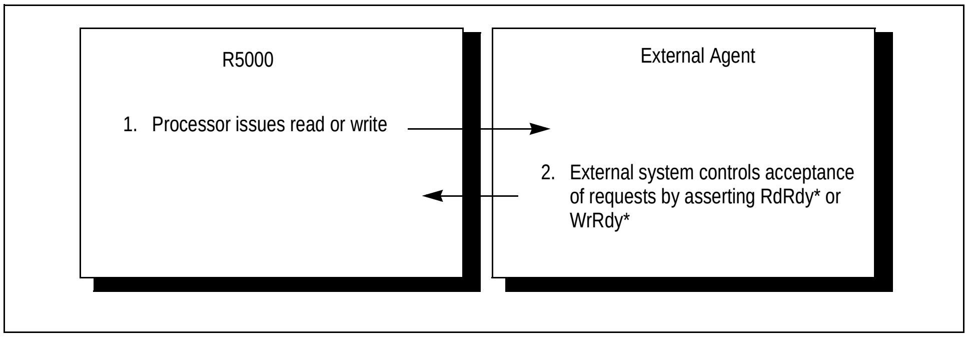

R5000
1. Processor issues read or write
External Agent
2. External system controls acceptance of requests by asserting RdRdy* or WrRdy*

**Figure 10.3 Processor Request Flow Control**

Notes

### Processor Read Request

When a processor issues a read request, the external agent must access the specified resource and return the requested data. A processor read request can be split from the external agent's return of the requested data; in other words, the external agent can initiate an unrelated external request before it returns the response data for a processor read. A processor read request is completed after the last word of response data has been received from the external agent. The data identifier associated with the response data can signal that the returned data is erroneous, causing the processor to take a bus error.

Processor read requests that have been issued, but for which data has not yet been returned, are said to be *pending*. A read remains pending until the requested read data is returned. The external agent must be capable of accepting a processor read request any time the following two conditions are met:

- *There is no processor read request pending.*
- *The signal RdRdy* has been asserted for two or more cycles before the issue cycle.*

### Processor Write Request

When a processor issues a write request, the specified resource is accessed and the data is written to it. A processor write request is complete after the last word of data has been transmitted to the external agent. The R5000 processor supports *R4000 compatible*, *write-reissue* and *pipelined write* operations as defined in section 4. The external agent must be capable of accepting a processor write request any time the following two conditions are met:

- *No processor read request is pending.*
- *The signal* ***WrRdy**** *has been asserted for two or more cycles.*

## External Requests

This section describes external requests, which include *read, write*, and *null* requests, and *read response*, a special case of an external request. The external requests (except read-response, which is described later) are shown in Figure 10.4:

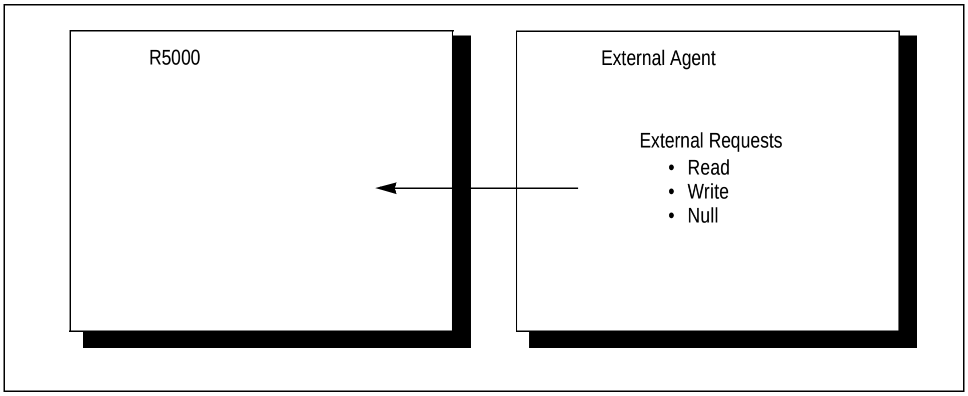


**Figure 10.4 External Requests to Processor (except Read Response)**

- *Read request asks for a word of data from the processor's internal resource.*
- *Write request provides a word of data to be written to the processor's internal resource.*
- *Null request requires no action by the processor; it provides a mechanism for the external agent to return the System interface to the master state without affecting the processor.*

The processor controls the flow of external requests through the arbitration signals ExtRqst* and Release*, as shown in Figure 10.5. The external agent must acquire mastership of the System interface before it is allowed to issue an external request; the external agent arbitrates for mastership of the System interface by asserting ExtRqst* and then waiting for the processor to assert Release* for one cycle. If Release* is asserted as part of an uncompelled change to slave state during a processor read request, and the secondary cache is enabled, the secondary cache access must be resolved and be a miss. Otherwise the system interface remains in the master state.

Notes

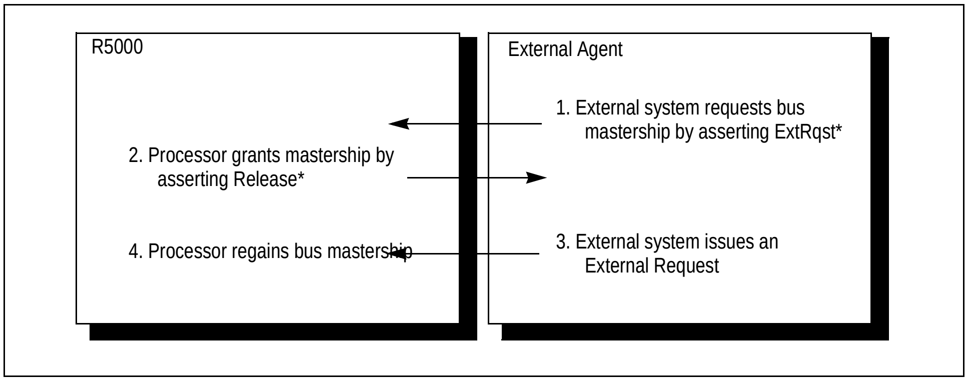

R5000
External Agent
1. External system requests bus mastership by asserting ExtRqst*
2. Processor grants mastership by asserting Release*
3. External system issues an External Request
4. Processor regains bus mastership

**Figure 10.5 External Request Arbitration**

Mastership of the System interface always returns to the processor after an external request is issued. The processor does not accept a subsequent external request until it has completed the current request. If there are no processor requests pending, the processor decides, based on its internal state, whether to accept the external request, or to issue a new processor request. The processor can issue a new processor request even if the external agent is requesting access to the System interface.

The external agent asserts ExtRqst* indicating that it wishes to begin an external request. The external agent then waits for the processor to signal that it is ready to accept this request by asserting Release*. The processor signals that it is ready to accept an external request based on the following criteria:

- *The processor completes any request in progress.*
- *While waiting for the assertion of RdRdy* to issue a processor read request, the processor can accept an external request if the request is delivered to the processor one or more cycles before RdRdy* is asserted.*
- *While waiting for the assertion of WrRdy* to issue a processor write request, the processor can accept an external request provided the request is delivered to the processor one or more cycles before WrRdy* is asserted.*
- *If waiting for the response to a read request after the processor has made an uncompelled change to a slave state, the external agent can issue an external request before providing the read response data.*

### External Read Request

The processor does not contain any resources that are readable by an external read request. In response to an external read request, the processor returns undefined data and a data identifier with its Erroneous Data bit, SysCmd(5), set.

### External Write Request

When an external agent issues a write request, the specified resource is accessed and the data is written to it. An external write request is complete after the word of data has been transmitted to the processor. *The only processor resource available to an external write request is the* IP *field of the Cause register.*

### Read Response

A *read response* returns data in response to a processor read request, as shown in Figure 10.6. While a read response is technically an external request, it has one characteristic that differentiates it from all other external requests—it does not perform System interface arbitration. For this reason, read responses are handled separately from all other external requests, and are simply called read responses. The data identifier associated with the response data can signal that the returned data is erroneous, causing the processor to take a bus error.

Notes

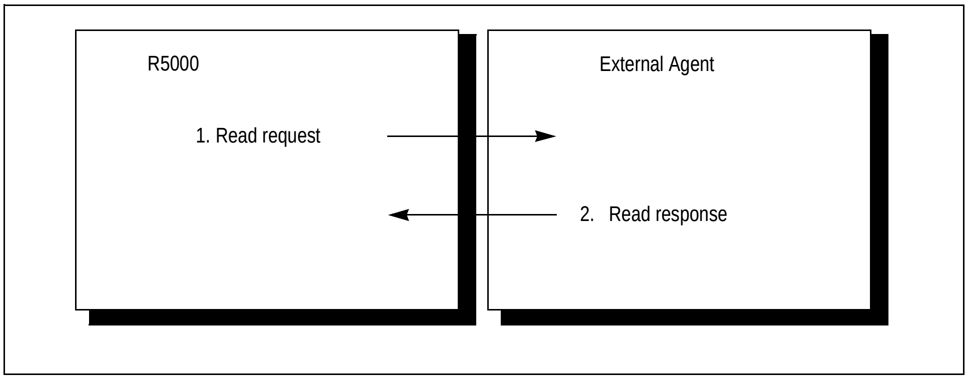


**Figure 10.6 External Agent Read Response to Processor**

## Secondary Cache Transactions

For processors configured with a secondary cache, the secondary cache is a special form of external agent that is jointly controlled by both the processor and the external agent. Figure 10.7 illustrates a processor request to the secondary cache and external agent.

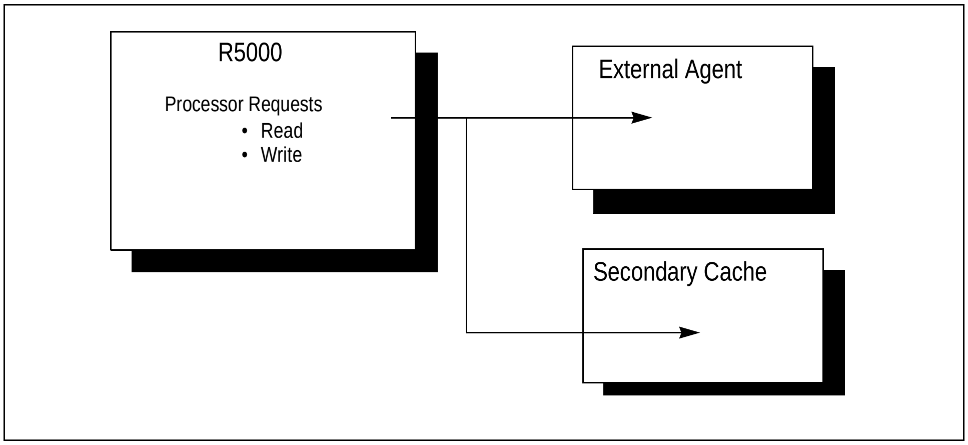


**Figure 10.7 Processor Requests to Secondary Cache and External Agent**

### Secondary Cache Probe, Invalidate, and Clear

For secondary cache invalidate, clear, and probe operations, the secondary cache is controlled by the processor and the external agent is not involved in these operations. Issuance of secondary cache invalidate, clear, and probe operations is not flow-controlled and proceeds at the maximum data rate. Figure 10.8 and Figure 10.9 show the secondary cache invalidate and tag probe operations.

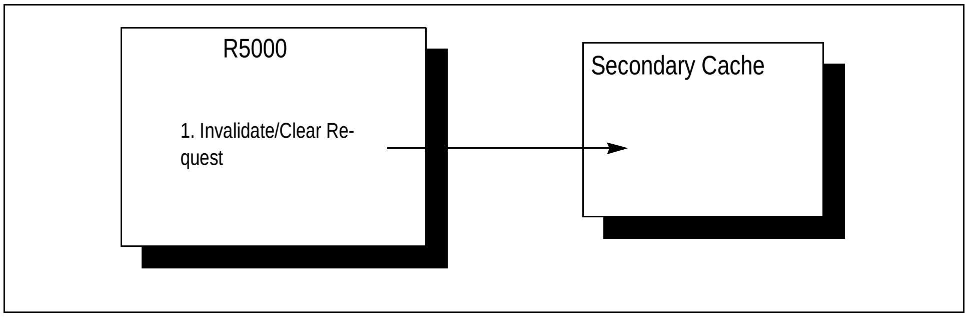


**Figure 10.8 Secondary Cache Invalidate and Clear**

Notes

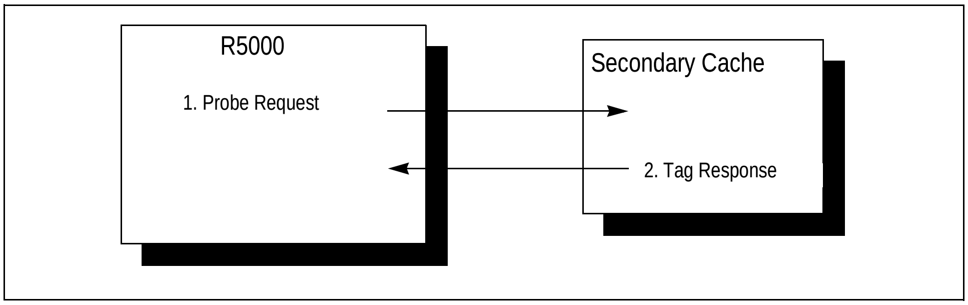


Figure 10.9 Secondary Cache Tag Probe

### Secondary Cache Write

For secondary cache write-through, the processor issues a block write operation that is directed to both the secondary cache and the external agent. Issuance of secondary cache writes is controlled by the normal WrRdy* flow control mechanism. Secondary cache write data transfers proceed at the data transfer rate specified in the Mode ROM for block writes. Figure 10.10 illustrates a secondary cache write operation.

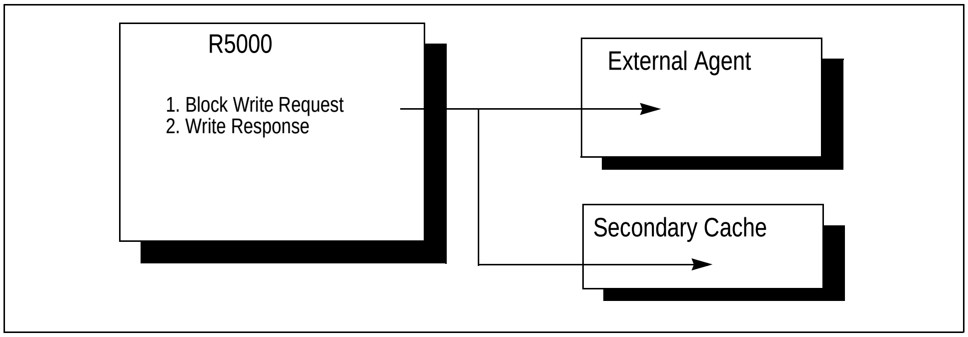


Figure 10.10 Secondary Cache Write Through

### Secondary Cache Read

For secondary cache reads, the processor issues a block read speculatively to both the secondary cache and the external agent.

- *If the block is present in the secondary cache, the secondary cache provides the read response and the block read to the external agent is aborted.*
- *If the block is not present in the secondary cache, the secondary cache read is aborted and the external agent provides the read response to both the secondary cache and the processor.*

Figure 10.11 and Figure 10.12 show a secondary cache read hit and miss respectively.

Notes

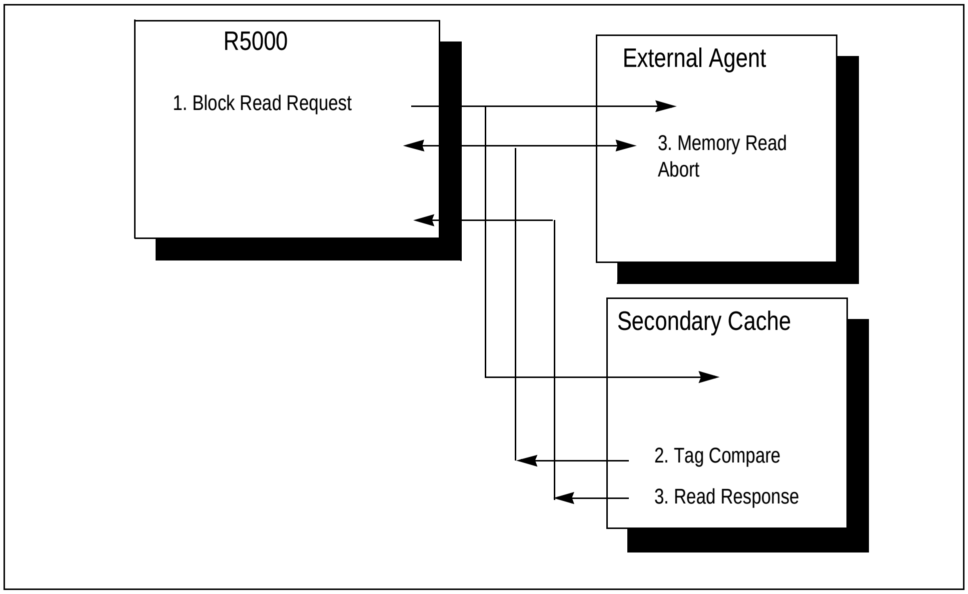


**Figure 10.11 Secondary Cache Read Hit**

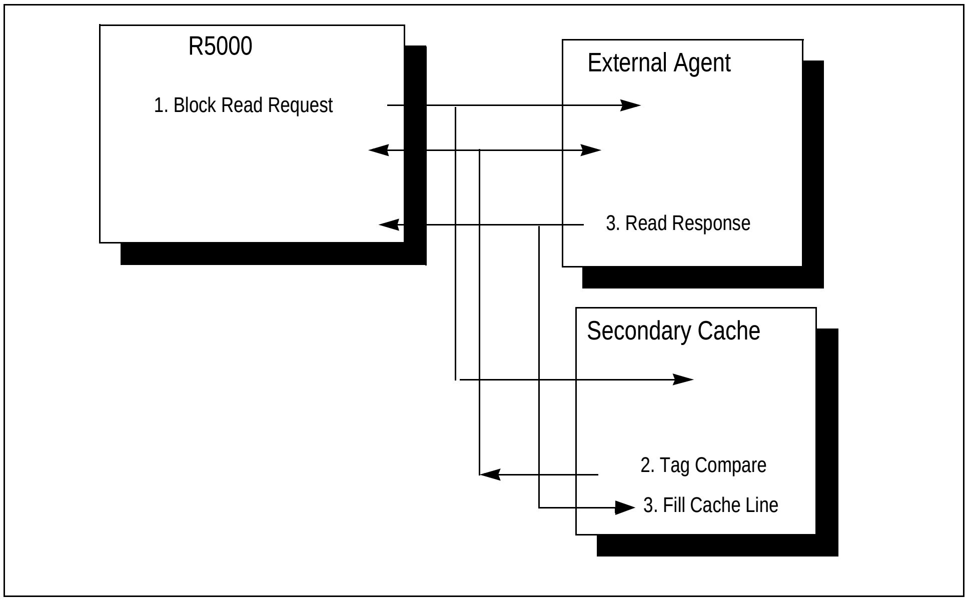


**Figure 10.12 Secondary Cache Read Miss**

Issuance of the secondary cache read is controlled by the normal RdRdy* flow control mechanism. Secondary cache read responses always proceed at the maximum data transfer rate. External agent read responses to the secondary cache proceed at the data transfer rate generated by the external agent.

## Handling Requests

This section details the *sequence*, *protocol*, and *syntax* of both processor and external requests. The following system events are discussed:

- *load miss*
- *store miss*
- *store hit*
- *uncached loads/stores*
- *load linked store conditional.*

Notes

### Load Miss

When a processor-load misses in the primary cache, before the processor can proceed it must obtain the cache line that contains the data element to be loaded from the external agent. The processor examines the coherency attribute in the TLB entry for the page that contains the requested cache line. Table 10.1 shows the actions taken on a load miss to the primary data cache.

| Page Attribute | State of Data Cache Line Being Replaced | |
|---|---|---|
| | Clean/Invalid | Dirty (W=1) |
| Noncoherent | NCBR<br>(Processor noncoherent block read request) | NCBR/W<br>(Processor noncoherent block read request followed by processor block write request) |

**Table 10.1 Action Taken On Load Miss to Primary Data Cache**

The processor performs the following steps:

1. Issues a noncoherent block read request for the cache line that contains the data element to be loaded. If the secondary cache is enabled and the page coherency attribute is write-back, the read request is also issued to the secondary cache.
2. Waits for an external agent to provide the read response.
3. Restarts the pipeline after the first doubleword of the data cache miss is received. The remaining three doublewords are placed in the cache in parallel with the pipeline restart.

If the new cache line replaces a current dirty exclusive or dirty shared cache line, the current cache line must be written back before the new line can be loaded in the primary cache. In this case, the processor issues a block write request to save the dirty cache line in memory. If the secondary cache is enabled and the page attribute is write-back, the write request is also issued to the secondary cache.

### Store Miss

When a processor-store misses in the primary cache, the processor may request, from the external agent, the cache line that contains the target location of the store for pages that are either write-back or write-through with write-allocate only. The processor examines the coherency attribute in the TLB entry for the page that contains the requested cache line to see if the cache line is being maintained with either a write-allocate or no-write-allocate.

The processor then executes one of the following requests:

- *If the coherency attribute is noncoherent write-back, or write-through with write-allocate, a noncoherent block read request is issued.*
- *If the coherency attribute is noncoherent write-through with no write-allocate, a non-block write request is issued.*

Table 10.2 shows the actions taken on a store miss to the primary cache.

| Page Attribute | State of Data Cache Line Being Replaced | |
|---|---|---|
| | Clean/Invalid | Dirty (W=1) |
| Noncoherent-write-back or noncoherent-write-through with write-allocate | NCBR<br>(Processor noncoherent block read request) | NCBR/W<br>(Processor noncoherent block read request followed by processor block write request) |
| Noncoherent-write-through with no-write-allocate | NCW<br>(Processor noncoherent write request) | NA |

**Table 10.2 Store Miss to Primary and Secondary Data Caches**

If the coherency attribute is write-back, or write-through with write-allocate, the processor issues a noncoherent block read request for the cache line that contains the data element to be loaded, then waits for the external agent to provide read data in response to the read request. If the secondary cache is enabled and the page coherency attribute is write-back, the read request is also issued to the secondary cache. If the current cache line must be written back, the processor issues a write request for the current cache line. If the page coherency attribute is write-through, the processor issues a non-block write request.

For a write-through, no-write-allocate store miss, the processor issues a non-block write request only.

### Store Hit

The action on the system bus is determined by whether the line is write-back or write-through. All lines that use a write-back policy are set to the dirty exclusive state. This means store hits cause no bus transactions. For lines with a write-through policy, the store generates a processor non-block write request for the store data.

### Uncached Loads or Stores

When the processor performs an uncached load, it issues a noncoherent doubleword, partial doubleword, word, or partial word read request. When the processor performs an uncached store, it issues a doubleword, partial doubleword, word, or partial word write request. All writes by the processor are buffered from the system interface by a 4-deep write buffer. The write requests are sent to the system bus only when no other requests are in progress. However, once the emptying of the write buffer has begun, it is allowed to complete. Therefore, if the write buffer contains any entries when a block read is requested, the write buffer is allowed to empty before the block read request is serviced. Uncached loads and stores do not affect the secondary cache.

### Uncached Instruction Fetch

The processor issues doubleword reads for instruction fetches to uncached addresses. Thus any system ROM address space accessed during a processor boot-restart must support 64-bit reads.

### Load Linked Store Conditional Operation

Generally, the execution of a Load-Linked/Store-Conditional instruction sequence is not visible at the System interface. That is, no special requests are generated due to the execution of this instruction sequence.

There is, however, one situation in which the execution of a Load Linked Store Conditional instruction sequence is visible, as indicated by the *link address retained* bit during a processor read request, as programmed by the SysCmd(2) bit. This situation occurs when the data location targeted by a Load Linked Store Conditional instruction sequence maps to the same cache line to which the instruction area containing the Load-Linked/Store- Conditional code sequence is mapped. In this case, immediately after executing the Load Linked instruction, the cache line that contains the link location is replaced by the instruction line containing the code. The link address is kept in a register separate from the cache, and remains active as long as the *link* bit, set by the Load Linked instruction, is set.

The *link* bit, which is set by the load linked instruction, is cleared by a change of cache state for the line containing the link address, or by a Return From Exception.

## Branch-Target Alignment

Since the instruction cache performs aligned fetches of two instructions per cycle from uncached addresses, compilers should attempt to align branch targets to allow dual-issue on the first target cycle. If code contains unaligned branch target, the processor requests an uncached doubleword read from an odd word address, which is an offset of 0x4 or 0xC. The external agent must ignore odd word addresses due to the alignment constraint. For example, if the processor issues an uncached doubleword read from address 0xBFCXXX04, the processor expects to read two instructions, one even-word instruction from 0xBFCXXX00 and one odd-word instruction from 0xBFCXXX04, even though the 0xBFCXXX00 instruction will not be used. The external agent must ignore the three least-significant bits [2:0] of the address because offsets 0x0 = 0x4 and 0x8 = 0xC.

Notes

# System Interface Protocols

Notes

## Introduction

The following sections contain a cycle-by-cycle description of the system interface protocols for each type of R5000 processor and external request.

## Address and Data Cycles

Cycles in which the **SysAD** bus contains a valid address are called *address cycles*. Cycles in which the **SysAD** bus contains valid data are called *data cycles*. Validity of addresses and data from the processor is determined by the state of the **ValidOut*** signal. Validity of addresses and data from the external agent is determined by the state of the **ValidIn*** signal. Validity of data from the secondary cache is determined by the state of the pipelined **ScDCE*** and **ScCWE*** signals from the processor and the **ScDOE*** signal from the external agent.

The **SysCmd** bus identifies the contents of the **SysAD** bus during any cycle in which it is valid from the processor or the external agent. The most significant bit of the **SysCmd** bus is always used to indicate whether the current cycle is an address cycle or a data cycle:

- *During address cycles **SysCmd(8)** = 0. The remainder of the **SysCmd** bus, **SysCmd(7:0)**, contains the encoded system interface command.*
- *During data cycles [**SysCmd(8)** = 1], the remainder of the **SysCmd** bus, **SysCmd(7:0)**, contains an encoded data identifier. There is no **SysCmd** associated with a secondary cache read response.*

## Issue Cycles

There are two types of processor issue cycles:

- *processor read request.*
- *processor write request.*

The processor samples the signal **RdRdy*** to determine the *issue* cycle for a processor read; the processor samples the signal **WrRdy*** to determine the *issue* cycle of a processor write request. As shown in Figure 11.1, **RdRdy*** must be asserted two cycles prior to the address cycle of the processor read request in order to define the address cycle as the issue cycle.

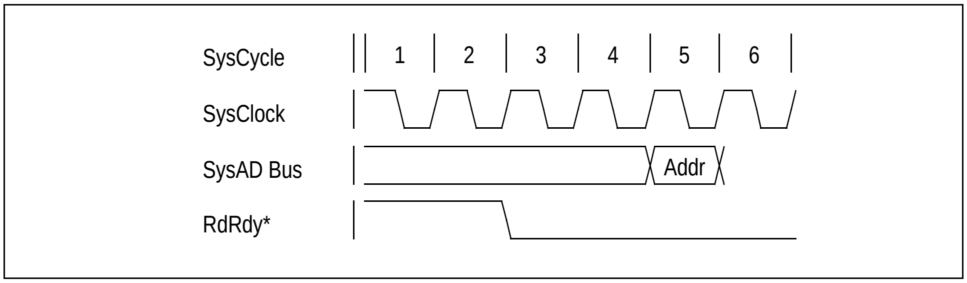


**Figure 11.1 State of RdRdy* Signal for Read Requests**

As shown in Figure 11.2, **WrRdy*** must be asserted two cycles prior to the first address cycle of the processor write request in order to define the address cycle as the issue cycle.

Notes

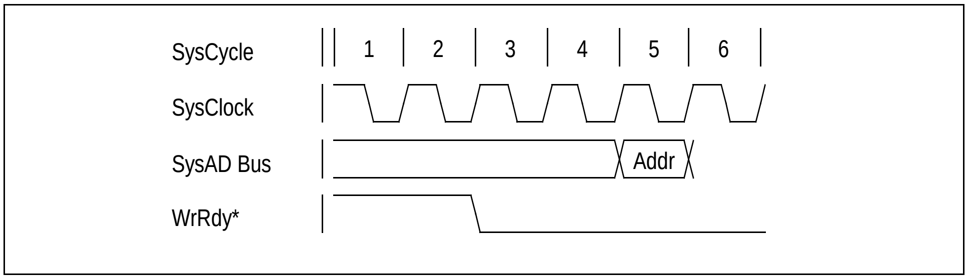


**Figure 11.2 State of WrRdy* Signal for Write Requests**

The processor repeats the address cycle for the request until the conditions for a valid issue cycle are met. After the issue cycle, if the processor request requires data to be sent, the data transmission begins. There is only one issue cycle for any processor request. The processor accepts external requests, even while attempting to issue a processor request, by releasing the System interface to slave state in response to an assertion of **ExtRqst*** by the external agent.

The rules governing the issue cycle of a processor request are strictly applied to determine which action the processor takes. The processor can either:

- *complete the issuance of the processor request in its entirety before the external request is accepted, or*
- *release the System interface to slave state without completing the issuance of the processor request.*

In the latter case, the processor issues the processor request (provided the processor request is still necessary) after the external request is complete. The rules governing an issue cycle again apply to the processor request.

## Handshake Signals

The processor manages the flow of requests through the following six control signals:

- ***RdRdy****, ***WrRdy**** *are used by the external agent to indicate when it can accept a new read (****RdRdy****) or write (****WrRdy****) transaction.*
- ***ExtRqst****, ***Release**** *are used to transfer control of the* ***SysAD*** *and* ***SysCmd*** *buses.* ***ExtRqst**** *is used by an external agent to indicate a need to control the interface.* ***Release**** *is asserted by the processor when it transfers the mastership of the System interface to the external agent. For secondary cache reads, assertion of* ***Release**** *to the external agent is speculative, and is aborted if there is a hit in the secondary cache.*
- *The R5000 processor uses* ***ValidOut**** *and the external agent uses* ***ValidIn**** *to indicate valid command/data on the* ***SysCmd/SysAD*** *buses.*
- *The secondary cache uses the* ***ScDCE****, ***ScCWE**** *and* ***ScDOE**** *signals to control validation on the* ***SysAD*** *and* ***SysADC*** *buses.*

## System Interface Operation

Figure 11.3 shows how the system interface operates from register to register. Processor outputs come directly from output registers and begin to change with the rising edge of **SysClock.** Processor inputs are fed directly to input registers that latch these input signals with the rising edge of **SysClock**. This allows the System interface to run at the highest possible clock frequency.

Notes

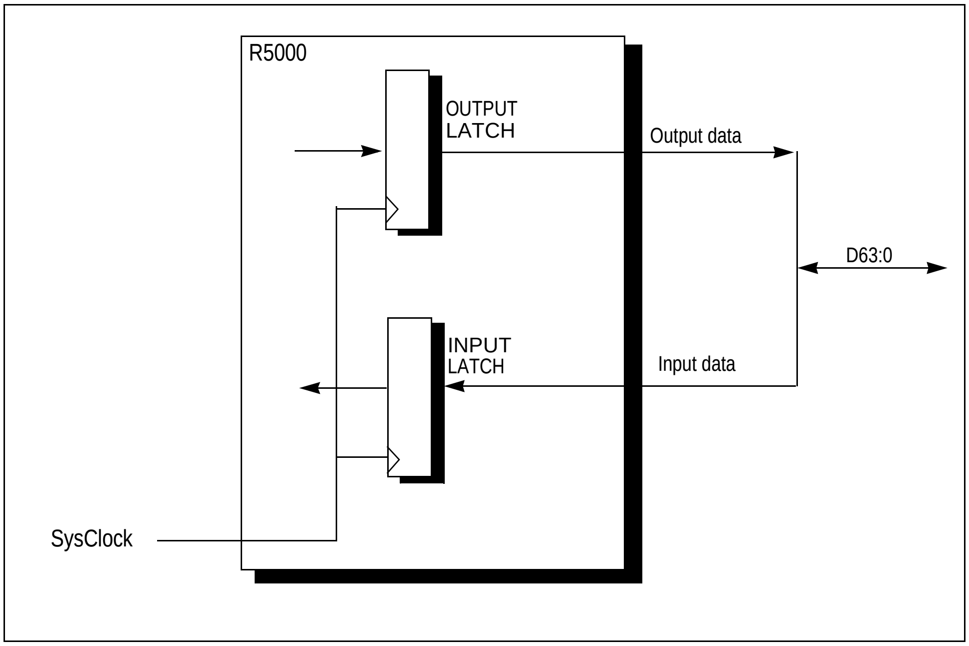


**Figure 11.3 System Interface Register-to-Register Operation**

## Master and Slave States

When the processor is driving the **SysAD** and **SysCmd** buses, the System interface is in *master state*. When the external agent is driving the **SysAD** and **SysCmd** buses, the System interface is in *slave state*. When the secondary cache is driving the **SysAD** and **SysADC** buses, the System interface is in *slave state*.

In master state, the processor asserts the signal **ValidOut*** whenever the **SysAD** and **SysCmd** buses are valid. In slave state, the external agent asserts the signal **ValidIn*** whenever the **SysAD** and **SysCmd** buses are valid and the secondary cache drives the **SysAD** and **SysADC** buses in response to the **ScDCE***, **ScCWE***, and **ScDOE*** signals.

The System interface remains in master state unless one of the following occurs:

- *The external agent requests and is granted the System interface (external arbitration).*
- *The processor issues a read request.*

## External Arbitration

The System interface must be in slave state for the external agent to issue an external request through the System interface. The transition from master state to slave state is arbitrated by the processor using the System interface handshake signals **ExtRqst*** and **Release***. This transition is described by the following procedure:

1. An external agent signals that it wishes to issue an external request by asserting **ExtRqst***.
2. When the processor is ready to accept an external request, it releases the System interface from master to slave state by asserting **Release*** for one cycle.
3. The System interface returns to master state as soon as the issue of the external request is complete.

## Uncompelled Change to Slave State

An *uncompelled* change to slave state is the transition of the System interface from master state to slave state, initiated by the processor when a processor read request is pending. **Release*** is asserted automatically after a read request and an uncompelled change to slave state then occurs. This transition to slave state allows the external agent to return read response data without arbitrating for bus ownership.

Notes

If the secondary cache is enabled and a secondary cache hit occurs, then the uncompelled change to slave state is aborted. If the secondary cache is enabled and a secondary cache miss occurs, the uncompelled change to slave state is delayed until the external agent has disabled the secondary cache outputs, even though **Release*** was issued with the read request.

After an uncompelled change to slave state, the processor returns to master state at the end of the next external request. This can be a read response, or some other type of external request. If the external agent issues some other type of external request while there is a pending read request, the processor performs another uncompelled change to slave state by asserting **Release*** for one cycle.

An external agent must note that the processor has performed an uncompelled change to slave state and begin driving the **SysAD** bus along with the **SysCmd** bus. As long as the System interface is in slave state, the external agent can begin an external request without arbitrating for the System interface; that is, without asserting **ExtRqst***.

Table 11.1 lists the abbreviations and definitions for each of the buses that are used in the timing diagrams that follow.

| Scope | Abbreviation | Meaning |
|---|---|---|
| Global | Unsd | Unused |
| SysAD bus | Addr | Physical address |
| | Data<n> | Data element number n of a block of data |
| SysCmd bus | Cmd | An unspecified System interface command |
| | Read | A processor or external read request command |
| | Write | A processor or external write request command |
| | SINull | A System interface release external null request command |
| | NData | A noncoherent data identifier for a data element other than the last data element |
| | NEOD | A noncoherent data identifier for the last data element |

**Table 11.1 System Interface Requests**

## Processor Request Protocols

Processor read and write request protocols are described in this section. In the timing diagrams, the two closely spaced, wavy vertical lines, such as those shown in Figure 11.4, indicate one or more identical cycles which are not illustrated due to space constraints.

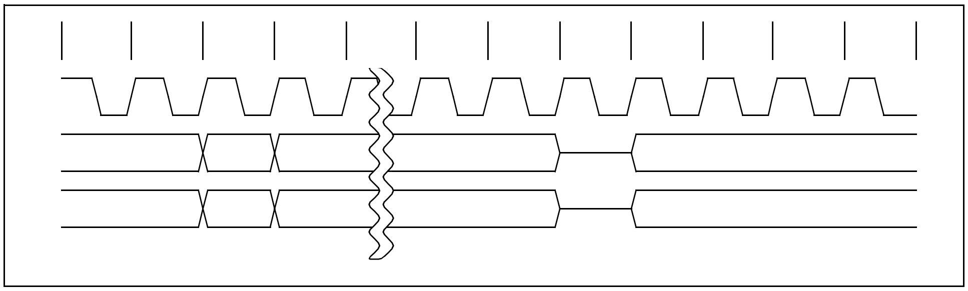


**Figure 11.4 Symbol for Undocumented Cycles**

### Processor Read Request Protocol

The following sequence describes the protocol for doubleword, partial doubleword, word, partial word, and non-secondary cache mode processor read requests. The secondary cache block read request protocol is described later in this section.

The following numbered steps correspond to Figure 11.5:

1. **RdRdy*** is asserted, indicating the external agent is ready to accept a read request.

Notes

2. With the System interface in master state, a processor read request is issued by driving a read command on the **SysCmd** bus and a read address on the **SysAD** bus. The physical address is driven onto **SysAD[35:0]**, and virtual address bits [13:12] are driven onto **SysAD[57:56]**. All other bits are driven to zero.
3. At the same time, the processor asserts **ValidOut*** for one cycle, indicating valid data is present on the **SysCmd** and the **SysAD** buses.
4. Only one processor read request can be pending at a time.
5. The processor makes an uncompelled change to slave state during the issue cycle of the read request. The external agent must not assert the signal **ExtRqst*** for the purposes of returning a read response, but rather must wait for the uncompelled change to slave state. The signal **ExtRqst*** can be asserted before or during a read response to perform an external request other than a read response.
6. The processor releases the **SysCmd** and the **SysAD** buses one **SysClock** after the assertion of **Release***.
7. The external agent drives the **SysCmd** and the **SysAD** buses within two cycles after the assertion of **Release***.

Once in slave state, the external agent can return the requested data through a read response. The read response can return the requested data or, if the requested data could not be successfully retrieved, an indication that the returned data is erroneous. If the returned data is erroneous, the processor takes a bus error exception.

Figure 11.5 illustrates a processor read request, coupled with an uncompelled change to slave state, that occurs as the read request is issued. Timings for the **SysADC** and **SysCmdP** buses are the same as those of the **SysAD** and **SysCmd** buses, respectively.

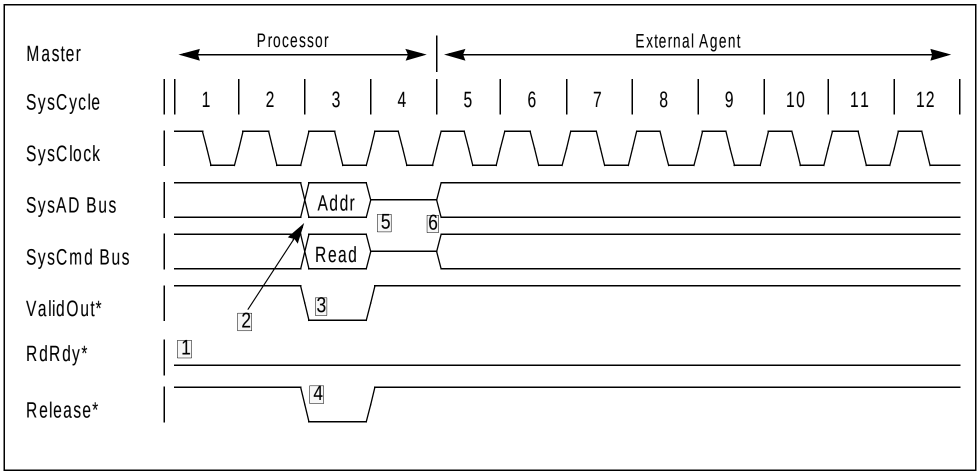


**Figure 11.5 Processor Read Request Protocol**

If a read request is pending while **ExtRqst*** is asserted and **Release*** is asserted for one cycle, it may be unclear if the assertion of **Release*** is in response to **ExtRqst***, or represents an uncompelled change to slave state. If these three conditions exist, the processor accepts the external request as opposed to the read response.

In all other cases, the assertion of **Release*** indicates either an uncompelled change to slave state, or a response to the assertion of **ExtRqst***, whereupon the processor accepts either a read response, or any other external request. If any external request other than a read response is issued, the processor performs another uncompelled change to slave state, asserting **Release***, after processing the external request.

## Processor Write Request Protocol

Processor write requests are issued using one of three protocols:

- *Doubleword, partial doubleword, word, or partial word writes use a non-block write request protocol.*
- *Non-secondary cache block writes use a block write request protocol.*
- *Secondary cache block write request protocol.*

Notes

Processor doubleword write requests are issued with the System interface in master state, as described below in the steps below (Figure 11.6 shows a processor noncoherent single non-block write request cycle):

1. **WrRdy*** is asserted, indicating the external agent is ready to accept a write request.
2. A processor single non-block write request is issued by driving a write command on the **SysCmd** bus and a write address on the **SysAD** bus. The physical address is driven onto **SysAD[35:0]**, and virtual address bits [13:12] are driven onto **SysAD[57:56]**. All other bits are driven to zero.
3. The processor asserts **ValidOut***.
4. The processor drives a data identifier on the **SysCmd** bus and data on the **SysAD** bus.
5. The data identifier associated with the data cycle must contain a last data cycle indication. At the end of the cycle, **ValidOut*** is deasserted.

Timings for the **SysADC** and **SysCmdP** buses are the same as those of the **SysAD** and **SysCmd** buses, respectively.

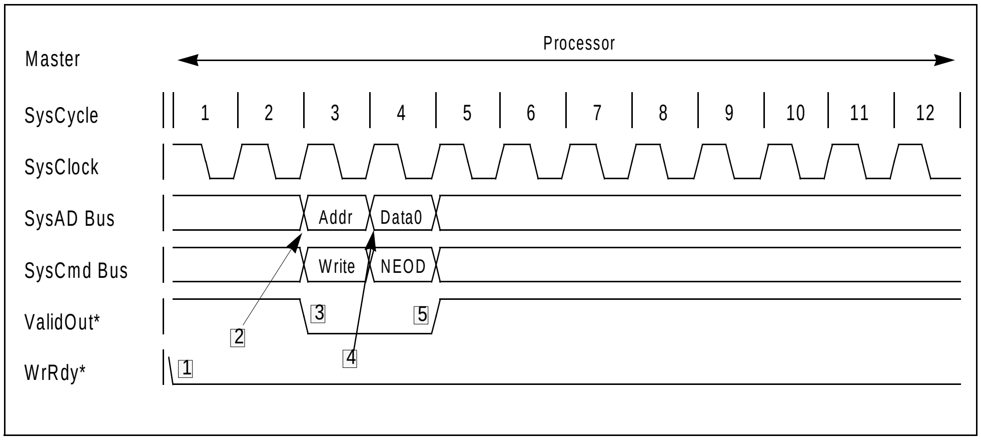


**Figure 11.6 Processor Noncoherent Single Word Write Request Protocol**

Figure 11.7 illustrates a non-secondary cache block write request.

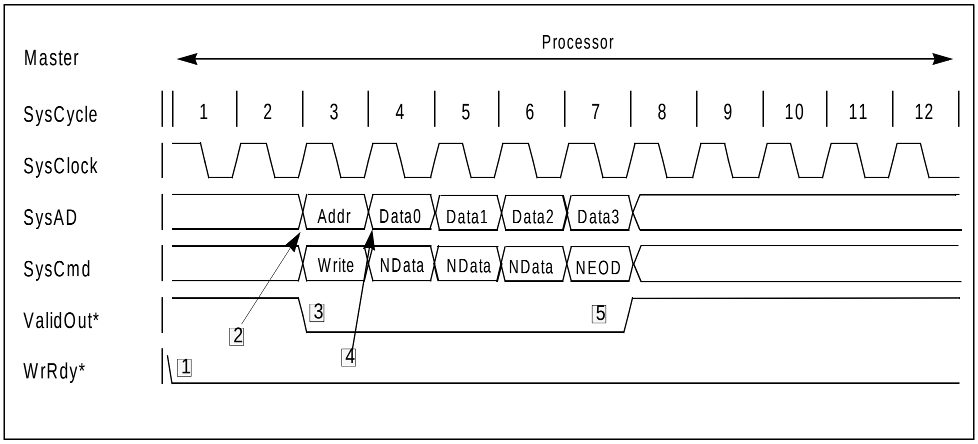


**Figure 11.7 Processor Non-Coherent, Non-Secondary Cache Block Write Request**

### Processor Request Flow Control

The external agent uses **RdRdy*** to control the flow of processor read requests. Figure 11.8 illustrates this flow control, as described in the steps below:

1. The processor samples **RdRdy*** to determine if the external agent is capable of accepting a read request.
2. Read request is issued to the external agent.
3. The external agent deasserts **RdRdy***, indicating it cannot accept additional read requests.
4. The read request issue is stalled because **RdRdy*** was negated two cycles earlier.
5. Read request is again issued to the external agent.

Notes

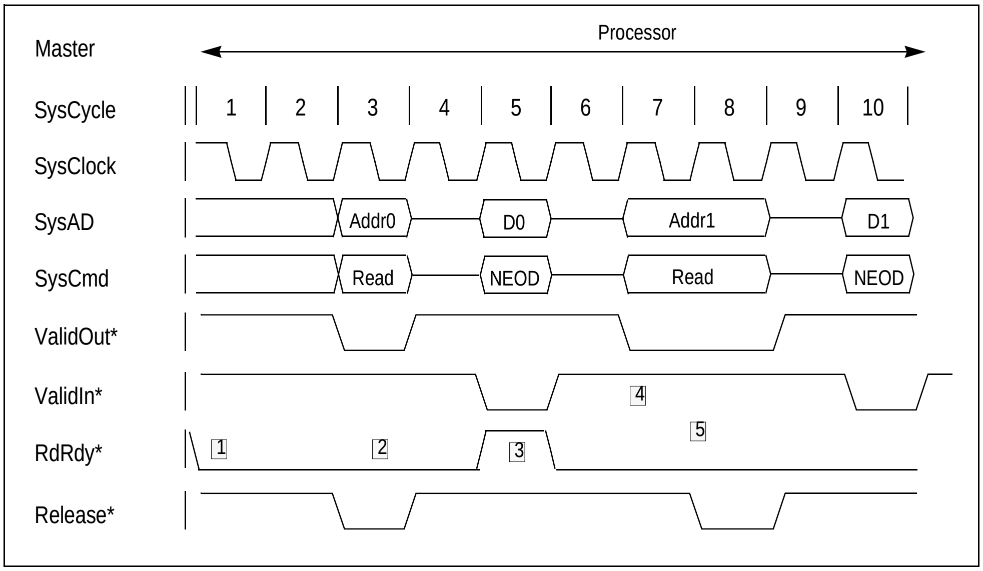


**Figure 11.8 Processor Request Flow Control**

Figure 11.9 illustrates two processor write requests in which the issue of the second is delayed for the assertion of **WrRdy***:

1. **WrRdy*** is asserted, indicating the external agent is ready to accept a write request.
2. The processor asserts **ValidOut***, a write command on the **SysCmd** bus, and a write address on the **SysAD** bus.
3. The second write request is delayed until the **WrRdy*** signal is again asserted.
4. The processor does not complete the issue of a write request until it issues an address cycle in response to the write request for which the signal **WrRdy*** was asserted two cycles earlier.

Timings for the **SysADC** and **SysCmdP** buses are the same as those of the **SysAD** and **SysCmd** buses, respectively.

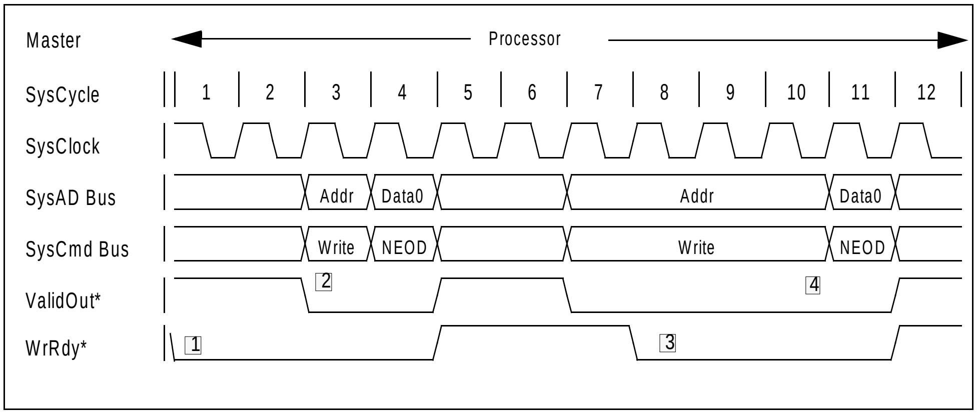


**Figure 11.9 Two Processor Write Requests with Second Write Delayed**

The processor interface requires that **WrRdy*** be asserted two system cycles prior to the issue of a write cycle. An external agent that negates **WrRdy*** immediately upon receiving the write that fills its buffer will suspend any subsequent writes for four system cycles in R4000 non-block write-compatible mode. The processor always inserts at least two unused system cycles after a write address/data pair in order to give the external agent time to suspend the next write.

Figure 11.10 shows back-to-back write cycles in R4000-compatible mode:

1. **WrRdy*** is asserted, indicating the processor can issue a write request.
2. **WrRdy*** remains asserted, indicating the external agent can accept another write request.
3. **WrRdy*** deasserts, indicating the external agent cannot accept another write request, stalling the issue of the next write request.

Notes

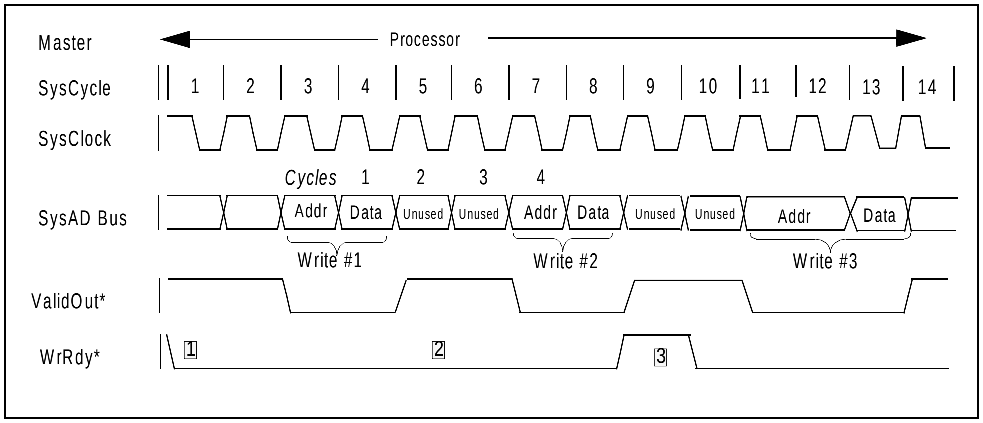


**Figure 11.10 R4000-Compatible Back-to-Back Write Cycle Timing**

An address/data pair every four system cycles is not sufficiently high performance for all applications. For this reason, the R5000 processor provides two protocol options that modify the R4000 back-to-back write protocol to allow an address/data pair every two system cycles. These two protocols are:

- *Write Reissue allows* ***WrRdy**** *to be negated during the address cycle and forces the write cycle to be re-issued.*
- *Pipelined Writes leave the sample point of* ***WrRdy**** *unchanged and require that the external agent accept one more write than dictated by the R4000 protocol.*

The write re-issue protocol is shown in Figure 11.11. Writes issue when **WrRdy*** is asserted both two cycles prior to the address cycle and during the address cycle:

1. **WrRdy*** is asserted, indicating the external agent can accept a write request.
2. **WrRdy*** remains asserted as the write is issued, and the external agent is ready to accept another write request.
3. **WrRdy*** deasserts during the address cycle. This write request is aborted and reissued.
4. **WrRdy*** is asserted, indicating the external agent can accept a write request.
5. **WrRdy*** remains asserted as the write is issued, and the external agent is able to accept another write request.

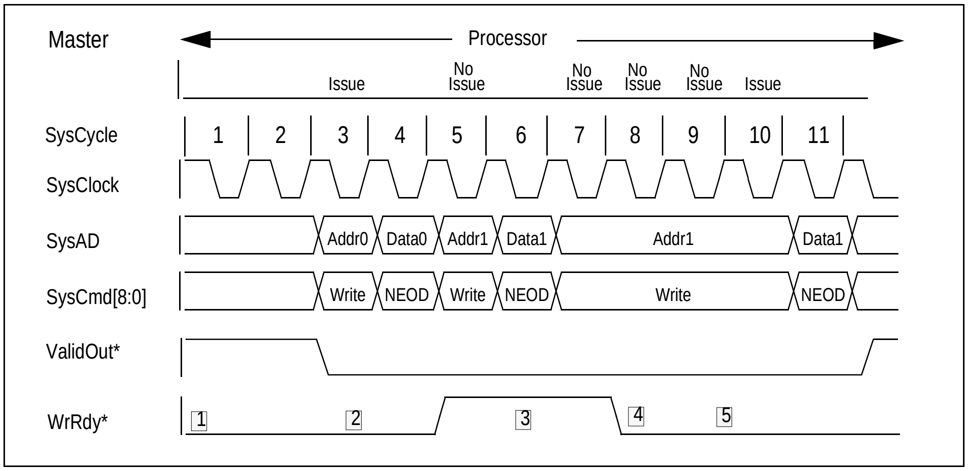


**Figure 11.11 Write Reissue**

The pipelined write protocol is shown in Figure 11.12. Writes issue when **WrRdy*** is asserted two cycles before the address cycle and the external agent is required to accept one more writes after **WrRdy*** is negated:

1. **WrRdy*** is asserted, indicating the external agent can accept a write request.
2. **WrRdy*** remains asserted as the write is issued, and the external agent is able to accept another write request.

Notes

3. **WrRdy*** is deasserted, indicating the external agent cannot accept another write request; it does, however, accept this write.
4. **WrRdy*** is asserted, indicating the external agent can accept a write request.

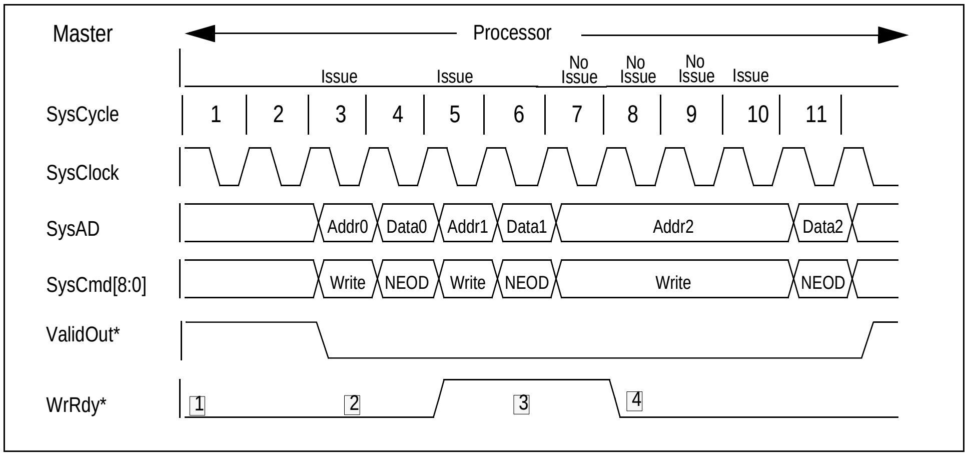


**Figure 11.12 Pipelined Writes**

## External Request Protocols

This section describes *read, null, write*, and *read-response* protocols. External requests can only be issued when the System interface is in slave state. An external agent asserts **ExtRqst*** to arbitrate for the System interface, then waits for the processor to release the System interface to slave state by asserting **Release*** before the external agent issues its request. If the System interface is already in slave state—that is, the processor has previously performed an uncompelled change to slave state—the external agent can begin an external request immediately.

After issuing an external request, the external agent must return the System interface to master state. If the external agent does not have any additional external requests to perform, **ExtRqst*** must be deasserted two cycles after the cycle in which **Release*** was asserted. For a string of external requests, the **ExtRqst*** signal is asserted until the last request cycle, whereupon it is deasserted two cycles after the cycle in which **Release*** was asserted.

The processor continues to handle external requests as long as **ExtRqst*** is asserted; however, the processor cannot release the System interface to slave state for a subsequent external request until it has completed the current request. As long as **ExtRqst*** is asserted, the string of external requests is not interrupted by a processor request.

### External Arbitration Protocol

System interface arbitration uses the signals **ExtRqst*** and **Release*** as described above. Figure 11.13 is a timing diagram of the arbitration protocol, in which slave and master states are shown. The arbitration cycle consists of the following steps:

1. The external agent asserts **ExtRqst*** when it wishes to submit an external request.
2. The processor waits until it is ready to handle an external request, whereupon it asserts **Release*** for one cycle.
3. The processor sets the **SysAD** and **SysCmd** buses to tri-state.
4. The external agent must wait at least two cycles after the assertion of **Release*** before it drives the **SysAD** and **SysCmd** buses.
5. The external agent negates **ExtRqst*** two cycles after the assertion of **Release***, unless the external agent wishes to perform an additional external request.
6. The external agent sets the **SysAD** and the **SysCmd** buses to tri-state at the completion of an external request. The processor can start issuing a processor request one cycle after the external agent sets the bus to tri-state.

Timings for the **SysADC** and **SysCmdP** buses are the same as those of the **SysAD** and **SysCmd** buses, respectively.

Notes

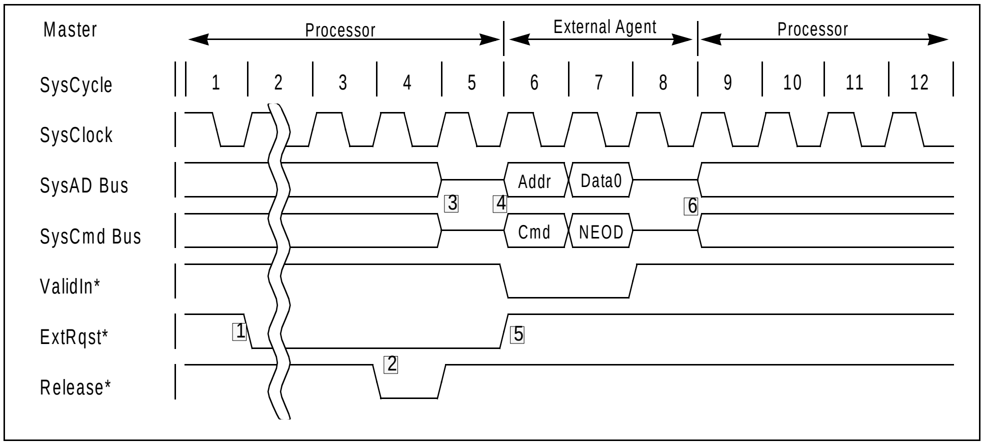


**Figure 11.13 Arbitration Protocol for External Requests**

## External Read Request Protocol

External reads are requests for a word of data from a processor internal resource, such as a register. *However, the R5000 processor does not contain any resources that are readable by an external read request. In response to an external read request, the processor returns undefined data and a data identifier with its* Erroneous Data *bit, SysCmd(5), set.*

Figure 11.14 shows a timing diagram of an external read request and the return of invalid data:

1. An external agent asserts **ExtRqst*** to arbitrate for the System interface.
2. The processor releases the System interface to slave state by asserting **Release*** for one cycle and then deasserting **Release***.
3. After **Release*** is deasserted, the **SysAD** and **SysCmd** buses are set to a tri-state for one cycle.
4. The external agent drives a read request command on the **SysCmd** bus and a read request address on the **SysAD** bus and asserts **ValidIn*** for one cycle.
5. After the address and command are sent, the external agent releases the **SysCmd** and **SysAD** buses by setting them to tri-state and allowing the processor to drive them. The processor returns undefined data to the external agent. The processor accomplishes this by driving a data identifier on the **SysCmd** bus with its Erroneous Data bit SysCmd(5) set, the invalid data on the **SysAD** bus, and asserting **ValidOut*** for one cycle. The data identifier indicates that this is last-data-cycle response data.
6. The System interface is in master state. The processor continues driving the **SysCmd** and **SysAD** buses after the read response is returned.

Timings for the **SysADC** and **SysCmdP** buses are the same as those of the **SysAD** and **SysCmd** buses, respectively.

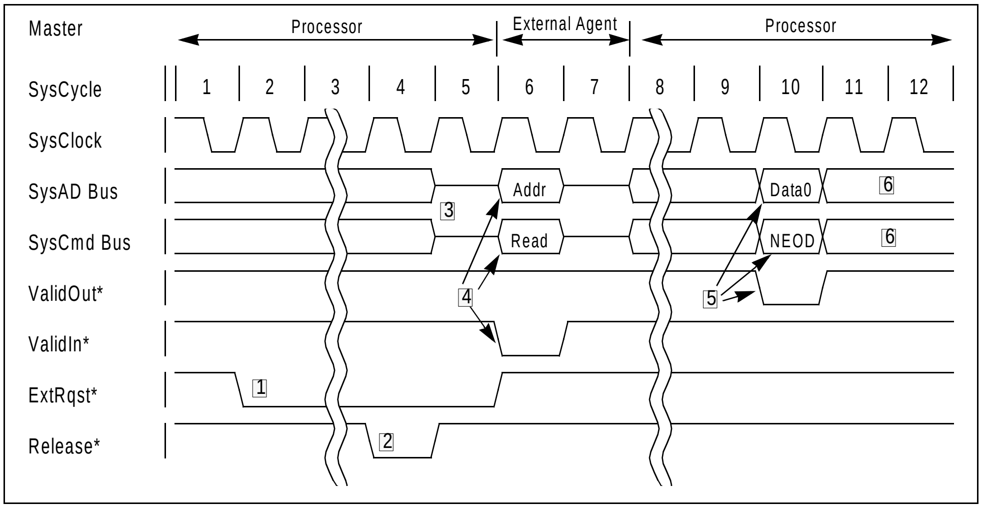


**Figure 11.14 External Read Request, System Interface in Master State**

Notes

## External Null Request Protocol

The processor supports a system interface external null request, which returns the System interface to master state from slave state without otherwise affecting the processor. External null requests require no action from the processor other than to return the System interface to master state.

Figure 11.15 shows a timing diagram of an external null request, which consist of the following steps:

1. The external agent drives a system interface release external null request command on the **SysCmd** bus, and asserts **ValidIn*** for one cycle to return system interface ownership to the processor.
2. The **SysAD** bus is unused (does not contain valid data) during the address cycle associated with an external null request.
3. After the address cycle is issued, the null request is complete.
4. For a *System interface release external null request*, the external agent releases the **SysCmd** and **SysAD** buses, and expects the System interface to return to the master state.

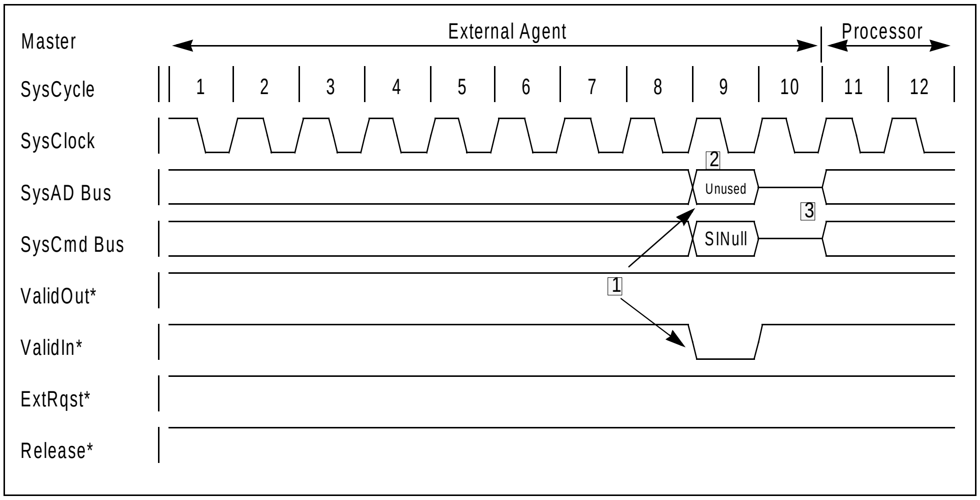


**Figure 11.15 System Interface Release External Null Request**

## External Write Request Protocol

External write requests use a protocol identical to the processor single-word write protocol, except the **ValidIn*** signal is asserted instead of **ValidOut***. *The only processor resource available to an external write request is the* IP *field of the Cause register.*

Figure 11.16 shows a timing diagram of an external write request, which consists of the following steps:

1. The external agent asserts **ExtRqst*** to arbitrate for the System interface.
2. The processor releases the System interface to slave state by asserting **Release***.
3. The external agent drives a write command on the **SysCmd** bus, a write address on the **SysAD** bus, and asserts **ValidIn***.
4. The external agent drives a data identifier on the **SysCmd** bus, data on the **SysAD** bus, and asserts **ValidIn***.
5. The data identifier associated with the data cycle must contain a coherent or noncoherent last data cycle indication.
6. After the data cycle is issued, the write request is complete and the external agent sets the **SysCmd** and **SysAD** buses to a tri-state, allowing the System interface to return to master state. Timings for the **SysADC** and **SysCmdP** buses are the same as those of the **SysAD** and **SysCmd** buses, respectively.

External write requests are only allowed to write a word of data to the processor. Processor behavior in response to an external write request for any data element other than a word is undefined.

Notes

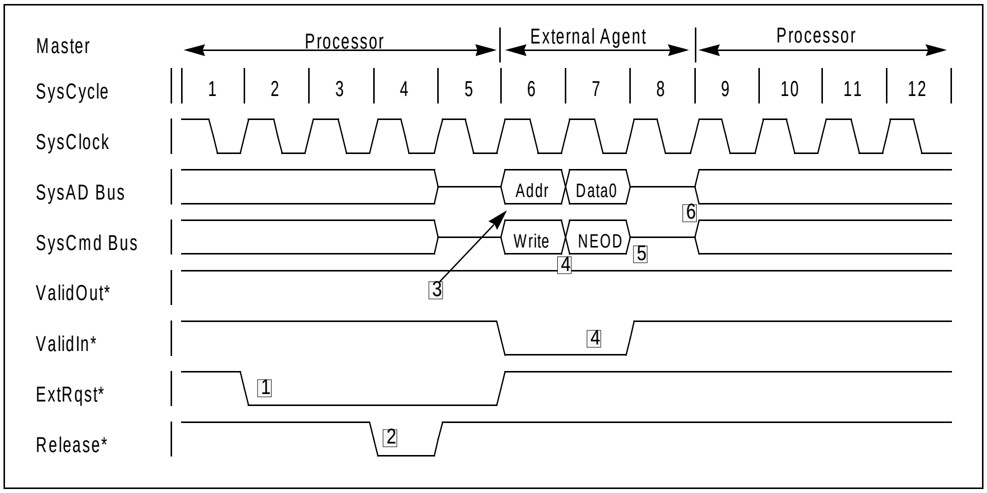


**Figure 11.16 External Write Request, with System Interface Initially a Bus Master**

## Read Response Protocol

An external agent must return data to the processor in response to a processor read request by using a read response protocol. A read response protocol consists of the following steps:

1. The external agent waits for the processor to perform an uncompelled change to slave state.
2. The processor returns the data through a single data cycle or a series of data cycles.
3. After the last data cycle is issued, the read response is complete and the external agent sets the **SysCmd** and **SysAD** buses to a tri-state.
4. The System interface returns to master state.
5. The processor always performs an uncompelled change to slave state after issuing a read request.
6. The data identifier for data cycles must indicate the fact that this data is *response data*.
7. The data identifier associated with the last data cycle must contain a *last data cycle* indication.

For read responses to non-coherent block read requests, the response data does not need to identify the initial cache state. The cache state is automatically assigned as dirty exclusive by the processor.

The data identifier associated with a data cycle can indicate that the data transmitted during that cycle is erroneous; however, an external agent must return a data block of the correct size regardless of the fact that the data may be in error. The processor only checks the error bit for the first doubleword of the block. The remaining error bits for the block are ignored.

Read response data must only be delivered to the processor when a processor read request is pending. The behavior of the processor is undefined when a read response is presented to it and there is no processor read pending.

Figure 11.17 illustrates a processor word read request followed by a word read response. Figure 11.18 illustrates a read response for a processor block read with the System interface already in slave state. Timings for the **SysADC** and **SysCmdP** buses are the same as those of the **SysAD** and **SysCmd** buses, respectively.

Notes

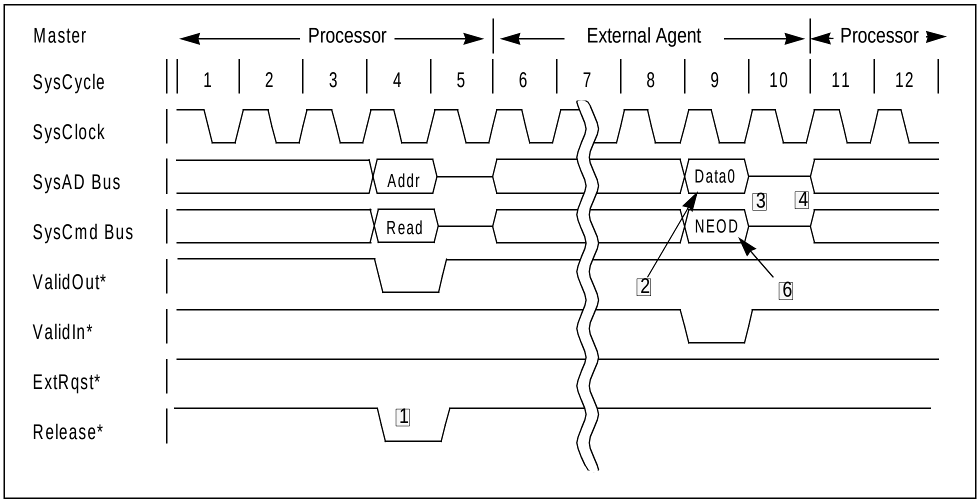


**Figure 11.17 Processor Word Read Request, followed by a Word Read Response**

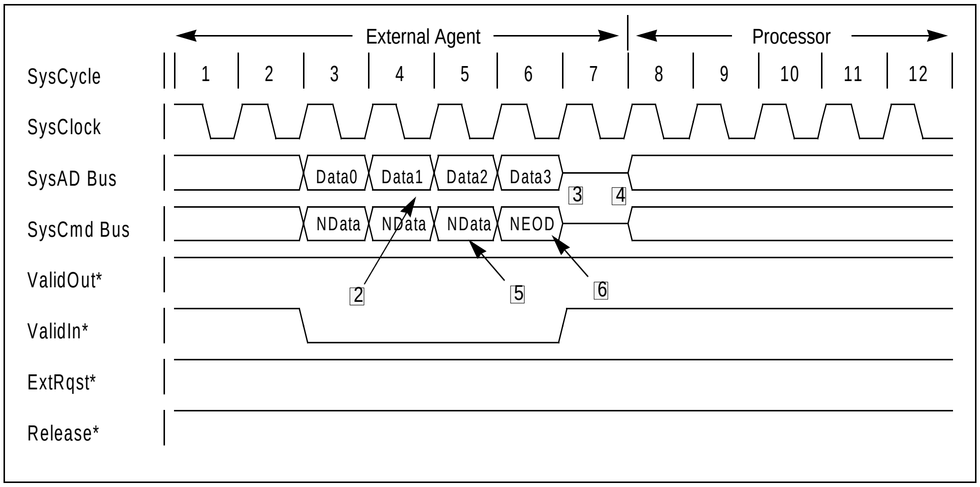


**Figure 11.18 Block Read Response, System Interface already in Slave State**

# Secondary Cache Protocols

## Secondary Cache Read Protocol

There are three possible scenarios which can occur on a secondary cache access.

- *Secondary cache read hit*
- *Secondary cache miss*
- *Secondary cache miss with bus error*

## Secondary Cache Read Hit

Figure 11.19 shows the secondary cache read hit protocol. When a block read request is speculatively issued to both the secondary cache and the external agent, but completed by the secondary cache:

1. The processor issues a block read request and also asserts the **ScTCE***, **ScTDE***, and **ScDCE*** secondary cache control signals. In addition the processor drives the cache index onto **ScLine[15:0]** and the sub-block order doubleword onto **ScWord[1:0]**. Assertion of **ScTCE***, along with **ValidOut*** and **SysCmd**, indicates to the external agent that this is a secondary cache read request. In addition, the assertion of **ScTCE*** initiates a tag RAM probe. The assertion of **ScTDE*** loads the tag portion of the **SysAD** bus into the tag RAM. The **ScValid** signal is asserted to probe for a valid cache tag. The assertion of **ScDCE*** initiates a speculative read of the secondary cache data RAMs.

2. The **ScMatch** signal from the tag RAM is sampled by both the processor and the external agent. Assertion of **ScMatch** indicates a secondary cache tag hit, causing the external agent to abort the memory read. Hence there is no uncompelled change to slave state. The data RAMs now own **SysAD** and supply the first of a 4 doubleword burst in response to the 4-cycle **ScDCE*** burst. The **SysCmd** bus is not driven during the secondary cache read.
3. Ownership of the **SysAD** bus is returned to the processor.

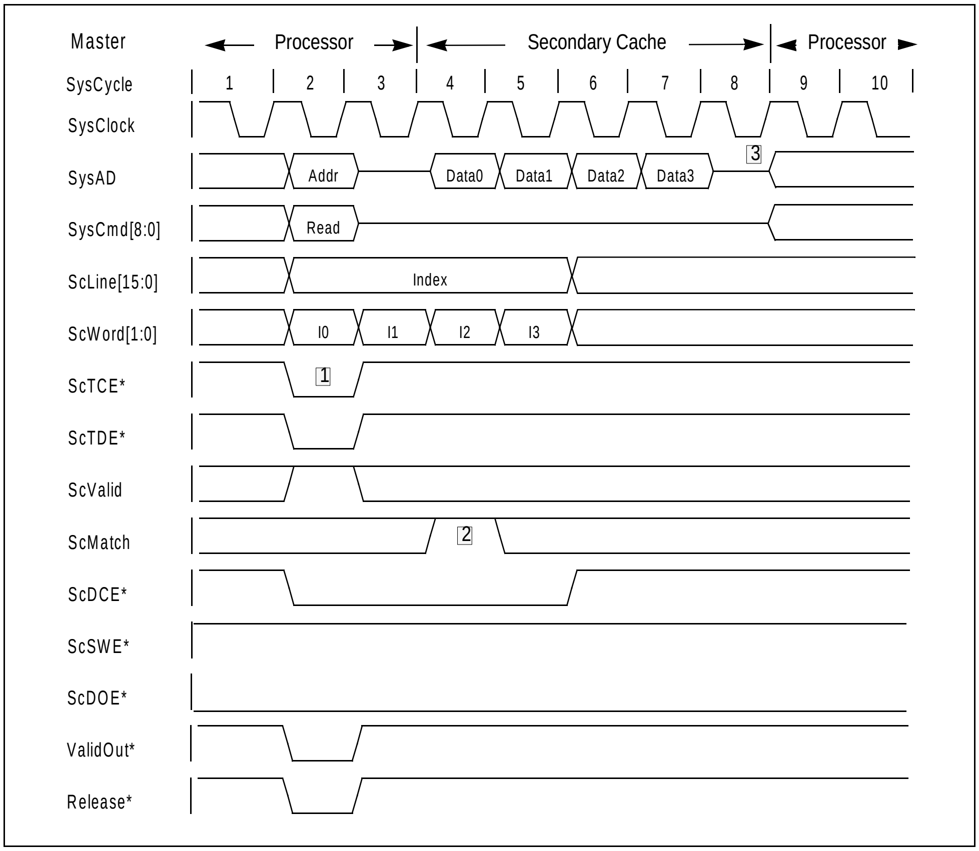


**Figure 11.19 Secondary Cache Read Hit**

## Secondary Cache Read Miss

Figure 11.20 shows the secondary cache read miss protocol when a block read request is speculatively issued to both the secondary cache and the external agent, but is completed by the external agent with a response to both the secondary cache and the processor.

1. The processor issues a clock read request and also asserts the **ScTCE***, **ScTDE***, **ScDCE***, and **ScValid** signals and drives the cache index onto **ScLine[15:0]** and **ScWord[1:0]**.
2. The **ScMatch** signal from the tag RAM is sampled by the processor and external agent. Since the signal is negated, indicating a secondary cache miss, the SysAD data from the secondary cache is invalid.
3. The external agent negates **ScDOE*** to tri-state the data RAM outputs, indicating that it will be supplying the read response. The processor tri-states its **ScWord[1:0]** outputs to allow the external agent to drive them during the read response.
4. The processor asserts **ScCWE*** to prepare the data RAMs for a write of the response data.
5. The external agent supplies the first doubleword of the read response and asserts **ValidIn***. The data is both written into the secondary cache and accepted by the processor. **SysCmd** indicates that data is not erroneous. Note that this response may be delayed additional cycles.
6. The processor asserts **ScTCE*** to write the tag value stored in the tag RAM data input register two cycles after **ValidIn*** is asserted.
7. The external agent asserts **ScDOE*** to indicate that it will supply the last doubleword of the read response in the next cycle.
8. The processor negates **ScDCE*** two cycles after the next assertion of **ScDOE*** in order to complete the secondary cache line fill.

Notes

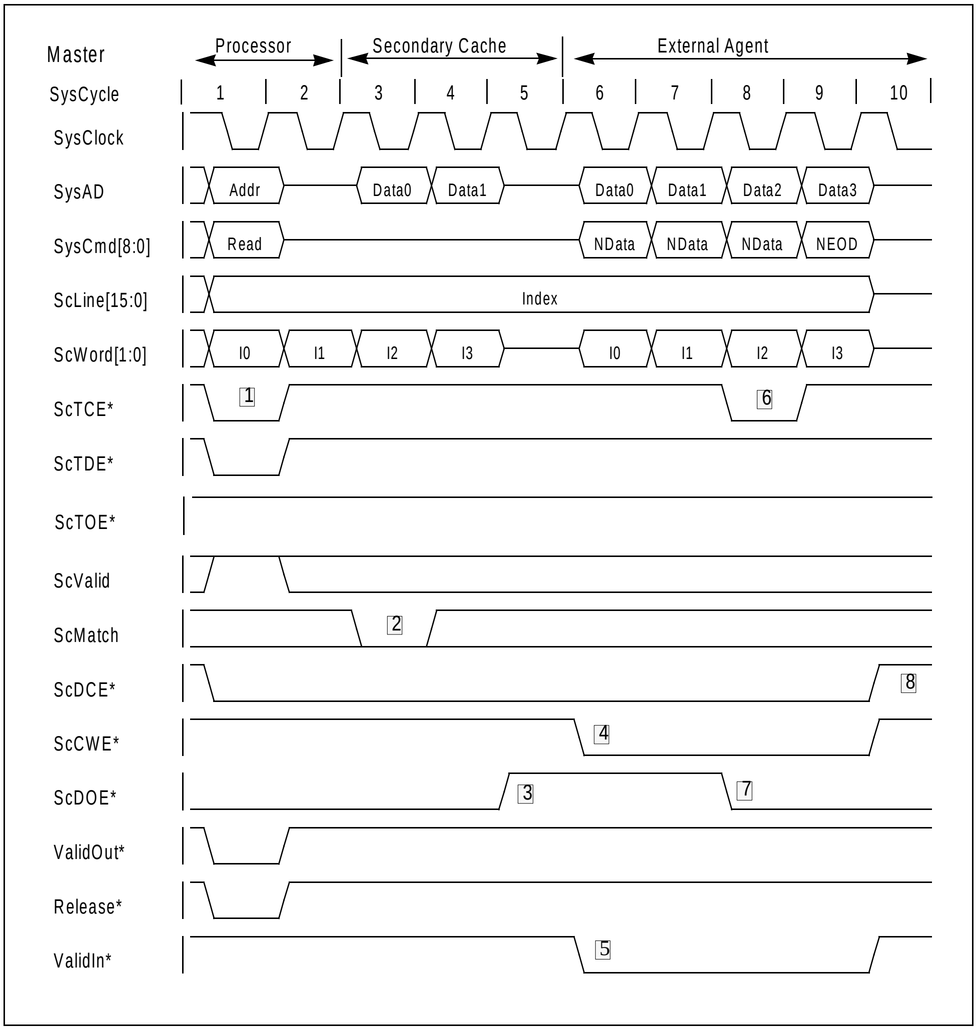


**Figure 11.20 Secondary Cache Read Miss**

### Secondary Cache Read Miss with Bus Error

Figure 11.21 shows a secondary cache read miss with bus error protocol. This protocol is the same as the secondary cache read miss except:

1. The external agent supplies the first doubleword of the read response data with the data error bit set (**SysCmd[5]=1**). Note that the data error bit of **SysCmd** is only checked during the first doubleword of a read response.
2. The processor asserts **ScTCE*** and **SCTDE*** to write the new tag value into the secondary cache tag RAM with **ScValid** negated to invalidate this line.

Notes

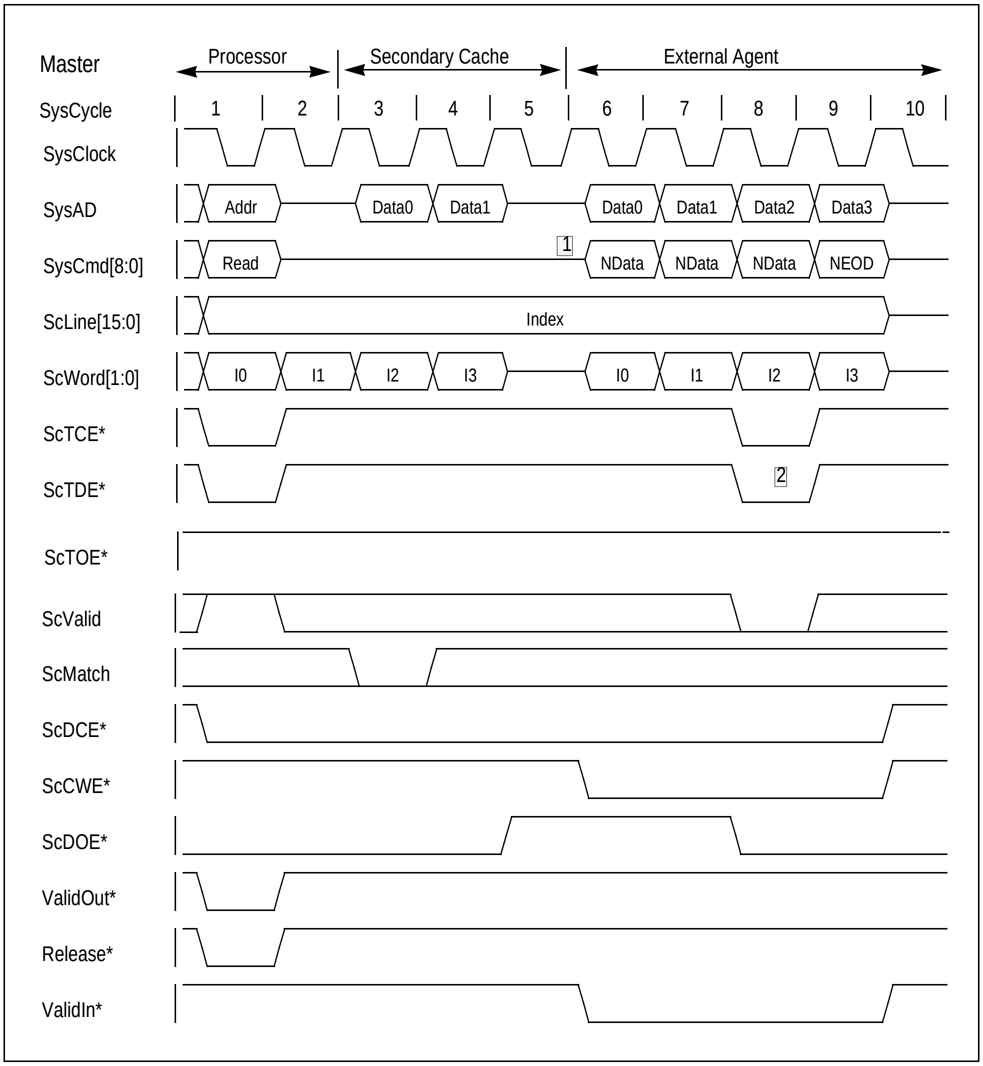


**Figure 11.21 Secondary Cache Read Miss with Bus Error**

## Secondary Cache Write

Figure 11.22 shows a secondary cache read write protocol. For the external agent, this protocol is the same as a non-secondary cache mode block write to the external agent, but the data is also written into the secondary cache.

1. The processor issues a block write and also asserts **ScTCE***, **ScTDE***, and **ScCWE*** in order to write the tag portion of the address on **SysAD** into the secondary cache tag RAM. The processor asserts **ScValid** to set the secondary cache tag to valid.
2. The processor asserts **ScDCE*** to write the block into the secondary cache data RAMs.

Notes

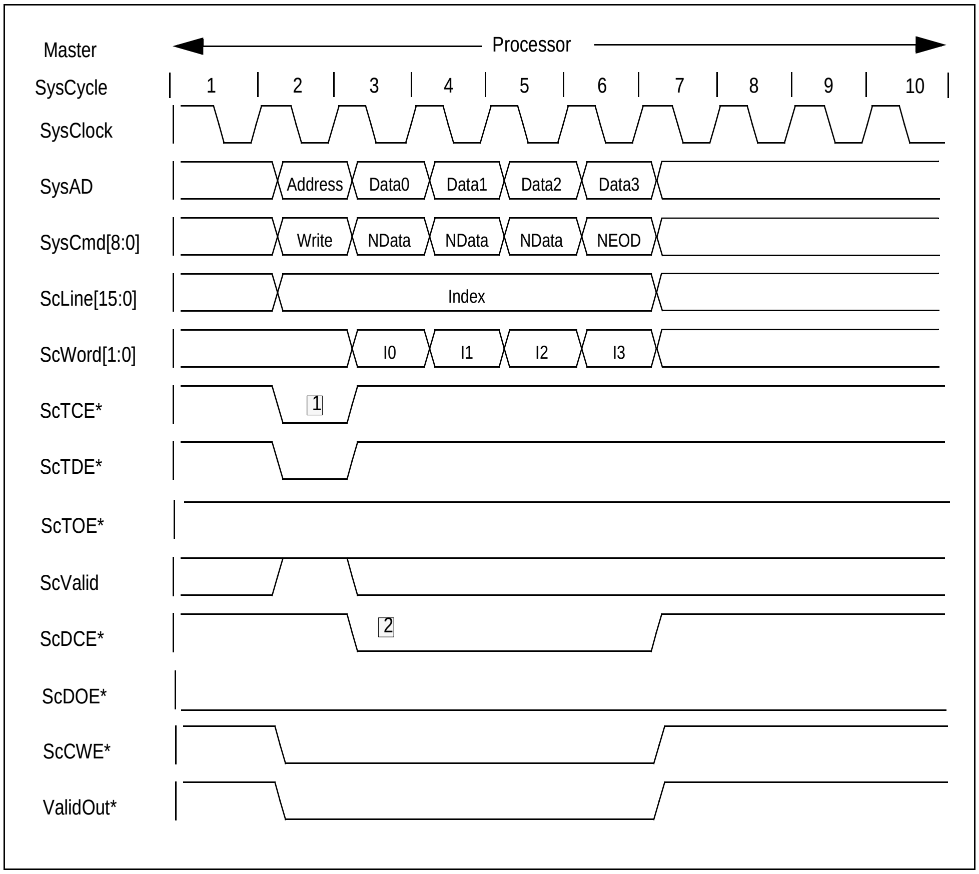


**Figure 11.22 Secondary Cache Write Operation**

### Secondary Cache Line Invalidate

The processor can invalidate either a single line of the secondary cache, or an entire secondary cache block. The invalidate operation is analogous to writing to the Tag RAM and invalidating the line in question. The **ScTCE***, **ScTDE***, and **ScCWE*** signals are driven active in the same clock as the **SysAD** and **ScLine** busses with **ScValid** negated. Invalidates are the only cache operations which may occur back-to-back. Note that **ValidOut*** is not asserted during secondary cache invalidate operations as the external agent does not participate in secondary cache invalidates.

Figure 11.23 shows the secondary cache invalidate protocol. The repeat rate for cache line invalidate instructions is two **SysClock**s. The repeat rate for cache page invalidate is one **SysClock** per line for 128 consecutive **SysClock** cycles.

Notes

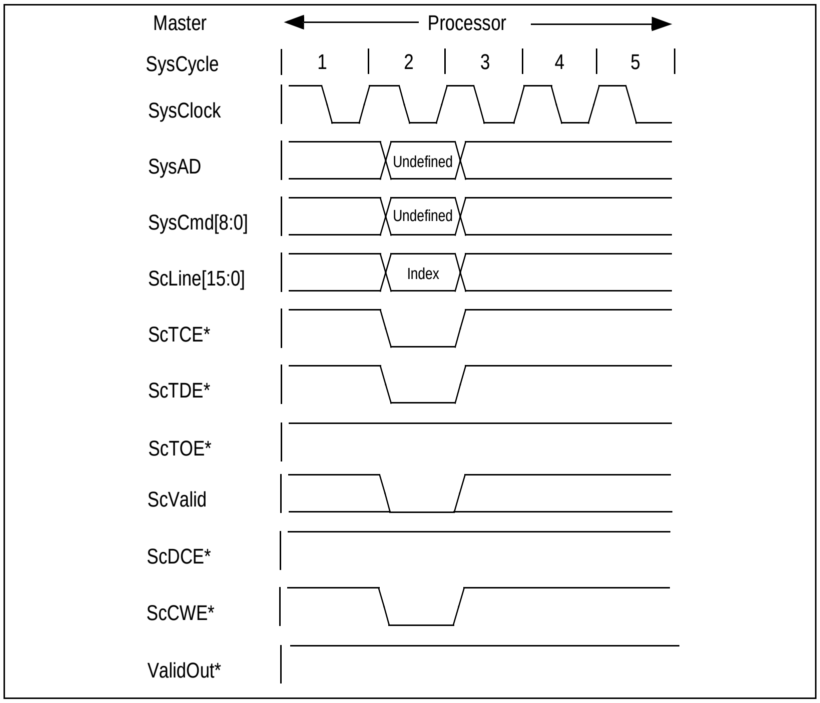


**Figure 11.23 Secondary Cache Line Invalidate**

### Secondary Cache Probe Protocol

The secondary cache probe operation is analogous to a Tag RAM read operation. The **ScTCE*** and **ScTDE*** signals are asserted in the same clock as system address and the secondary cache line index. The processor then tri-states the **SysAD** bus. **ScTOE*** is asserted one clock later and the tag information is driven onto the **SysAD** bus. **ValidOut*** is not asserted during a secondary cache probe operation as the external agent does not participate in secondary cache probes. The Tag RAM bits are driven onto **SysAD [35:19]** and **ScValid**, which are the only **SysAD** signals valid during a probe operation. Figure 11.24 shows a timing diagram of a secondary cache probe protocol.

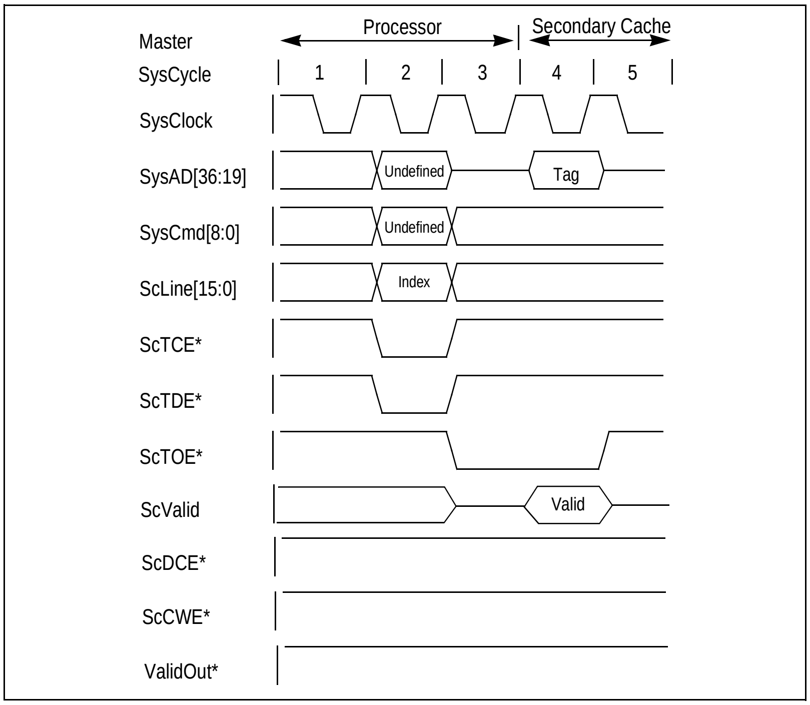


**Figure 11.24 Secondary Cache Probe (Tag RAM Read)**

Notes

### Secondary Cache Block Clear Protocol

In addition to the line invalidate operation, the R5000 processor also has the ability to invalidate the entire secondary cache in one operation. This operation allows the processor to clear the entire column of Tag RAM valid bits. In order to execute this operation the Tag RAM must support a flash clear of the valid bit column. As with the line invalidate operation, **ValidOut*** is not asserted during the block invalidate operation as the external agent does not participate in block clear operations. In addition, the **ScTCE***, **ScTDE***, and **ScWE*** signals need not be asserted. The assertion of **ScCLR*** is all that is necessary for the Tag RAM to perform the requested operation. Figure 11.25 illustrates the secondary cache block clear protocol.

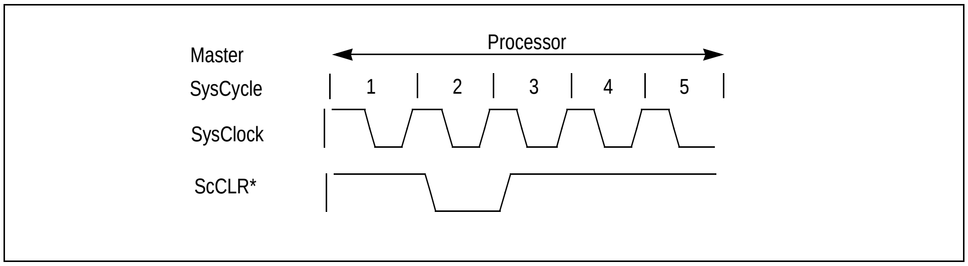


**Figure 11.25 Secondary Cache Block Clear**

## SysADC[7:0] Protocol

The following rules apply to the use of **SysADC[7:0]** during a block read response.

- *Data is checked on only the first doubleword of the transfer. If data is erroneous (**SysCmd[5]**=1), or if the check parity bit is set (**SysCmd[4]**=1), and a parity error is detected on the first doubleword, the primary and secondary cache lines are invalidated and a bus error exception is generated.*
- *On the following three doublewords; The data erroneous bit is ignored. The check parity bit is ignored. Parity for each of the three doublewords is written into the cache, but is not checked until the data is referenced.*
- *For a secondary cache mode read hit cycle; Data erroneous is implicitly OFF. Check parity is implicitly ON, indicating that the secondary cache must implement the SysADC bits.*
- *If a memory error occurs during a block read operation, the **SysADC** bits should be forced to bad parity for all bytes affected by the memory error during the read response. Since the processor performs an early-restart on data cache line fills, setting the **SysCmd[5]** bit on any transfer other than the first doubleword does not cause a bus error. Forcing bad parity will generate a cache error if any of the remaining three doublewords of the transfer are referenced.*

## Data Rate Control

The System interface supports a maximum data rate of one doubleword per cycle. The rate at which data is delivered to the processor can be determined by the external agent—for example, the agent can drive data and assert **ValidIn*** every *n* cycles, instead of every cycle. The processor only accepts cycles as valid when **ValidIn*** is asserted and the **SysCmd** bus contains a data identifier; thereafter, the processor continues to accept data until it receives the data word tagged as the last one.

## Data-Transfer Patterns

A data pattern is a sequence of letters indicating the *data* and *unused* cycles that repeat to provide the appropriate data rate. For example, the data pattern **DDxx** specifies a repeatable data rate of two doublewords every four cycles, with the last two cycles unused. Table 11.2 lists the maximum processor data rate for each of the possible block write modes that may be specified at boot time. In this table, data patterns are specified using the letters **D** and **x**; **D** indicates a data cycle and **x** indicates an unused cycle.

Notes

| Maximum Data Rate | Data Pattern |
|---|---|
| 1 Double/1 SysClock Cycle | DD |
| 2 Doubles/3 SysClock Cycles | DDxDDx |
| 1 Double/2 SysClock Cycles | DDxxDDxx |
| 1 Double/2 SysClock Cycles | DxDx |
| 2 Doubles/5 SysClock Cycles | DDxxxDDxxx |
| 1 Double/3 SysClock Cycles | DDxxxxDDxxxx |
| 1 Double/3 SysClock Cycles | DxxDxx |
| 1 Double/4 SysClock Cycles | DDxxxxxxDDxxxxxx |
| 1 Double/4 SysClock Cycles | DxxxDxxx |

Table 11.2 Transmit Data Rates and Patterns

Figure 11.26 shows a read response in which data is provided to the processor at a rate of two doublewords every three cycles using the data pattern **DDx**.

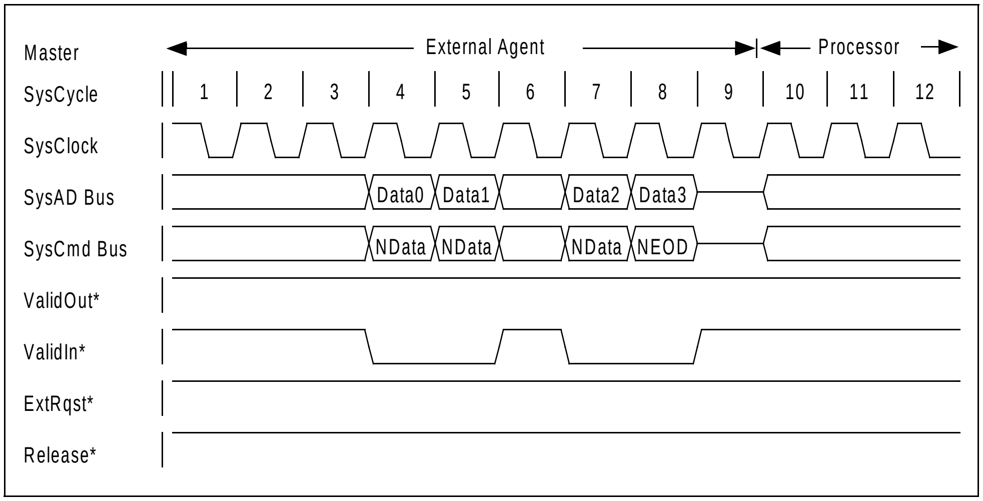


Figure 11.26 Read Response, Reduced Data Rate, System Interface in Slave State

## Independent Transmissions on the SysAD Bus

In most applications, the **SysAD** bus is a point-to-point connection, running from the processor to a bidirectional registered transceiver residing in an external agent. For these applications, the **SysAD** bus has only two possible drivers, the processor or the external agent.

Certain applications may require connection of additional drivers and receivers to the **SysAD** bus, to allow transmissions over the **SysAD** bus that the processor is not involved in. These are called *independent transmissions*. To effect an independent transmission, the external agent must coordinate control of the **SysAD** bus by using arbitration handshake signals and external null requests.

An independent transmission on the **SysAD** bus follows this procedure:

1. The external agent requests mastership of the **SysAD** bus, to issue an external request.
2. The processor releases the System interface to slave state.
3. The external agent then allows the independent transmission to take place on the **SysAD** bus, making sure that **ValidIn*** is not asserted while the transmission is occurring.
4. When the transmission is complete, the external agent must issue a *System interface release external null request* to return the System interface to master state.

Notes

## System Interface Endianness

The endianness of the System interface is programmed at boot time through the boot-time mode control interface and the **BigEndian** pin. The **BigEndian** pin allows the system to change the processor addressing mode without rewriting the mode ROM. If endianness is to be specified via the **BigEndian** pin, program mode ROM bit 8 to zero. If endianness is to be specified by the mode ROM, ground the **BigEndian** pin. Software cannot change the endianness of the System interface and the external system; software can set the reverse endian bit to reverse the interpretation of endianness inside the processor, but the endianness of the System interface remains unchanged.

## System Interface Cycle Time

The processor specifies minimum and maximum cycle counts for various processor transactions and for the processor response time to external requests. Processor requests themselves are constrained by the System interface request protocol, and request cycle counts can be determined by examining the protocol. The following System interface interactions can vary within minimum and maximum cycle counts; minimum and maximum cycle counts are described in the sections that follow:

- *waiting period for the processor to release the System interface to slave state in response to an external request (release latency)*
- *response time for an external request that requires a response (external response latency).*

## Release Latency

*Release latency* is generally defined as the number of cycles the processor can wait to release the System interface to slave state for an external request. When no processor requests are in progress, internal activity can cause the processor to wait some number of cycles before releasing the System interface. Release latency is therefore more specifically defined as the number of cycles that occur between the assertion of **ExtRqst*** and the assertion of **Release***.

There are three categories of release latency:

- *Category 1: when the external request signal is asserted two cycles before the last cycle of a processor request.*
- *Category 2: when the external request signal is not asserted during a processor request or is asserted during the last cycle of a processor request.*
- *Category 3: when the processor makes an uncompelled change to slave state.*

Table 11.3 summarizes the minimum and maximum release latencies for requests that fall into categories 1, 2, and 3. The maximum and minimum cycle count values are subject to change.

| Category | Minimum PCycles | Maximum PCycles |
|---|---|---|
| 1 | 4 | 6 |
| 2 | 4 | 24 |
| 3 | 0 | 0 |

**Table 11.3 Release Latency for External Requests**

## System Interface Commands/Data Identifiers

System interface commands specify the nature and attributes of any System interface request; this specification is made during the address cycle for the request. System interface data identifiers specify the attributes of data transmitted during a System interface data cycle.

The sections that follow describe the syntax, that is, the bitwise encoding of System interface commands and data identifiers. Reserved bits and reserved fields in the command or data identifier should be set to 1 for System interface commands and data identifiers associated with external requests. For System interface commands and data identifiers associated with processor requests, reserved bits and reserved fields in the command and data identifier are undefined.

Notes

### Command and Data Identifier Syntax

System interface commands and data identifiers are encoded in 9 bits and are transmitted on the **SysCmd** bus from the processor to an external agent, or from an external agent to the processor, during address and data cycles. Bit 8 (the most-significant bit) of the **SysCmd** bus determines whether the current content of the **SysCmd** bus is a command or a data identifier and, therefore, whether the current cycle is an address cycle or a data cycle. For System interface commands, **SysCmd(8)** must be set to 0. For System interface data identifiers, **SysCmd(8)** must be set to 1.

### System Interface Command Syntax

This section describes the **SysCmd** bus encoding for System interface commands. Figure 11.27 shows a common encoding used for all System interface commands. **SysCmd(8)** must be set to 0 for all System interface commands. **SysCmd(7:5)** specify the System interface request type which may be read, write, or null. Table 11.4 shows the types of requests encoded by the **SysCmd(7:5)** bits. **SysCmd(4:0)** are specific to each type of request and are defined in each of the following sections.

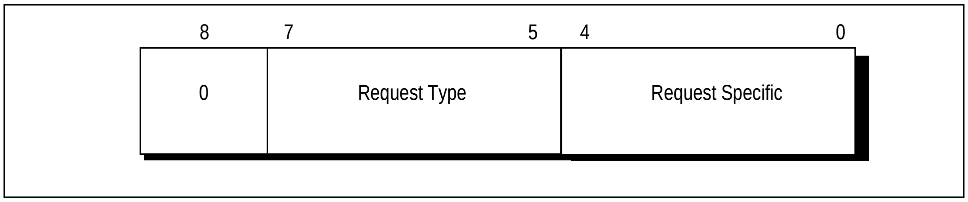


**Figure 11.27 System Interface Command Syntax Bit Definition**

| SysCmd(7:5) | Command |
|---|---|
| 0 | Read Request |
| 1 | Reserved |
| 2 | Write Request |
| 3 | Null Request |
| 4-7 | Reserved |

**Table 11.4 Encoding of SysCmd(7:5) for System Interface Commands**

### Read Requests

Figure 11.28 shows the format of a **SysCmd** read request. Table 11.5 through Table 11.7 list the encodings of **SysCmd(4:0)** for read requests.

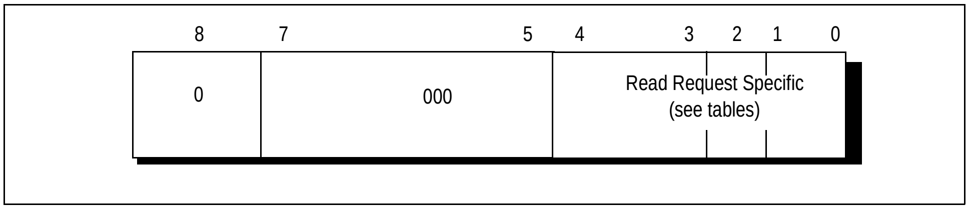


**Figure 11.28 Read Request SysCmd Bus Bit Definition**

| SysCmd(4:3) | Read Attributes |
|---|---|
| 0-1 | Reserved |
| 2 | Noncoherent block read |
| 3 | Doubleword, partial doubleword, word, or partial word |

**Table 11.5 Encoding of SysCmd(4:3) for Read Requests**

Notes

| SysCmd(2) | Link Address Retained Indication |
|---|---|
| 0 | Link address not retained |
| 1 | Link address retained |
| **SysCmd(1:0)** | **Read Block Size** |
| 0 | Reserved |
| 1 | 8 words |
| 2-3 | Reserved |

Table 11.6 Encoding of SysCmd(2:0) for Block Read Request

| SysCmd(2:0) | Read Data Size |
|---|---|
| 0 | 1 byte valid (Byte) |
| 1 | 2 bytes valid (Halfword) |
| 2 | 3 bytes valid (Tribyte) |
| 3 | 4 bytes valid (Word) |
| 4 | 5 bytes valid (Quintibyte) |
| 5 | 6 bytes valid (Sextibyte) |
| 6 | 7 bytes valid (Septibyte) |
| 7 | 8 bytes valid (Doubleword) |

Table 11.7 Read Request of Data Size Encoding of SysCmd(2:0)

### Write Requests

Figure 11.29 shows the format of a **SysCmd** write request. Table 11.8 lists the write attributes encoded in bits **SysCmd(4:3)**. Table 11.9 lists the block write replacement attributes encoded in bits **SysCmd(2:0)**. Table 11.10 lists the write request bit encodings in **SysCmd(2:0)**.

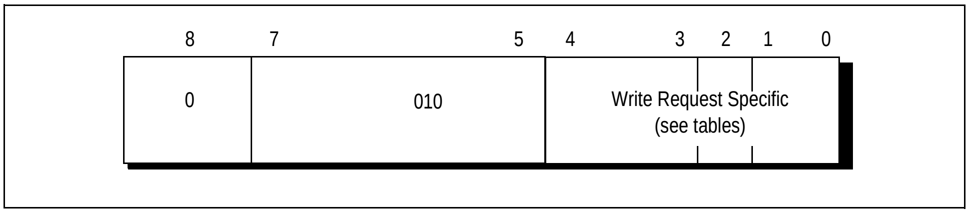


Figure 11.29 Write Request SysCmd Bus Bit Definition

| SysCmd(4:3) | Write Attributes |
|---|---|
| 0 | Reserved |
| 1 | Reserved |
| 2 | Block write |
| 3 | Doubleword, partial doubleword, word, or partial word |

Table 11.8 Write Request Encoding of SysCmd(4:3)

Notes

| SysCmd(2) | Cache Line Replacement Attributes |
|---|---|
| 0 | Cache line replaced |
| 1 | Cache line retained |
| **SysCmd(1:0)** | **Write Block Size** |
| 0 | Reserved |
| 1 | 8 words |
| 2-3 | Reserved |

Table 11.9 Block Write Request Encoding of SysCmd(2:0)

| SysCmd(2:0) | Write Data Size |
|---|---|
| 0 | 1 byte valid (Byte) |
| 1 | 2 bytes valid (Halfword) |
| 2 | 3 bytes valid (Tribyte) |
| 3 | 4 bytes valid (Word) |
| 4 | 5 bytes valid (Quintibyte) |
| 5 | 6 bytes valid (Sextibyte) |
| 6 | 7 bytes valid (Septibyte) |
| 7 | 8 bytes valid (Doubleword) |

Table 11.10 Write Request Data Size Encoding of SysCmd(2:0)

### Null Requests

Figure 11.30 shows the format of a **SysCmd** null request. System interface release external null requests use the null request command. Table 11.11 lists the encodings of **SysCmd(4:3)** for external null requests. **SysCmd(2:0)** are reserved for null requests.

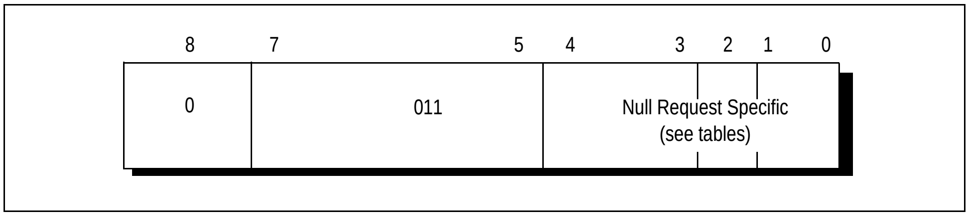


Figure 11.30 Null Request SysCmd Bus Bit Definition

| SysCmd(4:3) | Null Attributes |
|---|---|
| 0 | System Interface release |
| 1-3 | Reserved |

Table 11.11 External Null Request Encoding of SysCmd(4:3)

### System Interface Data Identifier Syntax

This section defines the encoding of the **SysCmd** bus for System interface data identifiers. Figure 11.31 shows a common encoding used for all System interface data identifiers. **SysCmd(8)** must be set to 1 for all System interface data identifiers. **SysCmd(4)** is reserved for processor data identifier. In an external data identifier, **SysCmd(4)** indicates whether or not to check the data and check bits for error.

## Notes

| 8 | 7 | 6 | 5 | 4 | 3 | 2 0 |
|---|---|---|---|---|---|---|
| 1 | Last Data | Resp Data | Err Data | See Note below | Reserved | Cache State |

**Figure 11.31 Data Identifier SysCmd Bus Bit Definition**

### Noncoherent Data

Noncoherent data is defined as:

- *data that is associated with processor block write requests and processor doubleword, partial doubleword, word, or partial word write requests*
- *data that is returned in response to a processor noncoherent block read request or a processor doubleword, partial doubleword, word, or partial word read request*
- *data that is associated with external write requests*
- *data that is returned in response to an external read request*

### Data Identifier Bit Definitions

- ***SysCmd(7)** marks the last data element and **SysCmd(6)** indicates whether or not the data is response data, for both processor and external coherent and noncoherent data identifiers. Response data is data returned in response to a read request.*
- ***SysCmd(5)** indicates whether or not the data element is error free. Erroneous data contains an uncorrectable error and is returned to the processor, forcing a bus error. In the case of a block response, the entire line must be delivered to the processor no matter how minimal the error. Note that the processor only checks **SysCmd[5]** during the first doubleword of a block read response. The processor delivers data with the good data bit deasserted if a primary parity error is detected for a transmitted data item.*
- ***SysCmd(4)** indicates to the processor whether to check the data and check bits for this data element, for both coherent and noncoherent external data identifiers.*
- ***SysCmd(3)** is reserved for external data identifiers.*
- ***SysCmd(4:3)** are reserved for noncoherent processor data identifiers.*
- ***SysCmd(2:0)** are reserved for non-coherent data identifiers.*
- *Table 11.12 lists the encodings of **SysCmd(7:3)** for processor data identifiers. Table 11.13 lists the encodings of **SysCmd(7:3)** for external data identifiers.*

| SysCmd(7) | Last Data Element Indication |
|---|---|
| 0 | Last data element |
| 1 | Not the last data element |
| **SysCmd(6)** | **Response Data Indication** |
| 0 | Data is response data |
| 1 | Data is not response data |
| **SysCmd(5)** | **Good Data Indication** |
| 0 | Data is error free |
| 1 | Data is erroneous |
| **SysCmd(4:3)** | **Reserved** |

**Table 11.12 Processor Data Identifier Encoding of SysCmd(7:3)**

| SysCmd(7) | Last Data Element Indication |
|---|---|
| 0 | Last data element |
| 1 | Not the last data element |
| **SysCmd(6)** | **Response Data Indication** |
| 0 | Data is response data |
| 1 | Data is not response data |
| **SysCmd*(5)*** | **Good Data Indication** |
| 0 | Data is error free |
| 1 | Data is erroneous |
| **SysCmd(4)** | **Data Checking Enable** |
| 0 | Check the data and check bits |
| 1 | Do not check the data and check bits |
| **SysCmd(3)** | **Reserved** |

**Table 11.13 External Data Identifier Encoding of SysCmd(7:3)**

## System Interface Addresses

System interface addresses are full 36-bit physical addresses presented on the least-significant 36 bits (bits 35 through 0) of the **SysAD** bus during address cycles; the remaining bits of the **SysAD** bus are unused during address cycles.

### Addressing Conventions

Addresses associated with doubleword, partial doubleword, word, or partial word transactions and update requests, are aligned for the size of the data element. The system uses the following address conventions:

- *Addresses associated with block requests are aligned to double-word boundaries; that is, the low-order 3 bits of address are 0.*
- *Doubleword requests set the low-order 3 bits of address to 0.*
- *Word requests set the low-order 2 bits of address to 0.*
- *Halfword requests set the low-order bit of address to 0.*
- *Byte, tribyte, quintibyte, sextibyte, and septibyte requests use the byte address.*

### Subblock Ordering

The order in which data is returned in response to a processor block read request is called *subblock ordering*. In subblock ordering, the processor delivers the address of the requested doubleword within the block. An external agent must return the block of data using subblock ordering, starting with the addressed doubleword.

For block write requests, the processor always delivers the address of the doubleword at the beginning of the block; the processor delivers data beginning with the doubleword at the beginning of the block and progresses sequentially through the doublewords that form the block.

A block of data elements (whether bytes, halfwords, words, or doublewords) can be retrieved from storage in two ways: in *sequential order*, or using a *subblock order*. Sequential ordering retrieves the data elements of a block in serial, or sequential, order. Figure 11.32 shows a sequential order in which doubleword 0 is taken first and doubleword 3 is taken last.

Notes

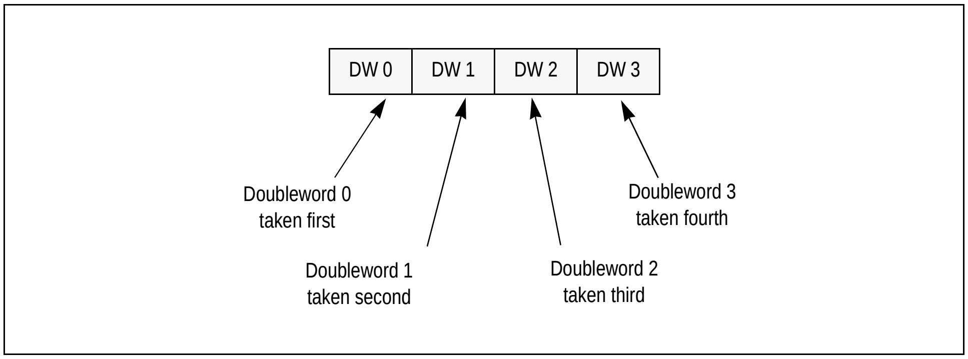


**Figure 11.32 Retrieving a Data Block in Sequential Order**

Subblock ordering allows the system to define the order in which the data elements are retrieved. The smallest data element of a block transfer for the R5000 is a doubleword. Figure 11.33 shows the retrieval of a block of data that consists of four doublewords, in which DW2 is taken first. Using the subblock ordering shown in Figure 11.33, the doubleword at the target address is retrieved first (DW2), followed by the remaining doubleword (DW3) in this quadword.

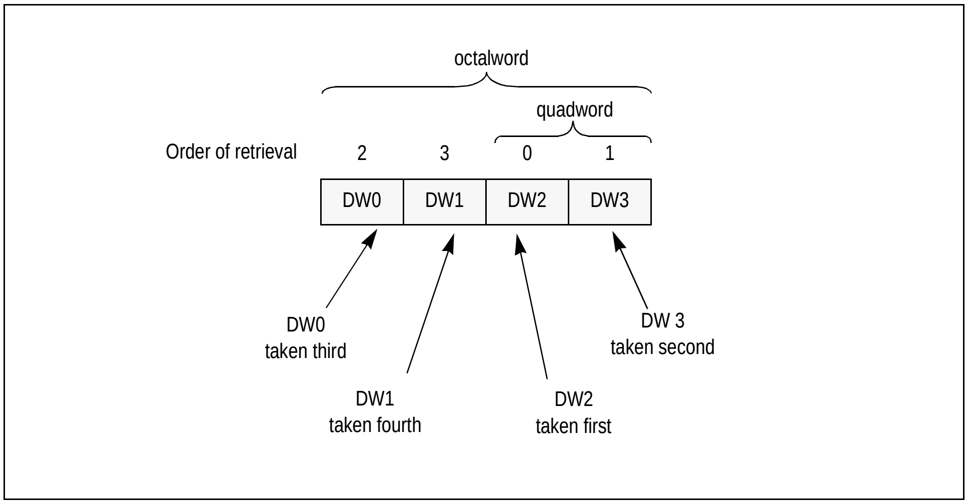


**Figure 11.33 Retrieving Data in a Subblock Order**

It may be easier to understand subblock ordering by taking a look at the method used for generating the address of each doubleword as it is retrieved. The subblock ordering logic generates this address by executing a bit-wise exclusive-OR (XOR) of the starting block address with the output of a binary counter that increments with each doubleword, starting at doubleword zero ($00_2$).

Using this scheme, Table 11.14 through Table 11.16 list the subblock ordering of doublewords for an 8-word block, based on three different starting-block addresses: $10_2$, $11_2$, and $01_2$. The subblock ordering is generated by an XOR of the subblock address (either $10_2$, $11_2$, and $01_2$) with the binary count of the doubleword ($00_2$ through $11_2$). Thus, the third doubleword retrieved from a block of data with a starting address of $10_2$ is found by taking the XOR of address $10_2$ with the binary count of DW2, $10_2$. The result is $00_2$, or DW0.

The remaining tables illustrate this method of subblock ordering, using various address permutations.

Notes

| Cycle | Starting Block Address | Binary Count | Double Word Retrieved |
|---|---|---|---|
| 1 | 10 | 00 | 10 |
| 2 | 10 | 01 | 11 |
| 3 | 10 | 10 | 00 |
| 4 | 10 | 11 | 01 |

Table 11.14 Subblock Ordering Sequence: Address $10_2$

| Cycle | Starting Block Address | Binary Count | Double Word Retrieved |
|---|---|---|---|
| 1 | 11 | 00 | 11 |
| 2 | 11 | 01 | 10 |
| 3 | 11 | 10 | 01 |
| 4 | 11 | 11 | 00 |

Table 11.15 Subblock Ordering Sequence: Address $11_2$

| Cycle | Starting Block Address | Binary Count | Double Word Retrieved |
|---|---|---|---|
| 1 | 01 | 00 | 01 |
| 2 | 01 | 01 | 00 |
| 3 | 01 | 10 | 11 |
| 4 | 01 | 11 | 10 |

Table 11.16 Subblock Ordering Sequence: Address $01_2$

### Valid Byte Lanes During Partial-Word Transfers

During data cycles, the valid byte lines depend upon the position of the data with respect to the aligned doubleword (this may be a byte, halfword, tribyte, quadbyte/word, quintibyte, sextibyte, septibyte, or an octalbyte/doubleword). For example, in little-endian mode, on a byte request where the address modulo 8 is 0, **SysAD(7:0)** are valid during the data cycles. Table 11.17 lists the byte lanes used for partial-word transfers for both big and little endian.

## Notes

| # Bytes<br>SysCmd[2:0] | Address<br>Mod 8 | SysAD byte lanes used (Big Endian) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 63:56 | 55:48 | 47:40 | 39:32 | 31:24 | 23:16 | 15:8 | 7:0 |
| 1<br>(000) | 0 | X | | | | | | | |
| | 1 | | X | | | | | | |
| | 2 | | | X | | | | | |
| | 3 | | | | X | | | | |
| | 4 | | | | | X | | | |
| | 5 | | | | | | X | | |
| | 6 | | | | | | | X | |
| | 7 | | | | | | | | X |
| 2<br>(001) | 0 | X | X | | | | | | |
| | 2 | | | X | X | | | | |
| | 4 | | | | | X | X | | |
| | 6 | | | | | | | X | X |
| 3<br>(010) | 0 | X | X | X | | | | | |
| | 1 | | X | X | X | | | | |
| | 4 | | | | | X | X | X | |
| | 5 | | | | | | X | X | X |
| 4<br>(011) | 0 | X | X | X | X | | | | |
| | 4 | | | | | X | X | X | X |
| 5<br>(100) | 0 | X | X | X | X | X | | | |
| | 3 | | | | X | X | X | X | X |
| 6<br>(101) | 0 | X | X | X | X | X | X | | |
| | 2 | | | X | X | X | X | X | X |
| 7<br>(110) | 0 | X | X | X | X | X | X | X | |
| | 1 | | X | X | X | X | X | X | X |
| 8 (111) | 0 | X | X | X | X | X | X | X | X |
| | | 7:0 | 15:8 | 23:16 | 31:24 | 39:32 | 47:40 | 55:48 | 63:56 |
| | | SysAD byte lanes used (Little Endian) | | | | | | | |

Table 11.17 Partial-Word Transfer Byte Lane Usage

### Processor Internal Address Map

External reads and writes provide access to processor internal resources that may be of interest to an external agent. The processor decodes bits **SysAD(6:4)** of the address associated with an external read or write request to determine which processor internal resource is the target. However, the processor does not contain any resources that are *readable* through an external read request. Therefore, in response to an external read request the processor returns undefined data and a data identifier with its *Erroneous Data* bit, **SysCmd(5)**, set. The *Interrupt* register is the only processor internal resource available for *write* access by an external request. The *Interrupt* register is accessed by an external write request with an address of $000_2$ on bits 6:4 of the **SysAD** bus.

## Notes

# Interrupts

Notes

## Introduction

The processor takes an exception on any of the following interrupts:

- *six hardware interrupts*
- *one internal timer interrupt*
- *two software interrupts*
- *one nonmaskable interrupt.*

This section describes the hardware and nonmaskable interrupts. The six CPU hardware interrupts can be caused by either an external write request to the R5000, or through dedicated interrupt pins. These pins are latched into an internal register by the rising edge of **SysClock**. The nonmaskable interrupt is caused either by an external write request to the R5000 or by a dedicated pin in the R5000. This pin is latched into an internal register by the rising edge of **SysClock**.

## Asserting Interrupts

External writes to the CPU are directed to various internal resources, based on an internal address map of the processor. When **SysAD[6:4]** = 0, an external write to any address writes to an architecturally transparent register called the *Interrupt* register. This register is available for external write cycles, but not for external reads.

During a data cycle, **SysAD[22:16]** are the write enables for the seven individual *Interrupt* register bits and **SysAD[6:0]** are the values to be written into these bits. This allows any subset of the *Interrupt* register to be set or cleared with a single write request. Figure 12.1 shows the mechanics of an external write to the *Interrupt* register.

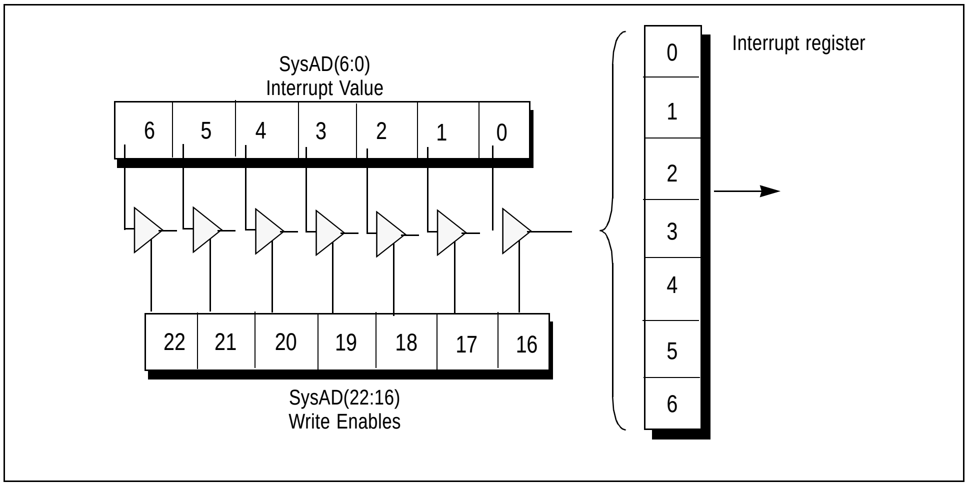


**Figure 12.1 Interrupt Register Bits and Enables**

Figure 12.2 shows how the interrupts are readable through the Cause register:

- *Bit 5 of the Interrupt register is OR'ed with the **Int*[5]** pin and then multiplexed with the **TimerInterrupt** signal. The result is directly readable as bit 15 of the Cause register.*
- *Bits 4:0 of the Interrupt register are bit-wise ORed with the current value of interrupt pins **Int*[4:0]**. The result is directly readable as bits 14:10 of the Cause register.*

Notes

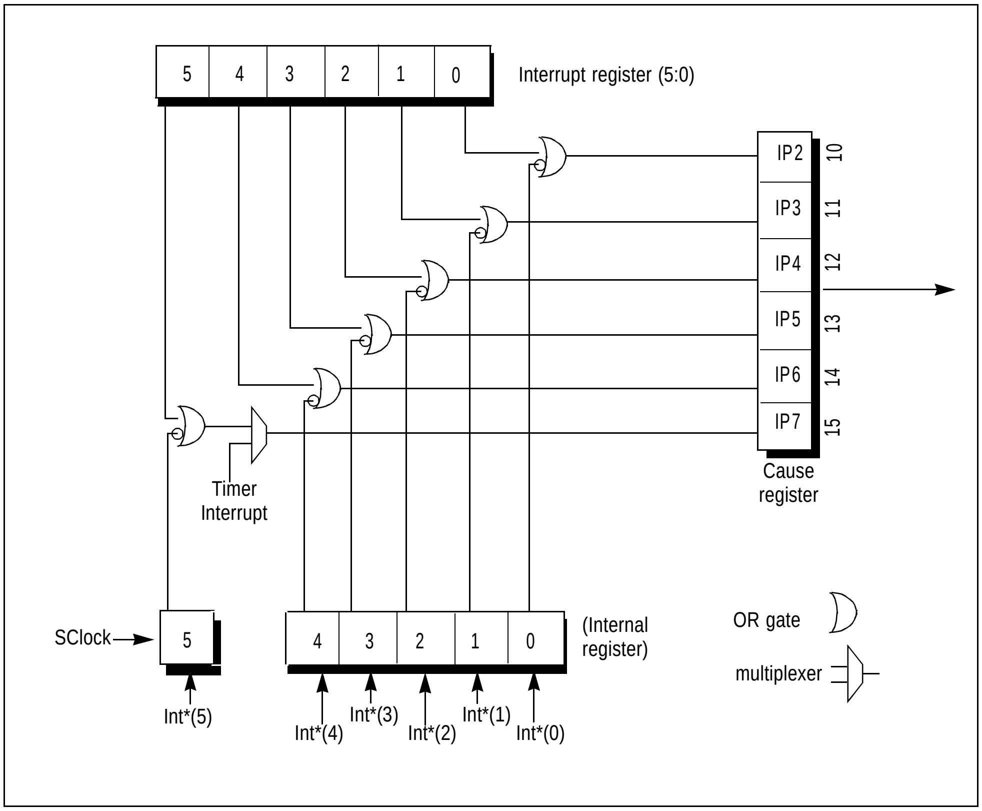


**Figure 12.2 R5000 Interrupt Signals**

Figure 12.3 shows the internal derivation of the **NMI** signal for the R5000 processor. The **NMI*** pin is latched by the rising edge of **SysClock**. Bit 6 of the *Interrupt* register is then ORed with the inverted value of **NMI*** to form the nonmaskable interrupt. Only the falling edge of the latched signal will cause the NMI.

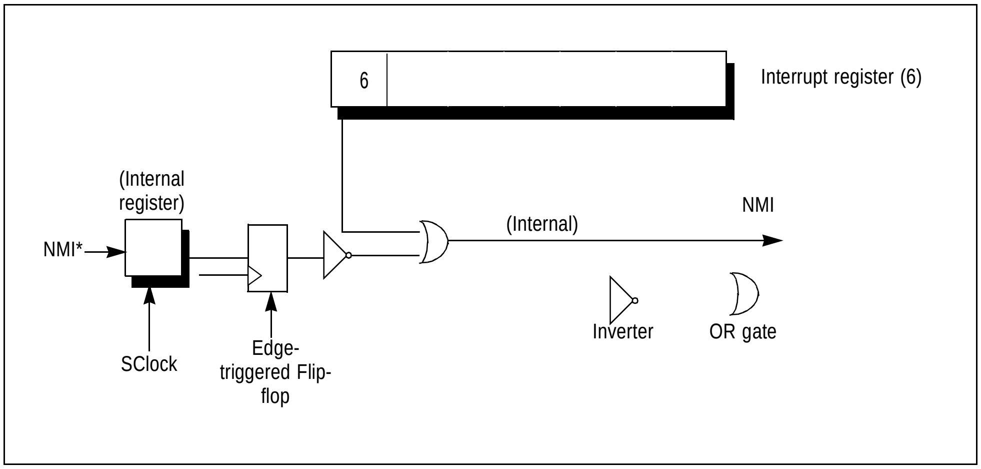


**Figure 12.3 R5000 Nonmaskable Interrupt Signal**

Figure 12.4 shows the masking of the R5000 interrupt signal:

- *Cause register bits 15:8 (IP7-IP0) are AND-ORed with Status register interrupt mask bits 15:8 (IM7-IM0) to mask individual interrupts.*
- *Status register bit 0 is a global Interrupt Enable (IE). It is ANDed with the output of the AND-OR logic to produce the R5000 interrupt signal.*

## Notes

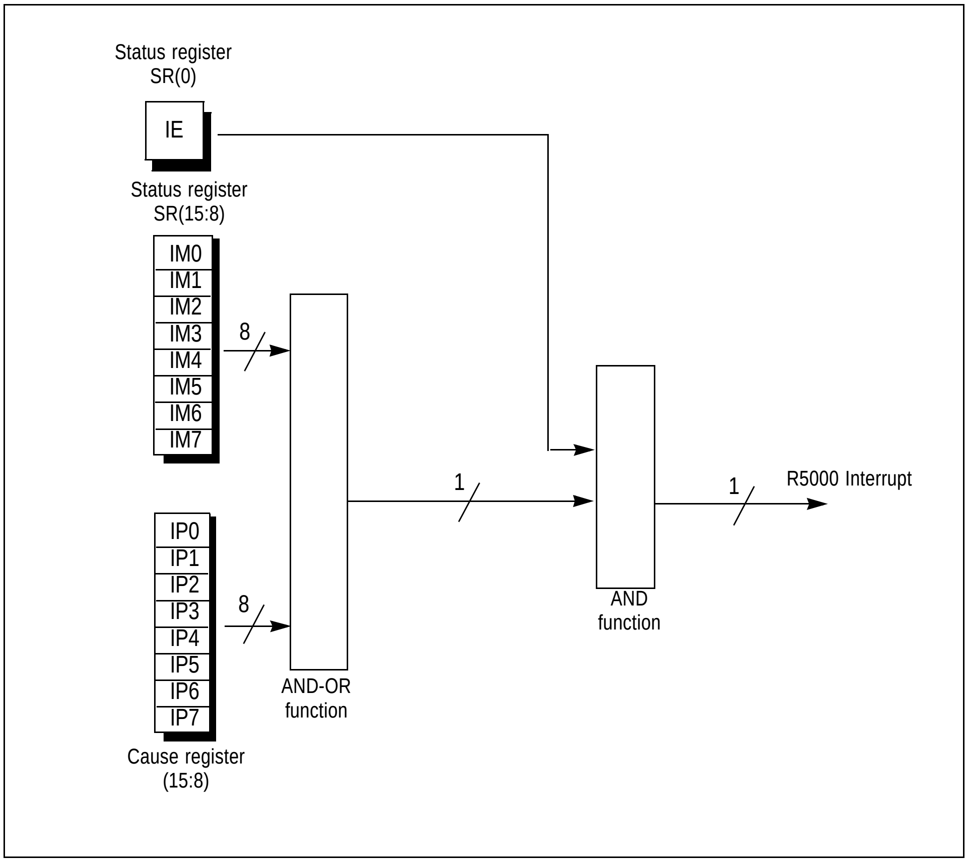


**Figure 12.4 Masking of the R5000 Interrupt**

Notes

# Error Checking

Notes

## Introduction

Two major types of data errors can occur in data transmission:

- *hard errors, which are permanent, arise from broken interconnects, internal shorts, or open leads*
- *soft errors, which are transient, are caused by system noise, power surges, and alpha particles.*

Hard errors must be corrected by physical repair of the damaged equipment and restoration of data from backup. Soft errors can be corrected by using error checking and correcting codes.

The processor verifies data correctness by using parity as it passes data from the System interface to/from the primary caches. By appending a bit to the end of an item of data—called a *parity bit*—single-bit errors can be detected; however, these errors cannot be corrected. Parity generation and checking allows single-bit error detection, but it does not indicate which bit is in error. There are two types of parity:

- *Odd Parity adds 1 to the data, if the data has an even number of 1s, making the total number of 1s odd (including the parity bit).*
- *Even Parity adds 1 to the data, if the data has an odd number of 1s, making the total number of 1s even (including the parity bit).*

## Parity Generation and Checking at the System Interface

The System interface command bus has a single parity bit, **SysCmdP**, that provides *even parity* over the 9 bits of this bus. The **SysCmdP** parity bit is not generated when the system interface is in master state and is not checked when the System interface is in slave state. This signal is defined to maintain R4000 compatibility and is not functional in the R5000.

The processor generates parity bits for doubleword, word, or partial-word data transmitted to the System interface. As it checks for data correctness, the processor passes data check bits from the primary cache, directly without changing the bits, to the System interface.

The processor does not check data received from the System interface for external writes. By setting the **SysCmd[4]** bit in the data identifier, it is possible to prevent the processor from checking read-response data from the System interface. The processor does not check addresses received from the System interface and does not generate check bits for addresses transmitted to the System interface.

The processor does not contain a data corrector; instead, the processor takes a cache error exception when it detects an error based on data check bits. Software is responsible for error handling.

## Summary of Parity Generation and Checking

Parity generation and checking operations are summarized in Table 13.1.

## Notes

| Bus | Uncached Load | Uncached Store | Primary Cache Load from System Interface | Primary Cache Write to System Interface | Cache Instruction | External Read | External Write |
|---|---|---|---|---|---|---|---|
| Processor Data | From system | Not checked | From system interface unchanged | Checked; Trap on error | Check on cache write-back; Trap on error | NA | NA |
| System Address, Command, and Check bits; Transmit | Not Generated | Not Generated | Not Generated | Not Generated | Not Generated | Not Generated | Not Generated |
| System Address, Command, and Check Bits; Receive | Not Checked | Not Checked | Not Checked | Not Checked | Not Checked | Not Checked | Not Checked |
| System Interface Data | Checked, Trap on error | From Processor | Checked, Trap on error | From primary cache | From primary cache | From Processor | Not Checked |
| System Interface Data Check Bits | Checked, Trap on error | Generated | Checked, Trap on error | From primary cache | From primary cache | Generated | Not Checked |

**Table 13.1 Parity Generation and Checking Operations**

# Initialization Interface

Notes

## Reset Signals

The R5000 processor has the following three types of resets:

- *Power-On Reset: starts when the power supply is turned on and completely reinitializes the internal state machines of the processor without saving any state information.*
- *Cold Reset: restarts all clocks, but the power supply remains stable. A cold reset completely reinitializes the internal state machines of the processor without saving any state information.*
- *Warm Reset: restarts the processor, but does not affect clocks. A warm reset preserves the processor internal state.*

The Initialization interface is a serial interface that operates at the frequency of the **SysClock** divided by 256: (**SysClock**/256). This low-frequency operation allows the initialization information to be stored in a low-cost ROM device. This section describes the following reset and control signals:

- ***VccOk:*** *When asserted,* ***VccOk*** *indicates to the processor that the Vcc Min power supply (Vcc) has been above 3.135 volts for more than 100 milliseconds (ms) and is expected to remain stable. The assertion of* ***VccOk*** *initiates the reading of the boot-time mode control serial stream (described in "Initialization Sequence" on page 14-3).*
- ***ColdReset*:*** *The* ***ColdReset**** *signal must be asserted (low) for either a power-on reset or a cold reset.* ***ColdReset**** *must be deasserted synchronously with* ***SysClock****.*
- ***Reset*:*** *the* ***Reset**** *signal must be asserted for any reset sequence. It can be asserted synchronously or asynchronously for a cold reset, or synchronously to initiate a warm reset.* ***Reset**** *must be deasserted synchronously with* ***SysClock****.*
- ***ModeIn****: Serial boot mode data in.*
- ***ModeClock****: Serial boot mode data clock, at the* ***SysClock*** *frequency divided by 256 (****SysClock****/ 256).*

## Power-on Reset

Figure 14.1 shows the power-on system reset timing diagram. The sequence for power-on reset is:

1. Power-on reset applies a stable Vcc of at least Vcc Min volts to the processor. It also supplies a stable, continuous system clock at the processor operational frequency.
2. After at least 100 ms of stable Vcc and **SysClock**, the **VccOk** signal is asserted to the processor. The assertion of **VccOk** initializes the processor operating parameters. After the mode bits have been read in, the processor allows its internal phase locked loops to lock, stabilizing the processor internal clock, **PClock.** Note that JTAG is not implemented; **JTCK** must be tied low at the rising edge of **VccOk** for the processor to properly reset.
3. **ColdReset*** is asserted for at least 64K ($2^{16}$) **SysClock** cycles after the assertion of **VCCOk**. Once the processor reads the boot-time mode control serial data stream, **ColdReset*** can be deasserted. **ColdReset*** must be deasserted synchronously with **SysClock**.
4. After **ColdReset*** is deasserted synchronously, **Reset*** is deasserted to allow the processor to begin running. (**Reset*** must be held asserted for at least 64 **SysClock** cycles after the deassertion of **ColdReset***.) **Reset*** must be deasserted synchronously with **SysClock**.
5. **ColdReset*** must be asserted when **VccOk** asserts. The behavior of the processor is undefined if **VccOk** asserts while **ColdReset*** is deasserted.

Notes

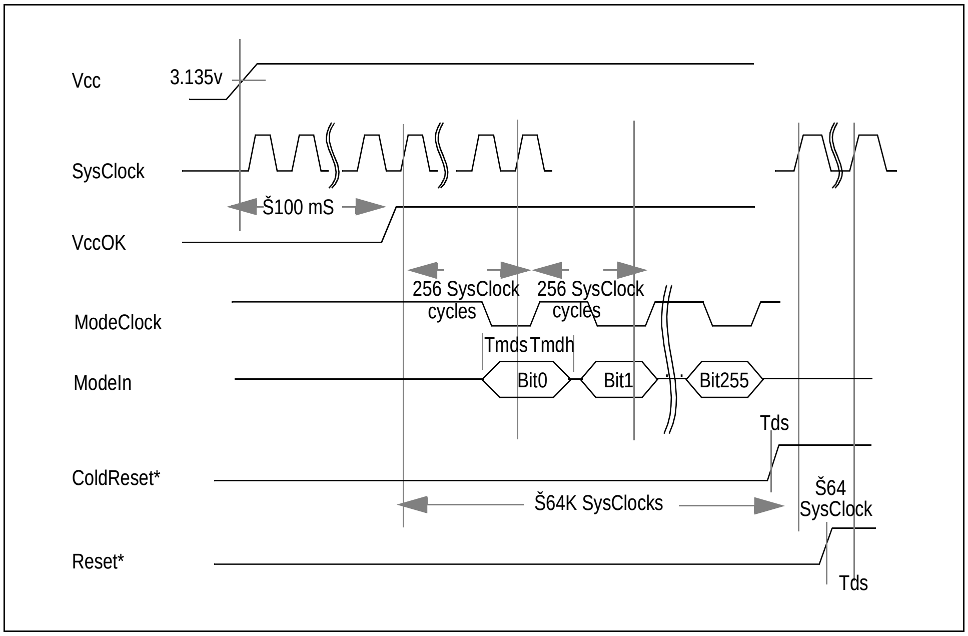


**Figure 14.1 Power-On Reset Timing Diagram**

## Cold Reset

A cold reset can begin anytime after the processor has read the initialization data stream, causing the processor to start with the Reset exception. A cold reset requires the same sequence as a power-on reset except that the power is presumed to be stable before the assertion of the reset inputs and the deassertion of **VccOk**. To begin the reset sequence, **VccOk** must be deasserted for a minimum of at least 64 MasterClock cycles before reassertion. Figure 14.2 shows the cold reset timing diagram.

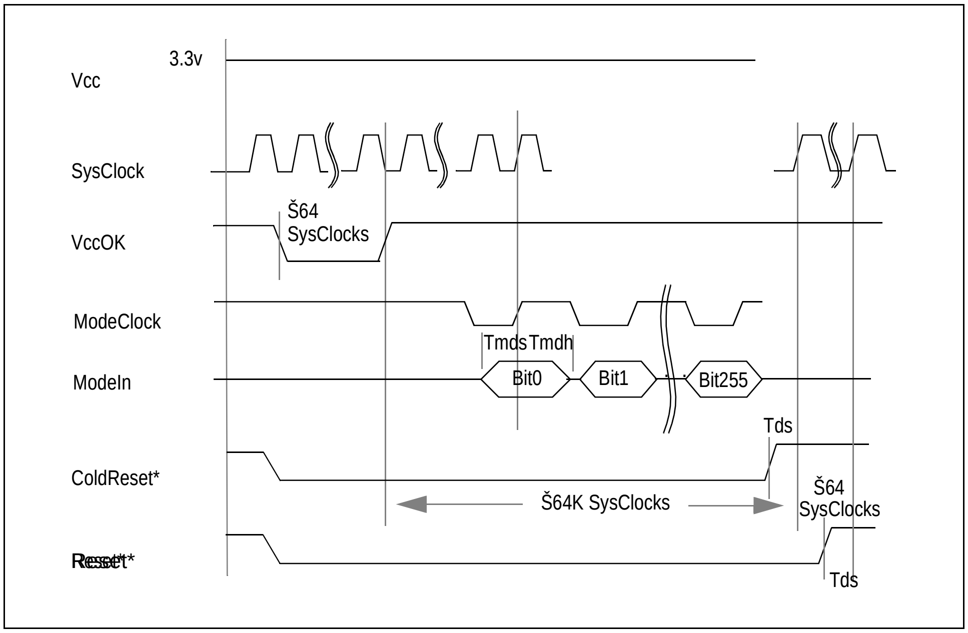


**Figure 14.2 Cold Reset Timing Diagram**

Notes

## Warm Reset

To execute a warm reset, the **Reset*** input is asserted synchronously with **SysClock**. It is then held asserted for at least 64 **SysClock** cycles before being deasserted synchronously with **SysClock**. The boot-time mode control serial data stream is not read by the processor on a warm reset. A warm reset forces the processor to start with a Soft Reset exception. Figure 14.3 shows the warm reset timing diagram.

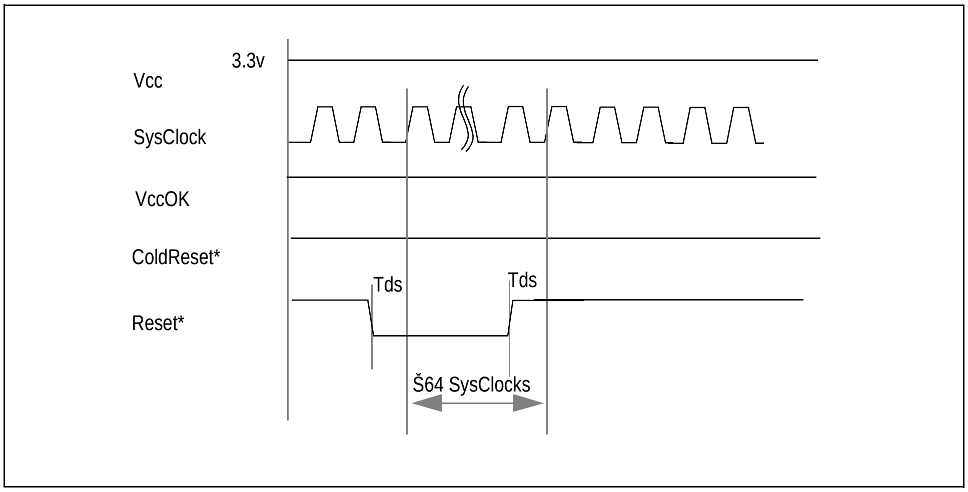


**Figure 14.3 Warm Reset Timing Diagram**

## Processor Reset State

After a power-on reset, cold reset, or warm reset, all processor internal state machines are reset, and the processor begins execution at the reset vector. All processor internal states are preserved during a warm reset, although the precise state of the caches depends on whether or not a cache miss sequence has been interrupted by resetting the processor state machines.

## Initialization Sequence

The boot-mode initialization sequence begins immediately after **VccOk** is asserted. As the processor reads the serial stream of 256 bits through the **ModeIn** pin, the boot-mode bits initialize all fundamental processor modes.

The initialization sequence is:

1. The system deasserts the **VccOk** signal. The **ModeClock** output is held asserted.
2. The processor synchronizes the **ModeClock** output at the time **VccOk** is asserted. The first rising edge of **ModeClock** occurs 256 **SysClock** cycles after **VccOk** is asserted.
3. Each bit of the initialization stream is presented at the **ModeIn** pin after each rising edge of the **ModeClock**. The processor samples 256 initialization bits from the **ModeIn** input.

## Boot-Mode Settings

The following rules apply to the boot-mode settings:

- *Bit 0 of the stream is presented to the processor when **VccOk** is first asserted.*
- *Selecting a reserved value results in undefined processor behavior.*
- *Zeros must be scanned in for all reserved bits.*

Table 14.1 shows the boot mode settings.

## Notes

| Bit | Value | Mode Setting |
|---|---|---|
| 0 | Reserved: must be zero | |
| 1:4 | XmitDatPat: System interface data rate for block writes only | |
| | 0 | DDDD |
| | 1 | DDxDDx |
| | 2 | DDxxDDxx |
| | 3 | DxDxDxDx |
| | 4 | DDxxxDDxxx |
| | 5 | DDxxxxDDxxxx |
| | 6 | DxxDxxDxxDxx |
| | 7 | DDxxxxxxDDxxxxxx |
| | 8 | DxxxDxxxDxxxDxxx |
| | 9:15 | Reserved |
| 5:7 | SysCkRatio: Pclock to SysClock Multiplier. | |
| | 0 | Multiply by 2 |
| | 1 | Multiply by 3 |
| | 2 | Multiply by 4 |
| | 3 | Multiply by 5 |
| | 4 | Multiply by 6 |
| | 5 | Multiply by 7 |
| | 6 | Multiply by 8 |
| | 7 | Reserved |
| 8 | EndBit: Specifies byte ordering. Logically ORed with the BigEndian signal. | |
| | 0 | Little-Endian |
| | 1 | Big Endian |
| 9:10 | Non-Block Write: Determines how non-block writes are handled. | |
| | 0 | R4x00 compatible |
| | 1 | Reserved |
| | 2 | Pipelined writes |
| | 3 | Write-reissue |
| 11 | TmrIntEn: Disables Timer Interrupt on Int*[5] | |
| | 0 | Timer Interrupt Enabled |
| | 1 | Timer Interrupt Disabled |
| 12 | Secondary Cache Enable | |
| | 0 | Secondary Cache Disabled |
| | 1 | Secondary Cache Enabled |

**Table 14.1 Boot Mode Settings (Part 1 of 2)**

Notes

| Bit | Value | Mode Setting |
|---|---|---|
| 13:14 | DrvOut: Output driver slew rate control | |
| | 10 | 100% (fastest) |
| | 11 | 83% |
| | 00 | 67% |
| | 01 | 50% (slowest) |
| 15 | Reserved: Must be zero | |
| 16:17 | Secondary cache size | |
| | 0 | 512 KByte secondary cache |
| | 1 | 1 MByte secondary cache |
| | 2 | 2 MByte secondary cache |
| | 3 | Reserved |
| 18:255 | Reserved: Must be zero | |

Table 14.1 Boot Mode Settings (Part 2 of 2)

## Driver Strength Control

The speed of the R5000 output drivers is statically controlled at boot time using an output buffer strength control mechanism. Two of the boot time mode bits are used to control the strength of the output buffer. These are boot mode bit 13 and 14. These bits select the static strength control for simple CMOS output buffers.

The output driver strength can be from 100% (fastest) to 50% (slowest), based on the value of boot mode bits 13 and 14. Table 14.2 shows the encoding for these boot mode bits and the selected driver strength.

| Boot Mode Bits<br>14 13 | Driver Strength |
|---|---|
| 1 0 | 100% |
| 1 1 | 83% |
| 0 0 | 67% |
| 0 1 | 50% |

Table 14.2 Boot Mode Bits and Drive Strength

## Notes

# Clock Interface and Standby Mode

Notes

## SysClock

The processor bases all internal and external clocking on the single **SysClock** input signal. Processor outputs and inputs have the following relationships to **SysClock**:

- *Output: Processor output data changes a minimum of $T_{dm}$ ns and becomes stable a maximum of $T_{do}$ ns after the rising edge of **SysClock**. This drive-time is the sum of the maximum delay through the processor output drivers together with the maximum clock-to-Q delay of the processor output registers.*
- *Input: Processor input data must be stable for a maximum of $T_{ds}$ ns before the rising edge of **SysClock** and must remain stable a minimum of $T_{dh}$ ns after the rising edge of **SysClock**.*

## PClock

The processor generates an internal clock, **PClock**, at the initialization-interface-specified frequency multiplier of **SysClock** and phase-aligned to **SysClock**. All internal registers and latches use **PClock**.

## Phase-Locked Loop (PLL)

The processor aligns **PClock** and **SysClock** with internal phase-locked loop (PLL) circuits that generate aligned clocks. By their nature, PLL circuits are only capable of generating aligned clocks for **SysClock** frequencies within a limited range.

Clocks generated using PLL circuits contain some inherent inaccuracy, or jitter; a clock aligned with **SysClock** by the PLL can lead or trail **SysClock** by as much as the related maximum jitter $T_{jo}$ allowed by the individual vendor. The $T_{jo}$ parameter must be added to the $T_{ds}$, $T_{dh}$, and $T_{do}$ parameters, and subtracted from the $T_{dm}$ parameters to get the total input and output timing parameters.

Figure 15.1 shows the **SysClock** timing parameters.

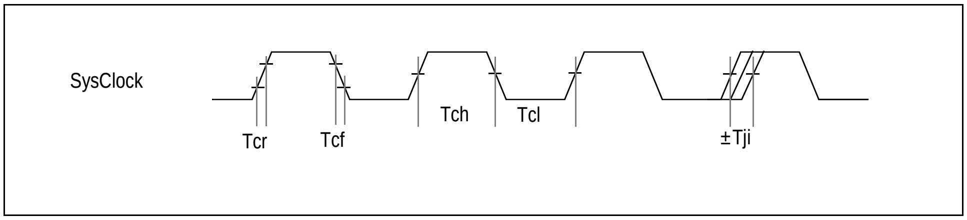


**Figure 15.1 SysClock Timing**

Figure 15.2 shows the input timing parameters.

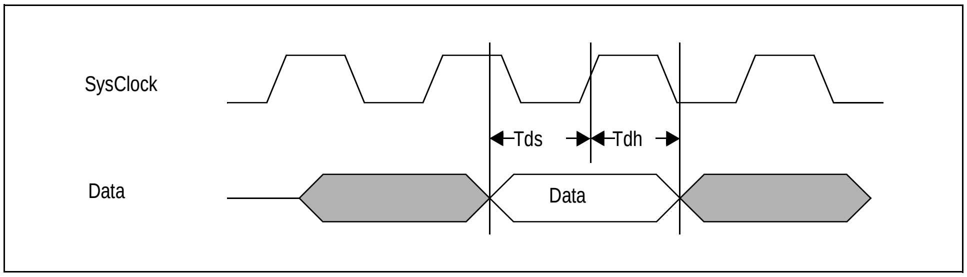


**Figure 15.2 Input Timing**

Figure 15.3 shows the output timing parameters measured at the midpoint of the rising clock edge.

Notes

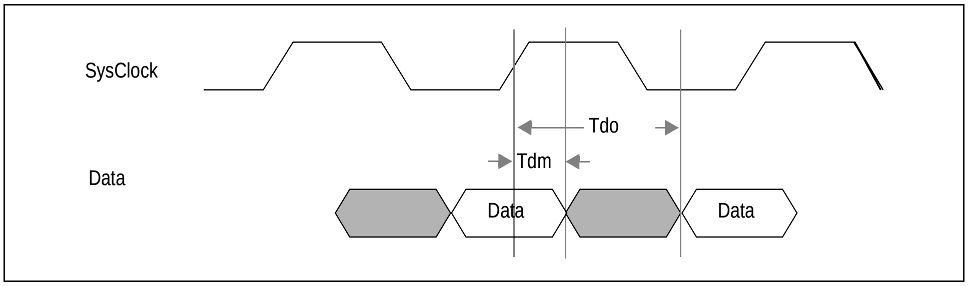


Figure 15.3 Output Timing

The **SysClock** input must meet the maximum rise time ($T_{cr}$), maximum fall time ($T_{cf}$), minimum $T_{ch}$ time, minimum $T_{cl}$ time, and $T_{ji}$ input jitter parameters for proper operation of the PLL.

## PLL Analog Power Filtering

For noisy module environments, a filter circuit of the following form (shown in Figure 15.4) is recommended.

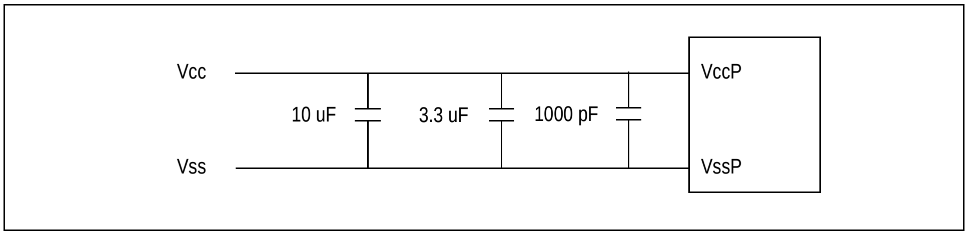


Figure 15.4 PLL Filter Circuit

## Standby Mode

The R5000 provides a *Standby Mode* in order to reduce the power consumed by the internal core when the CPU would otherwise not be performing any useful operations. To enter Standby Mode, first execute the WAIT instruction. When the WAIT instruction finishes the W pipe-stage, if the **SysAD** bus is currently idle, the internal clocks will shut down, thus freezing the pipeline. The PLL, internal timer, some of the input pin clocks (**Int[5:0]***, **NMI***, **ExtRqst***, **Reset*** and **ColdReset***) and the output clocks (**TClock[1:0]**, **RClock[1:0]**, **SyncOut**, **ModeClock** and **MasterOut**) will continue to run. If the conditions are not correct when the WAIT instruction finishes the W pipe-stage (i.e., the **SysAD** bus is not idle), the WAIT is treated as a NOP.

Once the CPU is in Standby Mode, any interrupt, including **ExtRqst*** or **Reset***, will cause the CPU to exit Standby Mode.

# Index