Mike Uhler
HLO 2-3/D11

MIPS R4000 Microprocessor User's Manual

2nd Edition (Preliminary)

Your comments on our products and publications are welcome.

MIPS Technologies, Inc.
950 DeGuigne Drive
Sunnyvale, CA 94088

Preface

This document describes the MIPS R4000 RISC-based microprocessor.

The chapters and appendices of this book are grouped in the following way:

- Architecture
- Implementation Details
- Electrical and Physical Specifications
- Instruction Set Summaries

Chapter 1 is a general discussion (including a historical context) of the RISC microprocessor in general and the R4000 in particular.

Chapter 2 provides an overview of the CPU instruction set by summarizing each instruction category in a table.

Chapter 3 describes the operation of the R4000 instruction execution pipeline. It describes the basic operation of the pipeline and interruptions to the pipeline flow caused by interlocks and exceptions.

Chapter 4 is a discussion of the memory management system including memory mapping, virtual memory, and address translation.

Chapter 5 is a discussion of the exception processing respurces and capabilities of the R4000. It presents an overview of the CPU exception handling process and describes the format and use of each CPU exception handling register.

Chapter 6 is a discussion of the Floating-Point Unit (FPU). The FPU is a coprocessor for the Central Processing Unit (CPU) that extends the CPU instruction set to perform floating-point arithmetic operations.

Chapter 7 is a discussion of the Floating-Point Unit's exception processing.

Chapter 8 is a discussion of the signals that comprise the interface between the R4000 and other components in the system. The signals discussed include the System Interface, the Clock/Control Interface, the Secondary Cache Interface, the Interrupt Interface, the Initialization Interface, and the JTAG Interface.

Chapter 9 is a discussion of the system interface. The system interface allows the processor access to external resources such as memory and I/O. It also allows an external agent access to certain processor internal resources.

Chapter 10 is a discussion of the clocks used in the R4000 and the processor status reporting mechanism. The topics covered include the basic System Clocks, interfacing to a Phase-Locked system, interfacing to a system without Phase Locking, and processor Status Outputs.

Chapter 11 is a discussion of the cache memory hierarchy, the operation of the primary and secondary caches, and the R4000's interface to the secondary cache. It also discusses cache-coherent operation in a multiprocessor system

Chapter 12 is a discussion of the Initialization interface. The fundamental, or 'start-up', operational modes for the processor are introduced to the processor through the initialization interface.

Chapter 13 is a discussion of the JTAG interface. The JTAG boundary scan mechanism provides a capability for testing the interconnection between the R4000 processor, the printed circuit board to which it is attached, and the other components on the board.

Chapter 14 is a discussion of the six hardware, two software, and one non-maskable processor interrupts.

Chapter 15 is a discussion of the Error Checking and Correcting (ECC) mechanisms of the R4000.

Chapter 16 is a discussion of the electrical and physical characteristics of the R4000.

Appendix A is a detailed description of the operation of each R4000 instruction in both 32- and 64-bit modes. The instructions are listed in alphabetical order.

Appendix B is a detailed description of the operation of each (FPU) instruction. The instructions are listed alphabetically.

Appendix C is a discussion of the Single Error Correcting Double Error Detecting (SECDED) codes. These are the codes chosen for the processor's secondary cache data and secondary cache tag.

Appendix D is a discussion of sub-block ordering. Sub-block ordering is an order for the transmission of data elements that form a block of data when the first transmitted data element is not the data element at the beginning of the block.

Appendix E is a discussion of the output buffer the Δi/Δt control mechanism which controls the speed of the R4000 output driver, ensuring drive-off times are only as fast as necessary to meet thesystem requirement of single cycle transfers.

Appendix F is a discussion of the passive components which comprise the Phase-Locked Loop (PLL).

Appendix G is a desciption of Coprocessor 0 hazards.

Introduction

1

This introductory chapter provides you with the following information:

- An explanation of RISC architecture, with subsections describing the benefits of using RISC design, the relationship between RISC architecture and optimizing compilers, and a description of the MIPS compiler family.
- An overview of the R4000 features, including the Memory Management System, pipeline architecture, memory hierarchy, and interfaces to external cache memory and the remainder of the system.

What Is RISC?

Historically, the evolution of computer architectures has been dominated by families of increasingly complex central processors. Under market pressures to preserve existing software, Complex Instruction Set Computer (CISC) architectures evolved by the accretion of microcode and increasingly intricate instruction sets. This intricacy in architecture was itself driven by the need to support high-level languages (HLLs) and operating systems, as advances in semiconductor technology made it possible to fabricate integrated circuits of greater and greater complexity. And at the time it seemed self-evident to designers that architectures *should* become more complex as technological advances made such VLSI designs possible.

In recent years however, Reduced Instruction Set Computer (RISC) architectures have implemented a different model for the interaction between hardware, firmware, and software. RISC concepts emerged from a statistical analysis of the manner in which software actually uses processor resources: dynamic measurement of system kernels and object modules generated by optimizing compilers showed that the simplest instructions were used most often—even in the code for CISC machines! Correspondingly, complex instructions were often unused because their single way of performing a complex operation rarely matched the precise needs of the high-level language.

RISC, on the other hand, eliminated microcode routines and turned low-level control of the machine over to software. The RISC approach was not new, but its application became more universal in recent years, due to the increasing prevalence of high-level languages, the development of compilers able to optimize at the microcode level, and dramatic advances in semiconductor memory and packaging. It is now feasible to replace a machine's relatively-slow microcode ROM with faster RAM, organized as an instruction cache. Machine control then resides in this instruction cache that is, in effect, customized on the fly: the instruction stream generated by system- and compiler-generated code provides a precise fit between the requirements of high-level software and the low-level capabilities of the hardware.

Reducing or simplifying the instruction set was not the primary goal of RISC architecture; it is a pleasant side effect of techniques used to gain the highest performance possible from available technology. Thus, the term *Reduced Instruction Set Computers* is a bit misleading: it is the push for performance that really drives and shapes RISC designs.

Benefits of RISC Design

Some of the benefits that result from RISC design techniques are not directly attributable to the drive to increase performance, but are a result of the basic reduction in complexity—a simpler design allows both chip-area resources and human resources to be applied to features that enhance performance. Some of these benefits are described below.

Shorter Design Cycle

The architectures of RISC processors can be implemented more quickly than their CISC counterparts: it is easier to fabricate and debug a streamlined, simplified architecture with no microcode than a complex, microcoded architecture. CISC processors have such a long design cycle that they may not be completely debugged by the time they have been rendered technologically obsolete. The shorter time required to design and implement RISC processors allows them to make use of the best available technologies.

Effective Utilization of Chip Area

The simplicity of RISC processors also frees scarce chip geography for performance-critical resources such as larger register files, Translation Lookaside Buffers (TLBs), coprocessors, and fast multiply and divide units. Such resources help RISC processors obtain an even greater performance edge.

User (Programmer) Benefits

Simplicity in architecture also helps the user in the following ways:

- A uniform instruction set is easier to use.
- A closer correlation is made possible between the instruction count and the cycle count, making it easier to measure code optimization activities.

Advanced Semiconductor Technologies

Each new VLSI technology (ECL, GaAs) is introduced with tight limits on the number of transistors that can be fit on each chip. Since the simplicity of a RISC processor allows it to be implemented in fewer transistors than its CISC counterpart, the first computers capable of exploiting these new VLSI technologies have been using and will continue to use RISC architecture.

Optimizing Compilers

RISC architecture is designed so that compilers, not assembly languages, have the optimal working environment. RISC philosophy assumes that high-level language (HLL) programming is used, a philosophy in contrast to the older CISC philosophy developed when assembly language programming was of primary importance.

The trend toward HLL instructions has led to the development of more efficient compilers to convert HLL instructions to machine code. Primary measures of a compiler's efficiency are:

- the compactness of its generated code
- the shortness of its execution time

Optimizing compilers and RISC architectures have a synergistic relationship; compilers perform their best job of optimization in a RISC environment. Reciprocally, RISC architectures rely on compilers to obtain their best performance.

During the development of more efficient compilers, an analysis of instruction streams revealed that the greatest amount of time was spent:

- executing simple instructions
- performing load and store operations

while the more complex instructions were used less frequently.

It was also learned that compilers produce code that is often a narrow subset of the processor's instruction set architecture (ISA). A compiler prefers instructions that perform simple, well-defined operations and generate minimal side-effects. Complex instructions and features are just not used by compilers; the more complex, powerful instructions are either too difficult for the compiler to use or those instructions do not precisely fit HLL requirements.

Thus, a natural match exists between RISC architectures and efficient, optimizing compilers. This match makes it easier for compilers to generate the most effective sequences of machine instructions to accomplish tasks defined by the high-level language.

Family of Compilers

Many compiler products—especially those designed for microprocessors—are cobbled from various sources and do not necessarily fit together very well. However, the MIPS *language suite* approach shares common elements across the family of compilers instead of treating each language's compiler as a separate entity. In this way the MIPS suite of compilers, RISCompilers™, can offer both tight integration and broad language coverage.

The MIPS suite of compilers does the following:

- Provides industry-standard front ends for six languages (C, FORTRAN, Pascal, Ada, PLI, COBOL)
- Uses a common intermediate language, thus offering an efficient way to add language front ends over time
- Shares all of the back end optimization and code generation
- Uses the same object format and calling conventions
- Supports mixed-language programs cleanly
- Supports debugging of programs written in all languages, including mixtures

This *language suite* approach yields high-quality compilers for all languages, since common elements make up the majority of each of the language products. In addition, the ability to develop and execute multi-language programs is provided, promoting flexibility in development, avoiding recode of proven program segments, and protecting the user's software investment. The common back-end also exports optimizing and code-generating improvements immediately throughout the suite of RISCompilers, thereby reducing maintenance.

64-bit Architecture

The MIPS R4000 family of RISC microprocessors consists of high-performance 32-bit and 64-bit processors; the natural mode of operation for the R4000 is as a 64-bit microprocessor. It can, however, be programmed to operate as a 32-bit processor.

The R4000 provides a 64-bit on-chip floating-point unit (FPU), 64-bit integer ALU, 64-bit integer registers, and a 64-bit virtual address space. 32-bit applications maintain compatibility even when the processor operates as a 64-bit processor.

The R4000 Processor

The R4000 has many features that differ from the R2000/R3000 processor family. In addition to a high-performance integer unit, the R4000 contains:

- a 48-entry fully-associative on-chip TLB, with two pages mapped to each entry
- separate on-chip primary data and instruction caches
- an optional off-chip secondary cache
- an on-chip FPU

Figure 1-1 shows a block diagram of the R4000.

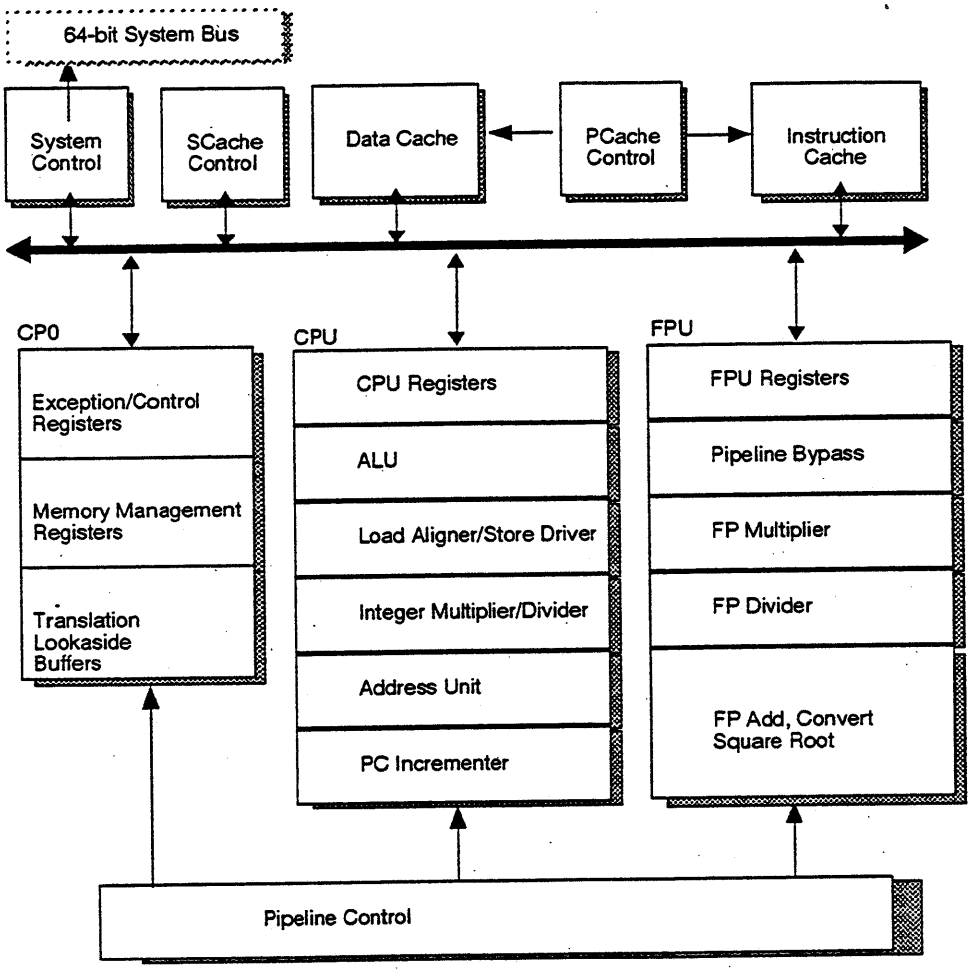


Figure 1-1 R4000 Internal Block Diagram

Processor General Features

This section briefly describes the programming model, the MMU, and the caches in the R4000. A more detailed description is given in succeeding sections.

- **Full 32-bit and 64-bit Operation.** The R4000 contains thirty-two general-purpose 64-bit registers. (When operating as a 32-bit processor, the general-purpose registers are 32-bits wide.) All instructions are thirty-two bits wide.

- **Efficient Pipelining.** The superpipeline design of the processor results in an execution rate approaching one instruction per cycle. Pipeline stalls and exceptional events are handled precisely and efficiently.
- **MMU.** The R4000 processor uses an on-chip TLB that provides rapid virtual-to-physical address translation of:
  - 2-GByte user virtual address space in 32-bit mode
  - 512-Gbyte user virtual address space in 64-bit mode.
- **Cache Control.** The R4000 primary instruction and data caches reside on-chip, and can each hold from 8 Kbytes to 32 Kbytes. An off-chip secondary cache can hold from 128 Kbytes to 4 MBytes. All R4000 cache control, including the secondary cache control, logic is on-chip.
- **Floating Point Unit.** The FPU is located on-chip and implements the ANSI/IEEE standard 754-1985.

CPU Registers

The CPU provides thirty-two general-purpose registers, a Program Counter (PC), and two registers that hold the results of integer multiply and divide operations. These registers are either 32-bits or 64-bits wide, depending on the mode of operation. Two general-purpose registers have special functions:

- *r0* is hardwired to a value of zero. *r0* can be used as the target register for any instruction the results of which can be discarded. *r0* can also be used as a source when a zero value is needed.
- *r31* is the link register for JumpAndLink instructions. It should not be used explicitly by other instructions.

The MIPS architecture defines three special registers whose use or modification is implicit with certain instructions. These special registers are:

• *PC*	Program Counter
• *HI*	Multiply and Divide Register higher result
• *LO*	Multiply and Divide Register lower result

The two Multiply and Divide Registers (*HI, LO*) store the doubleword, 64-bit result or quadword, 128-bit result of integer multiply operations and the quotient (in *LO*) and remainder (in *HI*) of integer divide operations.

Figure 1-2 shows the CPU Registers.

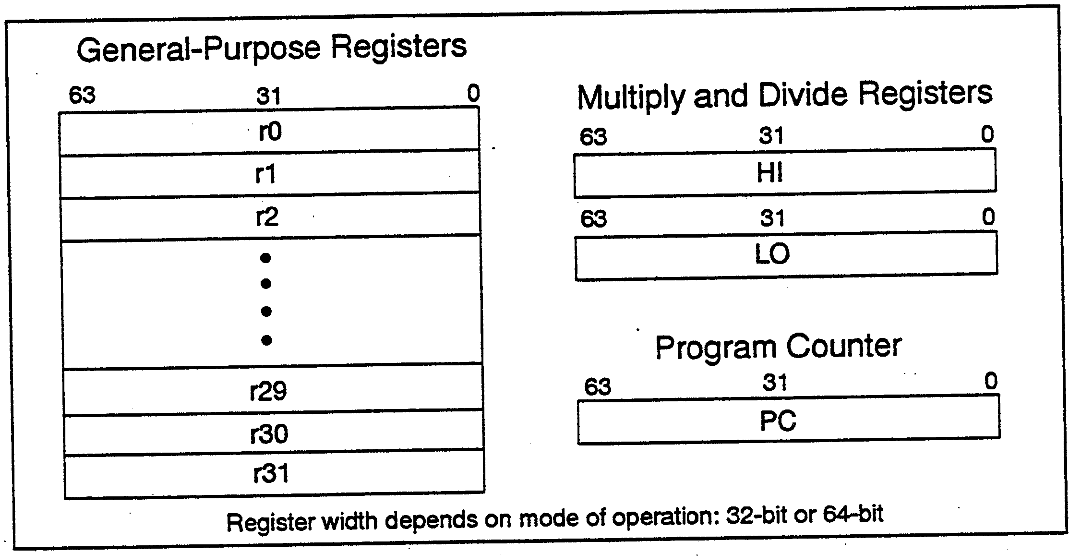

General-Purpose Registers
63
31
0
r0
r1
r2
r29
r30
r31
Multiply and Divide Registers
63
31
0
HI
63
31
0
LO
Program Counter
63
31
0
PC
Register width depends on mode of operation: 32-bit or 64-bit

Figure 1-2 CPU Registers

The R4000 has no Program Status Word (PSW) Register; its functions are provided by the *Status* and *Cause* Registers incorporated within Coprocessor 0 (CP0). CP0 registers are described later in this chapter.

CPU Instruction Set Overview

Each CPU instruction is thirty-two bits long. As shown in Figure 1-3, there are three instruction formats: immediate (I-type), jump (J-type), and register (R-type). Using only these three instruction formats simplifies instruction decoding, more complicated (and less frequently used) operations and addressing modes can be synthesized by the compiler using sequences of these simple instructions.

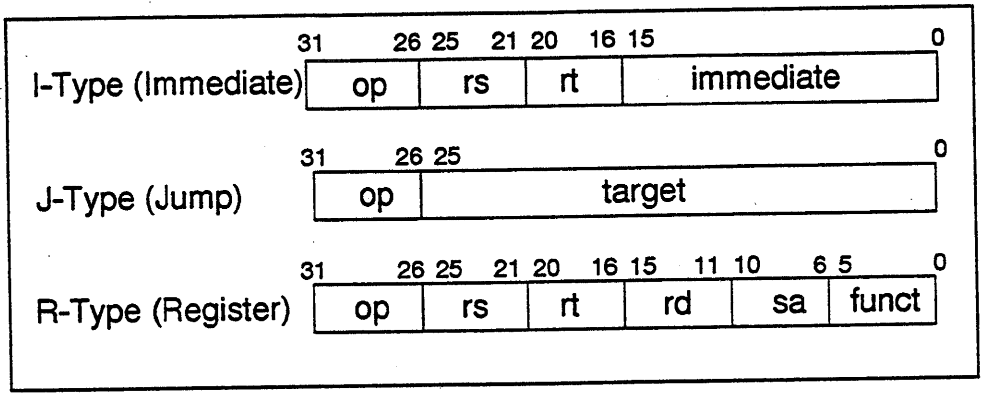


Figure 1-3 CPU Instruction Formats

The instruction set can be divided into the following groups:

- **Load and Store** instructions move data between memory and general registers. They are all 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 occur in both R-type (both the operands and the result are stored in registers) and I-type (one operand is a 16-bit immediate value) formats.
- **Jump and Branch** instructions change the control flow of a program. Jumps are always 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 addresses (R-type format). Branches have 16-bit offsets relative to the program counter (I-type). JumpAndLink instructions save a return address in register 31.
- **Coprocessor** instructions perform operations in the coprocessors. Coprocessor load and store instructions are I-type (see the FPU instructions in Chapter 5).
- **Coprocessor 0** instructions perform operations on CP0 registers to manipulate the memory management and exception handling facilities of the processor. Table 1-3 shows these instructions.
- **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.

A more detailed summary is provided in Chapter 2 and a complete description of each instruction is given in Appendix A.

Table 1-1 lists the instruction set (ISA) common to all MIPS R-Series processors; Table 1-2 lists R4000 instructions that are extensions to the ISA. These instructions result in code space reductions, multiprocessor support, and improved performance in operating system kernel code sequences and in situations where run-time bounds checking is frequently performed.

Table 1-1 CPU Instruction Set (ISA)

OP	Description
	Load and Store Instructions
LB	Load Byte
LBU	Load Byte Unsigned
LH	Load Halfword
LHU	Load Halfword Unsigned
LW	Load Word
LWL	Load Word Left
LWR	Load Word Right
SB	Store Byte
SH	Store Halfword
SW	Store Word
SWL	Store Word Left
SWR	Store Word Right
	Arithmetic Instructions (ALU Immediate)
ADDI	Add Immediate
ADDIU	Add Immediate Unsigned
SLTI	Set on Less Than Immediate
SLTIU	Set on Less Than Immediate Unsigned
ANDI	AND Immediate
ORI	OR Immediate
XORI	Exclusive OR Immediate
LUI	Load Upper Immediate
	Arithmetic Instructions (3-operand, R-type)
ADD	Add
ADDU	Add Unsigned
SUB	Subtract
SUBU	Subtract Unsigned
SLT	Set on Less Than
SLTU	Set on Less Than Unsigned
AND	AND
OR	OR
XOR	Exclusive OR
NOR	NOR
	Shift Instructions
SLL	Shift Left Logical
SRL	Shift Right Logical
SRA	Shift Right Arithmetic
SLLV	Shift Left Logical Variable
SRLV	Shift Right Logical Variable
SRAV	Shift Right Arithmetic Variable

OP	Description
	Multiply and Divide Instructions
MULT	Multiply
MULTU	Multiply Unsigned
DIV	Divide
DIVU	Divide Unsigned
MFHI	Move From HI
MTHI	Move To HI
MFLO	Move From LO
MTLO	Move To LO
	Jump and Branch Instructions
J	Jump
JAL	Jump And Link
JR	Jump Register
JALR	Jump And Link Register
BEQ	Branch on Equal
BNE	Branch on Not Equal
BLEZ	Branch on Less than or Equal to Zero
BGTZ	Branch on Greater Than Zero
BLTZ	Branch on Less Than Zero
BGEZ	Branch on Greater than or Equal to Zero
BLTZAL	Branch on Less Than Zero And Link
BGEZAL	Branch on Greater than or Equal to Zero And Link
	Coprocessor Instructions
LWCz	Load Word to Coprocessor z
SWCz	Store Word from Coprocessor z
MTCz	Move To Coprocessor z
MFCz	Move From Coprocessor z
CTCz	Move Control to Coprocessor z
CFCz	Move Control From Coprocessor z
COPz	Coprocessor Operation z
BCzT	Branch on Coprocessor z True
BCzF	Branch on Coprocessor z False
	Special Instructions
SYSCALL	System Call
BREAK	Break

Table 1-2 Extensions to the ISA

OP	Description	OP	Description
	Load and Store Instructions		**Multiply and Divide Instructions**
LD	Load Doubleword	DMULT	Doubleword Multiply
LDL	Load Doubleword Left	DMULTU	Doubleword Multiply Unsigned
LDR	Load Doubleword Right	DDIV	Doubleword Divide
LL	Load Linked	DDIVU	Doubleword Divide Unsigned
LLD	Load Linked Doubleword		**Jump and Branch Instructions**
LWU	Load Word Unsigned	BEQL	Branch on Equal Likely
SC	Store Conditional	BNEL	Branch on Not Equal Likely
SCD	Store Conditional Doubleword	BLEZL	Branch on Less than or Equal to Zero Likely
SD	Store Doubleword		
SDL	Store Doubleword Left	BGTZL	Branch on Greater Than Zero Likely
SDR	Store Doubleword Right	BLTZL	Branch on Less Than Zero Likely
SYNC	Sync	BGEZL	Branch on Greater than or Equal to Zero Likely
	Arithmetic Instructions (ALU Immediate)	BLTZALL	Branch on Less Than Zero And Link Likely
DADDI	Doubleword Add Immediate		
DADDIU	Doubleword Add Immediate Unsigned	BGEZALL	Branch on Greater than or Equal to Zero And Link Likely
		BCzTL	Branch on Coprocessor z True Likely
	Arithmetic Instructions (3-operand, R-type)	BCzFL	Branch on Coprocessor z False Likely
			Exception Instructions
DADD	Doubleword Add	TGE	Trap if Greater Than or Equal
DADDU	Doubleword Add Unsigned	TGEU	Trap if Greater Than or Equal Unsigned
DSUB	Doubleword Subtract	TLT	Trap if Less Than
DSUBU	Doubleword Subtract Unsigned	TLTU	Trap if Less Than Unsigned
	Shift Instructions	TEQ	Trap if Equal
DSLL	Doubleword Shift Left Logical	TNE	Trap if Not Equal
DSRL	Doubleword Shift Right Logical	TGEI	Trap if Greater Than or Equal Immediate
DSRA	Doubleword Shift Right Arithmetic	TGEIU	Trap if Greater Than or Equal Immediate Unsigned
DSLLV	Doubleword Shift Left Logical Variable		
		TLTI	Trap if Less Than Immediate
DSRLV	Doubleword Shift Right Logical Variable	TLTIU	Trap if Less Than Immediate Unsigned
		TEQI	Trap if Equal Immediate
DSRAV	Doubleword Shift Right Arithmetic Variable	TNEI	Trap if Not Equal Immediate
DSLL32	Doubleword Shift Left Logical + 32		
			Coprocessor Instructions
DSRL32	Doubleword Shift Right Logical + 32	DMFCz	Doubleword Move From Coprocessor z
		DMTCz	Doubleword Move To Coprocessor z
DSRA32	Doubleword Shift Right Arithmetic + 32	LDCz	Load Double Coprocessor z
		SDCz	Store Double Coprocessor z

Table 1-3 CP0 Instructions

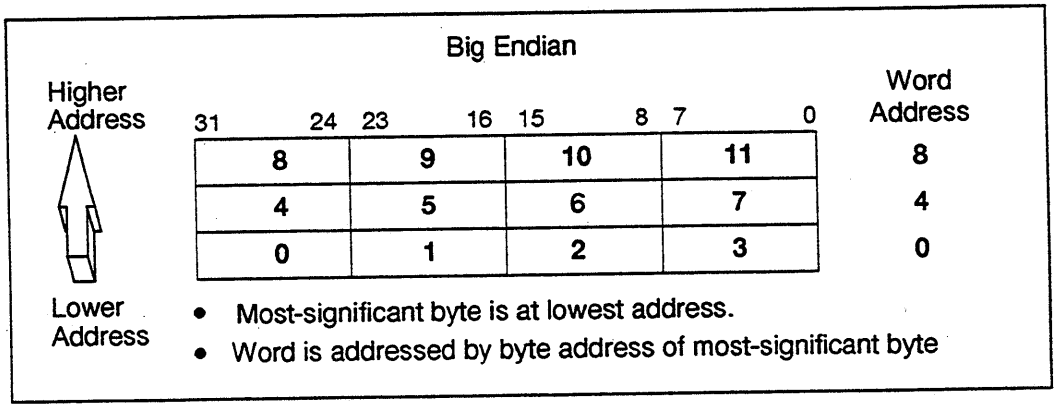
Op	Description
DMFC0	Doubleword Move From CP0
DMTC0	Doubleword Move To CP0
MTC0	Move to CP0
MFC0	Move from CP0
TLBR	Read Indexed TLB Entry
TLBWI	Write Indexed TLB Entry
TLBWR	Write Random TLB Entry
TLBP	Probe TLB for Matching Entry
ERET	Exception Return

Data Formats and Addressing

The R4000 uses four data formats: a 64-bit doubleword, a 32-bit word, a 16-bit halfword and an 8-bit byte. The byte ordering is configurable as either *Big-endian* or *Little-endian* format. Endianness refers to the location of byte 0 within a multi-byte structure.

Figure 1-4 and Figure 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 R4000 is configured as a Big-endian system, byte 0 is the most-significant (leftmost) byte, thereby providing compatibility with MC 68000® and IBM 370® conventions. This configuration is shown in Figure 1-4.

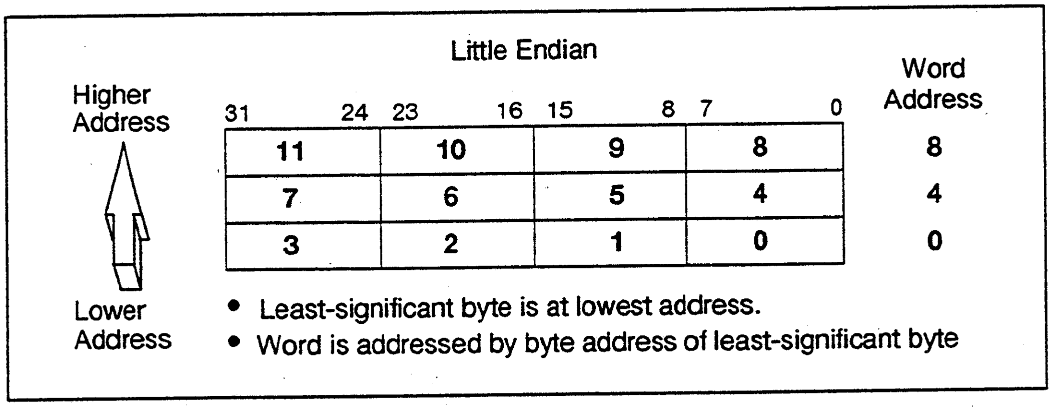


31–24	23–16	15–8	7–0	Word Address
8	9	10	11	8
4	5	6	7	4
0	1	2	3	0

- Most-significant byte is at lowest address.
- Word is addressed by byte address of most-significant byte

Figure 1-4 Addresses of Bytes within Words: Big-endian Byte Alignment

When configured as a Little-endian system, byte 0 is always the least-significant (rightmost) byte, which is compatible with iAPX® x86 and DEC VAX® conventions. This configuration is shown in Figure 1-5.



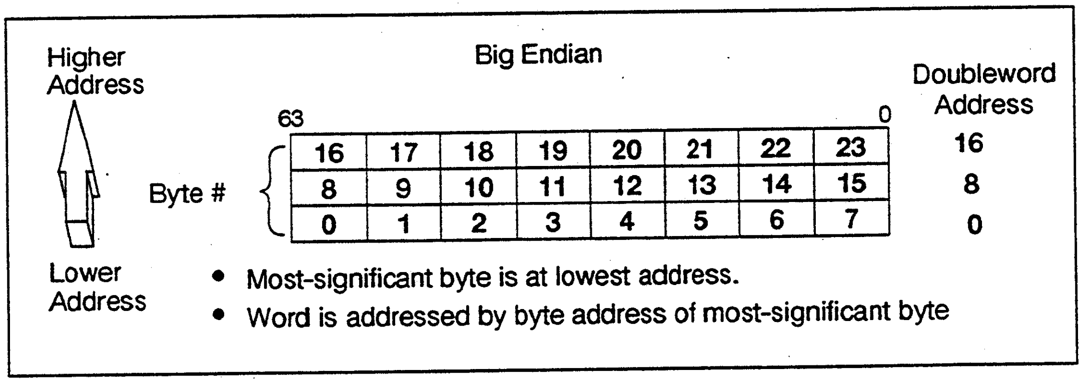
31 … 24	23 … 16	15 … 8	7 … 0	Word Address
11	10	9	8	8
7	6	5	4	4
3	2	1	0	0

- Least-significant byte is at lowest address.
- Word is addressed by byte address of least-significant byte

Figure 1-5 Addresses of Bytes within Words: Little-endian Byte Alignment

In this book, bit 0 is always the least-significant (rightmost) bit; thus, bit designations are always Little Endian (although no instructions explicitly designate bit positions within words).

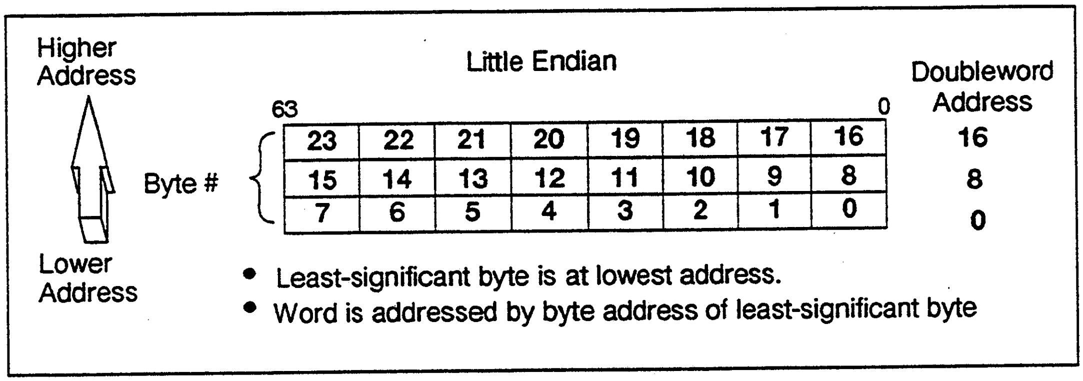
Figure 1-6 and Figure 1-7 show byte alignment in doublewords.



63							0	Doubleword Address
16	17	18	19	20	21	22	23	16
8	9	10	11	12	13	14	15	8
0	1	2	3	4	5	6	7	0

- Most-significant byte is at lowest address.
- Word is addressed by byte address of most-significant byte

Figure 1-6 Addresses of Bytes within Doublewords: Big-endian Byte Alignment



								Doubleword Address
23	22	21	20	19	18	17	16	16
15	14	13	12	11	10	9	8	8
7	6	5	4	3	2	1	0	0

- Least-significant byte is at lowest address.
- Word is addressed by byte address of least-significant byte

Figure 1-7 Addresses of Bytes within Doublewords: Little-endian Byte Alignment

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...).

As shown in Figure 1-6 and Figure 1-7, the address of a multiple-byte data item is the address of the most-significant byte on a Big-endian configuration, or the address of the least-significant byte on a Little-endian configuration.

Special instructions are provided for loading and storing words and doublewords that are not aligned on 4-byte (word) or 8-word (double word) boundaries: LWL, LWR, SWL, SWR, LDL, LDR, SDL, SDR. These instructions are used in pairs to provide addressing of misaligned words with one additional instruction cycle over that required for aligned words. For each of the two endianness conventions, Figure 1-8 shows the bytes that are accessed when addressing a misaligned word with byte address 3.

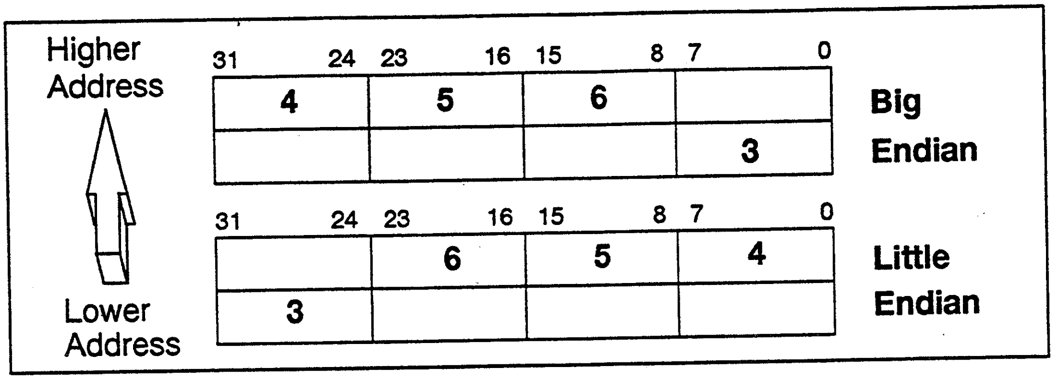


Figure 1-8 Example Misaligned Word: Byte Address $3

System Control Coprocessor (CP0)

The MIPS ISA allows up to four coprocessors (designated CP0 through CP3). Coprocessor 1 (CP1) is reserved for the on-chip, floating-point coprocessor. Coprocessor 2 (CP2) is reserved for future definition by MIPS, and the encoding for Coprocessor 3 (CP3) is used to provide certain extensions to the MIPS ISA. Coprocessor 0 (CP0) is also incorporated on the CPU chip and supports the virtual memory system and exception handling. The virtual memory system is implemented with an on-chip TLB and a group of programmable registers, as described in Table 1-4.

CP0 translates virtual addresses into physical addresses and manages exceptions and transitions between kernel, supervisor, and user states. It also controls the cache subsystem and provides diagnostic control and error recovery facilities. The R4000 also provides a generic system timer for interval timing, timekeeping, process accounting, and time-slicing (see the *Count* and *Compare* Registers in Chapter 5).

Register Name	Reg. #	Register Name	Reg. #
▨ Index	0	□ Config	16
▨ Random	1	□ LLAddr	17
▨ EntryLo0	2	□ WatchLo	18
▨ EntryLo1	3	□ WatchHi	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

Legend

□ Exception Processing ▨ Memory Management ■ Reserved

Figure 1-9 The R4000 CP0 Registers

The CP0 registers shown in Figure 1-9 and described in Table 1-4 manipulate the memory management and exception handling capabilities of the CPU. Refer to Chapter 4 for a detailed description of the registers associated with the virtual memory system and to Chapter 5 for descriptions of the exception processing registers.

Table 1-4 System Control Coprocessor (CP0) Registers

Number	Register	Description
0	Index	Programmable pointer into TLB array
1	Random	Pseudorandom pointer into TLB array (read only)
2	EntryLo0	Low half of TLB entry for even VPN
3	EntryLo1	Low half of TLB entry for odd VPN
4	Context	Pointer to kernel virtual PTE table in 32-bit addressing mode
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	WatchLo	Memory reference trap address low bits
19	WatchHi	Memory reference trap address high bits
20	XContext	Pointer to kernel virtual PTE table in 64-bit addressing mode
21–25	—	Reserved
26	ECC	Secondary-cache 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

Floating-Point Unit (FPU)

The MIPS Floating-Point Unit (FPU) operates as a coprocessor for the CPU and extends the CPU instruction set to perform arithmetic operations on values in floating-point representations. The FPU, with associated system software, fully conforms to the requirements of ANSI/IEEE Standard 754–1985, "IEEE Standard for Binary Floating-Point Arithmetic."

The FPU features:

- **Full 64-bit operation.** The FPU contains 16 64-bit registers or, optionally, thirty-two 64-bit registers that hold single-precision or double-precision values. The 16 additional floating-point registers are enabled by setting the *FR* bit in the *Status* register. The FPU also includes a 32-bit *Status/ Control* Register that provides access to all IEEE-Standard exception handling capabilities.
- **Load and Store Instruction Set.** Like the CPU, the FPU uses a load- and store-oriented instruction set. Floating-point operations are started in a single cycle and their execution is overlapped with other fixed-point or floating-point operations.
- **Tightly coupled Coprocessor Interface.** The FPU is on-chip and appears to the programmer as an extension of the CPU (the FPU is accessed as Coprocessor 1). This forms 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 rate (2 instructions per cycle) as fixed-point-instructions. The FPU instructions are summarized in Chapter 6, *Floating-Point Unit*.

On-chip Caches

The R4000 incorporates on-chip instruction and data caches to keep the high-performance pipeline full. Each cache has its own 64-bit data path that can be accessed in parallel. The caches can be accessed twice in one cycle. Combining this feature with a pipelined, single-cycle access of each cache, the cache subsystem provides the integer and floating-point units with an aggregate bandwidth of 1.6 GBytes per second at a Master Clock frequency of 50 MHz. The R4000 caches are described in detail in Chapter 11, *Cache Organization, Operation, and Coherency*.

Memory Management System

The R4000 has a physical addressing range of 64 Gbytes (36 bits). However, since most systems implement a physical memory smaller than 4 Gbytes, the CPU provides a logical expansion of memory space by translating addresses composed in a large virtual address space into available physical memory addresses. In 32-bit mode, the virtual address space is divided into 2 Gbytes per user process and 2 Gbytes for the kernel. In 64-bit mode, the virtual address is expanded to allow 512 Gbytes of user virtual address space.

The Translation Lookaside Buffer (TLB)

Virtual memory mapping is assisted by a TLB. This TLB caches virtual address translations. The fully-associative, on-chip TLB contains 48 entries, and each of these entries maps a pair of variable-sized pages (page size varies from 4 KBytes to 16 MBytes, increasing by multiples of 4). An address translation value is tagged with the most-significant bits of its virtual address (the number of these bits depends upon the size of the page) and a per-process identifier. If there is no matching entry in the TLB, an exception is taken and software refills the on-chip TLB from a Page Table resident in memory. An entry, chosen at random, is replaced to make way for the new one. This TLB is referred to as the JTLB.

The R4000 also has a two-entry instruction TLB (ITLB) to assist in instruction address translation. The ITLB is completely invisible to software and is present for performance reasons only.

Operating Modes

The R4000 CPU has three operating modes: *User* mode, *Kernel* mode, and *Supervisor* mode. The CPU normally operates in *User* mode until an exception is detected forcing it into *Kernel* mode. It remains in *Kernel* mode until an Exception Return (ERET) instruction is executed. The *Supervisor* mode can be used to design secure operating systems. The manner in which memory addresses are translated or *mapped* depends on the operating mode of the CPU. Chapter 4 describes the MMU and Operating modes in greater detail.

R4000 Superpipeline Architecture

The R4000 exploits instruction-level parallelism using a superpipelined implementation. The R4000 uses an 8-stage superpipeline which places no restrictions on the instruction issued. Under normal circumstances, any two instructions are issued each cycle.

The internal pipeline of the R4000 operates at twice the frequency of the master clock. This is shown in Figure 1-10. The 8-stage superpipeline of the R4000 achieves high throughput by pipelining cache accesses, shortening register access times, implementing virtual indexed primary caches, and allowing the latency of functional units to span multiple pipeline clock cycles (pcycles). In the rest of this document, the internal pipeline clock and clock cycles are often referred to as pclock and pcycles respectively. The R4000 superpipeline is covered in greater detail in Chapter 3.

The execution of a single R4000 CPU instruction consists of the following eight primary steps:

IF Instruction fetch First half. Virtual address is presented to the I-cache and TLB.

IS Instruction fetch Second half. The I-cache outputs the instruction and the TLB generates the physical address.

RF Register File. Three activities occur in parallel:

- instruction is decoded and a check is made for interlock conditions,
- instruction tag check is made to determine if there is a cache hit or not,
- operands are fetched from the register file.

EX Instruction EXecute. One of three activities can occur:

- if the instruction is a register-to-register operation, an arithmetic, logical, shift, multiply, or divide operation is performed;
- if the instruction is a load and store, the data virtual address is calculated;

- if the instruction is a branch, the branch target virtual address is calculated and branch conditions are checked.

DF Data cache First half. A virtual address is presented to the D-cache and TLB.

DS Data cache Second half. The D-cache outputs the instruction and the TLB generates the physical address.

TC Tag Check. A tag check is performed for loads and stores to determine if there is a hit or not.

WB Write Back. The instruction result is written back to the register file.

The R4000 uses an 8-stage pipeline; thus, execution of 8 instructions at a time are overlapped, as shown in Figure 1-10.

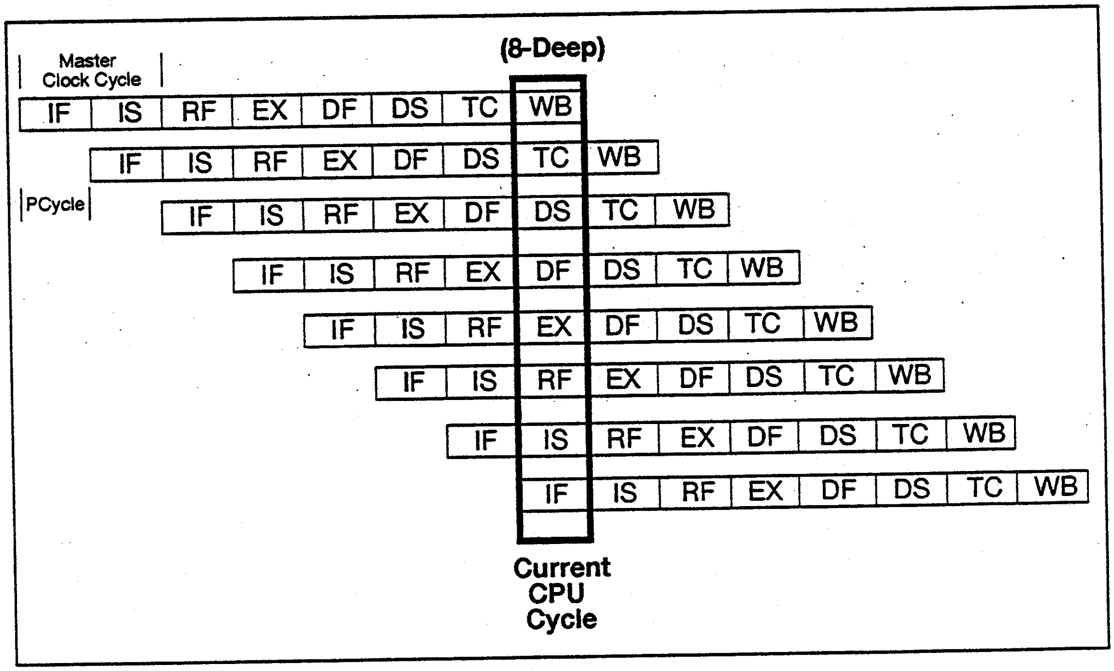


Figure 1-10 R4000 Pipeline and Instruction Overlapping

Cache Memory Hierarchy

To achieve its high performance in uniprocessor and multiprocessor systems, the R4000 supports a cache memory hierarchy that increases memory access bandwidth and reduces the latency of load and store instructions. The two-level cache memory hierarchy consists of on-chip instruction and data caches, and an optional external secondary cache that can vary in size from 128 Kbytes to 4 Mbytes.

The secondary cache is assumed to consist of one bank of industry-standard static RAM (SRAM) with output enables. The secondary cache consists of a quadword (128 bit) wide data array and a 25-bit wide tag array. Check fields are added to both the data and tag arrays to improve data integrity. The secondary cache may be configured as either a joint cache or split instruction/data cache. The maximum secondary cache size is 4 MBytes and the minimum secondary cache size is 128 KBytes for a joint cache and 256 KBytes for split instruction/data cache. The secondary cache is direct-mapped, and is addressed with the lower part of the physical address.

A detailed description of the cache hierarchy is given in Chapter 11, *Cache Organization, Operation, and Coherency*.

Secondary Cache Interface

The R4000SC and R4000MC versions of the R4000 interface to an optional secondary cache. The R4000 provides all of the secondary cache control circuitry, including ECC protection, on chip. The secondary cache interface consists of a 128-bit data bus, a 25-bit tag bus, an 18-bit address bus and SRAM control signals. The 128-bit wide data bus minimizes cache miss penalty, and allows the use of standard low-cost SRAMs in the secondary cache design.

System Interface

The R4000 supports a 64-bit system interface that can be used to construct uniprocessor systems with a direct DRAM interface with or without a secondary cache or cache-coherent multiprocessor systems. The interface consists of a 64-bit multiplexed address and data bus with 8 check bits and a 9-bit parity-protected command bus. In addition, there are 8 handshake signals. The interface has a simple timing specification and is capable of transferring data between the processor and memory at a peak rate of 400 Mbytes/second at 50 MHz.

R4000 Configurations

The R4000 is packaged in three different configurations. All processors are implemented in sub-1 micron CMOS technology:

- The R4000SC is designed for use in high-performance uniprocessor systems. It is packaged in a 447-pin LGA/PGA and includes integrated control for large secondary caches built from standard SRAMs.
- The R4000MC is designed for use in large cache-coherent multiprocessor systems. The R4000MC is also packaged in 447-pin LGA/PGA and includes, in addition, support for a wide variety of bus designs and cache-coherency mechanisms.
- The R4000PC is designed for cost-sensitive systems such as inexpensive desktop systems and high-end embedded controllers. It is packaged in a 179-pin PGA. The R4000PC does not support a secondary cache.

Compatibility

The R4000 provides complete application software compatibility with the MIPS R2000, R3000, and R6000 processors. Although the architecture has evolved in response to a compromise between software and hardware resources in the computer system, this evolution maintains object-code compatibility for programs that execute in User mode (see Chapter 4, *Memory Management System*, for a description of operating modes). Like its predecessors, the R4000 implements the MIPS Instruction Set Architecture (ISA) for user-mode programs; this guarantees that user-mode programs conforming to the ISA will execute on any MIPS hardware implementation.

CPU Instruction Set Summary

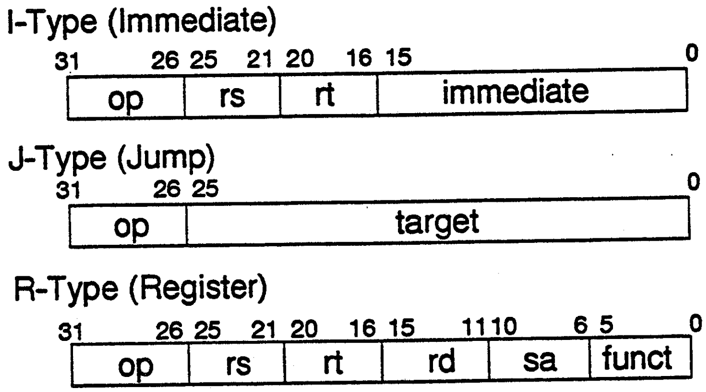
2

This chapter provides an overview of the CPU instruction set by summarizing each instruction category in a table. Refer to Appendix A for individual descriptions of each CPU instruction.

The FPU instructions are summarized in Chapter 6, and are described in detail in Appendix B.

Instruction Formats

Each CPU instruction consists of a single word (32 bits) aligned on a word boundary. There are three instruction formats, as shown in Figure 2-1. The use of these three instruction formats simplifies instruction decoding since the compiler can synthesize more complicated (and less frequently used) operations and addressing modes. In the MIPS architecture, coprocessor instructions are implementation-dependent; see Appendix A for R4000 Coprocessor 0 instruction details.



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

Figure 2-1 CPU Instruction Formats

Load and Store Instructions

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

The instruction immediately following a load can use the contents of the loaded register. In such cases, hardware interlocks require additional real cycles; consequently, scheduling load delay slots is still desirable, for both performance and R3000 compatibility. However, the scheduling of load delay slots is not absolutely required for functional code.

The load and store instruction opcode determines the access type which indicates the size of the data item to be loaded or stored as shown in Figure 2-2. Regardless of access type or byte-numbering order (endianness), the address specifies the byte with the smallest byte address in the addressed field. For a Big-endian configuration, it is the most-significant byte; for a Little-endian configuration, it is the least-significant byte.

The bytes that are used within the addressed doubleword can be determined from the access type and the three low-order bits of the address, as shown in Figure 2-2. Only the combinations shown in Figure 2-2 are permissible; other combinations cause address error exceptions. Table 2-1 lists the load and store instructions defined by the ISA. Table 2-2 lists the instructions which are extensions to the ISA.

Figure 2-2 Byte Specifications for Load and Store Instructions

Access Type Mnemonic (Value)	Low Order Address Bits			Bytes Accessed															
				Big-Endian 63——0 Byte								Little-Endian 63——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	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
	1	0	0	▨	▨	▨	▨	4	5	6	7	▨	▨	▨	▨	▨	▨	▨	▨
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 2-1 Load and Store Instruction Summary (ISA)

Instruction	Format and Description	op	base	rt	offset
Load Byte	*LB rt,offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Sign-extend contents of addressed byte and load into register *rt.*				
Load Byte Unsigned	*LBU rt,offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Zero-extend contents of addressed byte and load into register *rt.*				
Load Halfword	*LH rt,offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Sign-extend contents of addressed halfword and load into register *rt.*				
Load Halfword Unsigned	*LHU rt,offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Zero-extend contents of addressed halfword and load into register *rt.*				
Load Word	*LW rt,offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Load contents of addressed word into register *rt.* (sign extended if 64-bit mode)				
Load Word Left	*LWL rt,offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Shift addressed word left so that addressed byte is leftmost byte of a word. Merge bytes from memory with contents of register *rt* and load the result into register *rt.* (sign extended if 64-bit mode)				
Load Word Right	*LWR rt,offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Shift addressed word right so that addressed byte is rightmost byte of a word. Merge bytes from memory with contents of register *rt* and load the result into register *rt.* (sign extended if 64-bit mode)				
Store Byte	*SB rt,offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Store the least-significant byte of register *rt* at addressed location.				
Store Halfword	*SH rt,offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Store the least-significant halfword of register *rt* at addressed location.				
Store Word	*SW rt,offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Store the contents of the least significant word of register *rt* at addressed location.				
Store Word Left	*SWL rt,offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Shift contents of register *rt* left so that the leftmost byte of the low-order word is in the position of the addressed byte. Store the bytes containing the original data in the low-order word into corresponding bytes at addressed byte.				
Store Word Right	*SWR rt,offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Shift contents of register *rt* right so that the rightmost byte of the low-order word is in theposition of the addressed byte. Store the bytes containing the original data in the low-order word into corresponding bytes at addressed byte.				

Table 2-2 Load and Store Instruction (ISA Extensions)

Instruction	Format and Description op | base | rt | offset
Load Doubleword	*LD rt,offset(base)* Sign-extend 16-bit *offset* and add to contents of register ***base*** to form address. Load contents of addressed double word into register *rt*.
Load Doubleword Left	*LDL rt, offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Shift addressed doubleword left so that addressed byte is leftmost byte of a doubleword. Merge bytes from memory with contents of register *rt* and load the result into register *rt*.
Load Doubleword Right	*LDR rt, offset(base)* Sign-extend 16-bit *offset* and add to contents of register ***base*** to form address. Shift addressed doubleword right so that addressed byte is rightmost byte of a doubleword. Merge bytes from memory with contents of register *rt* and load the result into register *rt*.
Load Linked	*LL rt,offset(base)* Sign-extend 16-bit *offset* and add to contents of register ***base*** to form address. Sign-extend contents of addressed word and load into register *rt*.
Load Linked Doubleword	*LLD rt,offset(base)* Sign-extend 16-bit *offset* and add to contents of register ***base*** to form address. Load contents of addressed doubleword into register *rt*.
Load Word Unsigned	*LWU rt,offset(base)* Sign-extend 16-bit *offset* and add to contents of register ***base*** to form address. Zero extend contents of addressed word and load into register *rt*.
Store Doubleword	*SD rt,offset(base)* Sign-extend 16-bit *offset* and add to contents of register ***base*** to form address. Store contents of register *rt* at addressed location.
Store Conditional	*SC rt,offset(base)* Sign-extend 16-bit *offset* and add to contents of register ***base*** to form address. Conditionally store low-order word of register *rt* at addressed location.
Store Conditional Doubleword	*SCD rt,offset(base)* Sign-extend 16-bit *offset* and add to contents of register ***base*** to form address. Conditionally store contents of register *rt* at addressed location.
Store Doubleword Left	*SDL rt,offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Shift contents of register *rt* left so that the leftmost byte of the word is in the position of the addressed byte. Store the bytes containing the original data in the low-order doubleword into corresponding bytes at the addressed byte.
Store Doubleword Right	*SDR rt,offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Shift contents of register *rt* right so that the rightmost byte of the word is in the position of the addressed byte. Store the bytes containing the original data in the low-order doubleword into corresponding bytes at the addressed byte.
Sync	*SYNC* Complete all outstanding load or store instructions before allowing any new load and store instruction to start.

Computational Instructions

Computational instructions perform arithmetic, logical, shift, multiply, and divide operations on values in registers. They occur in both register (R-type) format, in which both operands are registers, and immediate (I-type) format, in which one operand is a 16-bit immediate. There are four categories of computational instructions:

- ALU Immediate instructions
- Three-Operand Register-Type instructions
- Shift instructions
- Multiply and Divide instructions

When operating in 64-bit mode, 32-bit operands must be correctly sign extended. The result of operations which use incorrectly sign-extended, 32-bit values is unpredictable.

Table 2-3 ALU Immediate Instruction Summary

Instruction	**Format and Description** op rs rt immediate
ADD Immediate	*ADDI rt,rs,immediate* Add 16-bit sign-extended *immediate* to register *rs* and place the 32-bit result (sign-extended in 64-bit mode) in register *rt*. Trap on 2's-complement overflow.
ADD Immediate Unsigned	*ADDIU rt,rs,immediate* Add 16-bit sign-extended *immediate* to register *rs* and place the 32-bit result (sign-extended in 64-bit mode) in register *rt*. Do not trap on overflow.
Set on Less Than Immediate	*SLTI rt,rs,immediate* Compare 16-bit sign-extended *immediate* with register *rs* as signed integers. Result is set to 1 if *rs* is less than *immediate*; otherwise result is set to 0. Place result in register *rt*.
Set on Less Than Immediate Unsigned	*SLTIU rt,rs,immediate* Compare 16-bit sign-extended *immediate* with register *rs* as unsigned integers. Result is set to 1 if *rs* is less than *immediate*; otherwise result is set to 0. Place result in register *rt*.
AND Immediate	*ANDI rt,rs,immediate* Zero-extend 16-bit *immediate*, AND with contents of register *rs* and place the result in register *rt*.
OR Immediate	*ORI rt,rs,immediate* Zero-extend 16-bit *immediate*, OR with contents of register *rs* and place the result in register *rt*.
Exclusive OR Immediate	*XORI rt,rs,immediate* Zero-extend 16-bit *immediate*, exclusive OR with contents of register *rs* and place the result in register *rt*.
Load Upper Immediate	*LUI rt,immediate* Shift 16-bit *immediate* left 16 bits. Set least-significant 16 bits of word to zeros. Store the result in register *rt*.

Table 2-4 ALU Immediate Instruction (ISA Extensions)

Instruction	**Format and Description** **op** | **rs** | **rt** | **immediate**
DADD Immediate	*DADDI rt,rs,immediate* Add 16-bit sign-extended *immediate* to register *rs* and place the 64-bit result in register *rt*. Trap on 2's-complement overflow.
DADD Immediate Unsigned	*DADDIU rt,rs,immediate* Add 16-bit sign-extended *immediate* to register *rs* and place the 64-bit result in register *rt*. Do not trap on overflow.

Table 2-5 Three-Operand Register-Type Instruction Summary

Instruction	Format and Description op | rs | rt | rd | sa | function
Add	*ADD rd,rs,rt* Add contents of registers *rs* and *rt* and place the 32-bit result (sign-extended in 64-bit mode) in register *rd.* Trap on 2's-complement overflow.
Add Unsigned	*ADDU rd,rs,rt* Add contents of registers *rs* and *rt* and place the 32-bit result (sign-extended in 64-bit mode) in register *rd.* Do not trap on overflow.
Subtract	*SUB rd,rs,rt* Subtract contents of registers *rt* from *rs* and place the 32-bit result (sign-extended in 64-bit mode) in register *rd.* Trap on 2's-complement overflow
Subtract Unsigned	*SUBU rd,rs,rt* Subtract contents of registers *rt* from *rs* and place the 32-bit result (sign-extended in 64-bit mode) in register *rd.* Do not trap on overflow.
Set on Less Than	*SLT rd,rs,rt* Compare contents of register *rt* to register *rs* as signed integers. Result is set to 1 if *rs* is less than *rt*; otherwise result is set to 0.Place result in register *rd.*
Set on Less Than Unsigned	*SLTU rd,rs,rt* Compare contents of register *rt* to register *rs* as unsigned integers. Result is set to 1 if *rs* is less than *rt*; otherwise result is set to 0.Place result in register *rd.*
AND	*AND rd,rs,rt* Bitwise AND the contents of registers *rs* and *rt*, and place the result in register *rd.*
OR	*OR rd,rs,rt* Bitwise OR the contents of registers *rs* and *rt*, and place the result in register *rd.*
Exclusive OR	*XOR rd,rs,rt* Bitwise exclusive OR the contents of registers *rs* and *rt*, and place the result in register *rd.*
NOR	*NOR rd,rs,rt* Bitwise NOR the contents of registers *rs* and *rt*, and place the result in register *rd.*

Table 2-6 Three-Operand Register-Type Instruction (ISA Extensions)

Instruction	**Format and Description**	**op**	**rs**	**rt**	**rd**	**sa**	**function**
Doubleword Add	*DADD rd,rs,rt* Add contents of registers *rs* and *rt* and place the 64-bit result in register *rd.* Trap on 2's-complement overflow.						
Doubleword Add Unsigned	*DADDU rd,rs,rt* Add contents of registers *rs* and *rt* and place the 64-bit result in register *rd.* Do not trap on overflow.						
Doubleword Subtract	*DSUB rd,rs,rt* Subtract contents of registers *rt* from *rs* and place the 64-bit result in register *rd.* Trap on 2's-complement overflow.						
Doubleword Subtract Unsigned	*DSUBU rd,rs,rt* Subtract contents of registers *rt* from *rs* and place the 64-bit result in register *rd.* Do not trap on overflow.						

Table 2-7 Shift Instruction Summary

Instruction	Format and Description op rs rt rd sa function
Shift Left Logical	*SLL rd,rt,sa* Shift the contents of register *rt* left by *sa* bits, and insert zeros into the low-order bits. Place the 32-bit result in register *rd*. (sign-extended in 64-bit mode)
Shift Right Logical	*SRL rd,rt,sa* Shift the contents of register *rt* right by *sa* bits, and insert zeros into the high-order bits. Place the 32-bit result in register *rd*. (sign-extended in 64-bit mode)
Shift Right Arithmetic	*SRA rd,rt,sa* Shift the contents of register *rt* right by *sa* bits, and sign-extend the high-order bits. Place the 32-bit result in register *rd*. (sign-extended in 64-bit mode)
Shift Left Logical Variable	*SLLV rd,rt,rs* Shift the contents of register *rt* left. The low-order 5 bits of register *rs* specify the number of bits to shift left; insert zeros into the low-order bits of *rt* and place the 32-bit result in register *rd*. (sign-extended in 64-bit mode)
Shift Right Logical Variable	*SRLV rd,rt,rs* Shift the contents of register *rt* right. The low-order 5 bits of register *rs* specify the number of bits to shift right; insert zeros into the high-order bits of *rt* and place the 32-bit result in register *rd*. (sign-extended in 64-bit mode)
Shift Right Arithmetic Variable	*SRAV rd,rt,rs* Shift the contents of register *rt* right. The low-order 5 bits of register *rs* specify the number of bits to shift right; sign-extend the high-order bits of *rt* and place the 32-bit result in register *rd*. (sign-extended in 64-bit mode)

Table 2-8 Shift Instruction (ISA Extensions)

Instruction	Format and Description	op	rs	rt	rd	sa	function
Doubleword Shift Left Logical	*DSLL rd,rt,sa* Shift the contents of register *rt* left by *sa* bits, and insert zeros into the low-order bits. Place the 64-bit result in register *rd.*						
Doubleword Shift Right Logical	*DSRL rd,rt,sa* Shift the contents of register *rt* right by *sa* bits, and insert zeros into the high-order bits. Place the 64-bit result in register *rd.*						
Doubleword Shift Right Arithmetic	*DSRA rd,rt,sa* Shift the contents of register *rt* right by *sa* bits, and sign-extend the high-order bits. Place the 64-bit result in register *rd.*						
Doubleword Shift Left Logical Variable	*DSLLV rd,rt,rs* Shift the contents of register *rt* left. The low-order 6 bits of register *rs* specify the number of bits to shift left; insert zeros into the low-order bits of *rt* and place the 64-bit result in register *rd.*						
Doubleword Shift Right Logical Variable	*DSRLV rd,rt,rs* Shift the contents of register *rt* right. The low-order 6 bits of register *rs* specify the number of bits to shift right; insert zeros into the high-order bits of *rt* and place the 64-bit result in register *rd.*						
Doubleword Shift Right Arithmetic Variable	*DSRAV rd,rt,rs* Shift the contents of register *rt* right. The low-order 6 bits of register *rs* specify the number of bits to shift right; sign-extend the high-order bits of *rt* and place the 64-bit result in register *rd.*						
Doubleword Shift Left Logical+32	*DSLL32 rd,rt,sa* Shift the contents of register *rt* left by *32+sa* bits, and insert zeros into the low-order bits. Place the 64-bit result in register *rd.*						
Doubleword Shift Right Logical+32	*DSRL32 rd,rt,sa* Shift the contents of register *rt* right by *32+sa* bits, and insert zeros into the highorder bits. Place the 64-bit result in register *rd.*						
Doubleword Shift Right Arithmetic+32	*DSRA32 rd,rt,sa* Shift the contents of register *rt* right by *32+sa* bits, and sign-extend the high-order bits. Place the 64-bit result in register *rd.*						

Table 2-9 Multiply and Divide Instruction Summary

Instruction	Format and Description	op	rs	rt	rd	sa	function
Multiply	*MULT rs,rt* Multiply the contents of registers *rs* and *rt* as 2's-complement values. Place the 64-bit result in special registers *HI* and *LO*. (sign-extended in 64-bit mode)						
Multiply Unsigned	*MULTU rs,rt* Multiply the contents of registers *rs* and *rt* as unsigned integers. Place the 64-bit result in special registers *HI* and *LO*. (sign-extended in 64-bit mode)						
Divide	*DIV rs,rt* Divide the contents of register *rs* by *rt*, treating operands as 2's-complement values. Place the 32-bit quotient in special register *LO* and the 32-bit remainder in *HI*. (sign-extended in 64-bit mode)						
Divide Unsigned	*DIVU rs,rt* Divide the contents of register *rs* by *rt*, treating operands as unsigned values. Place the 32-bit quotient in special register *LO* and the 32-bit remainder in *HI*. (sign-extended in 64-bit mode)						
Move From HI	*MFHI rd* Move the contents of special register *HI* to register *rd*.						
Move From LO	*MFLO rd* Move the contents of special register *LO* to register *rd*.						
Move To HI	*MTHI rd* Move the contents of register *rd to* special register *HI*.						
Move To LO	*MTLO rd* Move the contents of register *rd to* special register *LO*.						

Table 2-10 Multiply and Divide Instruction (ISA Extensions)

Instruction	Format and Description	op	rs	rt	rd	sa	function
Doubleword Multiply	*DMULT rs,rt* Multiply the contents of registers *rs* and *rt* as 2's-complement values. Place the 128-bit result in special registers *HI* and *LO*.						
Doubleword Multiply Unsigned	*DMULTU rs,rt* Multiply the contents of registers *rs* and *rt* as unsigned integers. Place the 128-bit result in special registers *HI* and *LO*.						
Doubleword Divide	*DDIV rs,rt* Divide the contents of register *rs* by *rt*, treating operands as 2's-complement values. Place the 64-bit quotient in special register *LO* and the 64-bit remainder in *HI*.						
Doubleword Divide Unsigned	*DDIVU rs,rt* Divide the contents of register *rs* by *rt*, treating operands as unsigned values. Place the 64-bit quotient in special register *LO* and the 64-bit remainder in *HI*.						

The number of cycles required for multiply and divide operations is shown in Table 2-11. The MFHI and MFLO instructions are interlocked so that any attempt to read them before prior operations have completed will cause execution of these instructions to be delayed until the operation finishes. Table 2-11 gives the number of pcycles required between a MULT, MULTU, DIV, DIVU, DMULT, DMULTU, DDIV or DDIVU operation, and a subsequent MFHI or MFLO operation, to resolve an interlock or stall.

Table 2-11 Multiply/Divide Instruction Cycle Timing

PCycles Required							
MULT	MULTU	DIV	DIVU	DMULT	DMULTU	DDIV	DDIVU
10	10	69	69	20	20	133	133

Jump and Branch Instructions

Jump and branch instructions change the control flow of a program. All jump and branch instructions occur with an architectural delay of one instruction: that is, the instruction immediately following the jump or branch (the instruction in the *delay slot*) is always executed while the target instruction is being fetched from storage. (Taken branches have a 3 cycle penalty in this implementation. Refer to Chapter 3:*The R4000 Pipeline* for details.).

Subroutine calls in high-level languages are usually implemented with Jump or JumpAndLink instructions. Both are J-type instructions. In this format, the 26-bit target address is shifted left two bits, and combined with the high-order four bits of the current program counter to form an absolute address.

Returns, dispatches, and large cross-page jumps are usually implemented with the JumpRegister and JumpAndLinkRegister instructions. Both are R-type instructions which take a 32-bit or 64-bit byte address contained in one of the general-purpose registers.

Table 2-12 and Table 2-13 summarize those CPU jump and branch instructions that are shared by all MIPS R-Series processors; Table 2-14 summarizes branch instructions that are extensions for the R4000.

Table 2-12 Jump Instruction Summary

Instruction	**Format and Description** | op | target |
Jump	*J target* Shift the 26-bit *target* address left two bits, combine with high-order four bits of the PC, and jump to the address with a 1-instruction delay.
Jump And Link	*JAL target* Shift the 26-bit *target* address left two bits, combine with high-order four bits of the PC, and jump to the address with a 1-instruction delay. Place the address of the instruction following the delay slot in *r31* (*Link* register).

Instruction	**Format and Description** | op | rs | rt | rd | sa | function |
Jump Register	*JR rs* Jump to the address contained in register *rs*, with a 1-instruction delay.
Jump And Link Register	*JALR rs, rd* Jump to the address contained in register *rs*, with a 1-instruction delay. Place the address of the instruction following the delay slot in register *rd*.

The following description is common to Table 2-13 and Table 2-14.

- Branch target: All branch instruction target addresses are computed by adding the address of the instruction in the delay slot and the 16-bit *offset* (shifted left two bits and sign-extended to 32 bits). All branches occur with a delay of one instruction.
- Conditional branch (Table 2-14): If the conditional branch is not taken, the instruction in the delay slot is nullified.

The following format fields are found in Table 2-13 to Table 2-20:

- REGIMM - Opcode
- Sub - Sub-operation Code
- CO - Sub-operation Specifier
- BC - BC Sub-opcode
- br - Branch Condition Specifier
- cofun - Coprocessor Function Field
- op - Operation Code

Table 2-13 Branch Instruction Summary

Instruction	Format and Description op rs rt offset
Branch on Equal	*BEQ rs,rt,offset* Branch to target address if register *rs* is equal to register *rt*.
Branch on Not Equal	*BNE rs,rt,offset* Branch to target address if register *rs* is not equal to register *rt*.
Branch on Less than or Equal Zero	*BLEZ rs,offset* Branch to target address if register *rs* is less than or equal to zero.
Branch on Greater Than Zero	*BGTZ rs,offset* Branch to target address if register *rs* is greater than zero.

Instruction	Format and Description REGIMM rs sub offset
Branch on Less Than Zero	*BLTZ rs,offset* Branch to target address if register *rs* is less than zero.
Branch on Greater than or Equal Zero	*BGEZ rs,offset* Branch to target address if register *rs* is greater than or equal to zero.
Branch on Less Than Zero And Link	*BLTZAL rs,offset* Place address of instruction following the delay slot in register *r31* (Link register). Branch to target address if register *rs* is less than zero.
Branch on Greater than or Equal Zero And Link	*BGEZAL rs,offset* Place address of instruction following the delay slot in register *r31* (Link register). Branch to target address if register *rs* is greater than or equal to zero.

Table 2-14 Branch Instruction Summary - (ISA Extensions)

Instruction	**Format and Description** op rs rt offset
Branch on Equal Likely	*BEQL rs,rt,offset* Branch to target address if register *rs* is equal to register *rt*.
Branch on Not Equal Likely	*BNEL rs,rt,offset* Branch to target address if register *rs* is not equal to register *rt*.
Branch on Less Than or Equal to Zero Likely	*BLEZL rs,offset* Branch to target address if register *rs* is less than or equal to zero.
Branch on Greater Than Zero Likely	*BGTZL rs,offset* Branch to target address if register *rs* is greater than zero.

Instruction	**Format and Description** REGIMM rs sub offset
Branch on Less Than Zero Likely	*BLTZL rs,offset* Branch to target address if register *rs* is less than zero.
Branch on Greater Than or Equal to Zero Likely	*BGEZL rs,offset* Branch to target address if register *rs* is greater than or equal to zero.
Branch on Less Than Zero And Link Likely	*BLTZALL rs,offset* Place address of instruction following the delay slot in register *r31* (Link register). Branch to target address if register *rs* is less than zero.
Branch on Greater Than or Equal to Zero And Link Likely	*BGEZALL rs,offset* Place address of instruction following the delay slot in register *r31* (Link register). Branch to target address if register *rs* is greater than or equal to zero.

Special Instructions

Special instructions allow the software to initiate traps and are always R-type. Special instructions that are valid for all MIPS R-Series processors are shown in Table 2-15.

Table 2-15 Special Instructions

Instruction	Format and Description	SPECIAL	rs	rt	rd	sa	function
System Call	*SYSCALL* Initiates system call trap, immediately transferring control to exception handler.						
Breakpoint	*BREAK* Initiates breakpoint trap, immediately transferring control to exception handler.						

Exception Instructions

Exception instructions are extensions to the ISA and are shown in Table 2-16 and Table 2-17.

Table 2-16 Exception Instructions (ISA Extensions)

Instruction	Format and Description	SPECIAL	rs	rt	rd	sa	function
Trap if Greater Than or Equal	*TGE rs,rt* Trap exception occurs if register *rs* is greater than or equal to register *rt*, considering both quantities as signed integers.						
Trap if Greater Than or Equal Unsigned	*TGEU rs,rt* Trap exception occurs if register *rs* is greater than or equal to register *rt*, considering both quantities as unsigned integers.						
Trap if Less Than	*TLT rs,rt* Trap exception occurs if register *rs* is less than register *rt*, considering both quantities as signed integers..						
Trap if Less Than Unsigned	*TLTU rs,rt* Trap exception occurs if register *rs* is less than register *rt*, considering both quantities as unsigned integers..						
Trap if Equal	*TEQ rs,rt* Trap exception occurs if register *rs* is equal to register *rt*.						
Trap if Not Equal	*TNE rs,rt* Trap exception occurs if register *rs* is not equal to register *rt*.						

Table 2-17 Exception Immediate Instructions (ISA Extensions)

Instruction	**Format and Description** REGIMM, rs, sub, immediate
Trap if Greater Than or Equal Immediate	*TGEI rs,immediate* Trap exception occurs if register *rs* is greater than or equal to sign-extended 16-bit *immediate*, considering both quantities as signed integers.
Trap if Greater Than or Equal Unsigned Immediate	*TGEIU rs,immediate* Trap exception occurs if register *rs* is greater than or equal to to sign-extended 16-bit *immediate*, considering both quantities as unsigned integers.
Trap if Less Than Immediate	*TLTI rs,immediate* Trap exception occurs if register *rs* is less than to sign-extended 16-bit *immediate*, considering both quantities as signed integers..
Trap if Less Than Unsigned Immediate	*TLTIU rs,immediate* Trap exception occurs if register *rs* is less than to sign-extended 16-bit *immediate*, considering both quantities as unsigned integers.
Trap if Equal Immediate	*TEQI rs,immediate* Trap exception occurs if register *rs* is equal to *immediate*.
Trap if Not Equal Immediate	*TNEI rs,immediate* Trap exception occurs if register *rs* is not equal to *immediate*.

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. Table 2-18 summarizes the coprocessor instructions valid on all MIPS R-Series processors; Table 2-19 summarizes those instructions defined as extensions to the ISA.

Table 2-18 Coprocessor Instruction Summary

<table>
<tr><th>Instruction</th><th>Format and Description</th><th>op</th><th>base</th><th>rt</th><th>offset</th></tr>
<tr><td>Load Word to Coprocessor z</td><td colspan="5"><i>LWCz rt,offset(base)</i>
Sign-extend 16-bit <i>offset</i> and add to contents of register <i>base</i> to form address. Load contents of addressed word into coprocessor register <i>rt</i> of coprocessor unit z.</td></tr>
<tr><td>Store Word from Coprocessor z</td><td colspan="5"><i>SWCz rt,offset(base)</i>
Sign-extend 16-bit <i>offset</i> and add to contents of register <i>base</i> to form address. Store contents of coprocessor register <i>rt</i> from coprocessor unit z at addressed memory word.</td></tr>
</table>

<table>
<tr><th>Instruction</th><th>Format and Description</th><th>COPz</th><th>sub</th><th>rt</th><th>rd</th><th>0</th></tr>
<tr><td>Move To Coprocessor z</td><td colspan="6"><i>MTCz rt,rd</i>
Move contents of CPU register <i>rt</i> into coprocessor register <i>rd</i> of coprocessor unit z.</td></tr>
<tr><td>Move From Coprocessor z</td><td colspan="6"><i>MFCz rt,rd</i>
Move contents of coprocessor register <i>rd</i> of coprocessor unit z into CPU register <i>rt</i>.</td></tr>
<tr><td>Move Control To Coprocessor z</td><td colspan="6"><i>CTCz rt,rd</i>
Move contents of CPU register <i>rt</i> into coprocessor control register <i>rd</i> of coprocessor unit z.</td></tr>
<tr><td>Move Control From Coprocessor z</td><td colspan="6"><i>CFCz rt,rd</i>
Move contents of control register <i>rd</i> of coprocessor unit z into CPU register <i>rt</i>.</td></tr>
</table>

<table>
<tr><th>Instruction</th><th>Format and Description</th><th>COPz</th><th>CO</th><th>cofun</th></tr>
<tr><td>Coprocessor z Operation</td><td colspan="4"><i>COPz cofun</i>
Coprocessor unit z performs an operation. The state of the CPU is not modified by a coprocessor operation.</td></tr>
</table>

<table>
<tr><th>Instruction</th><th>Format and Description</th><th>COPz</th><th>BC</th><th>br</th><th>offset</th></tr>
<tr><td>Branch on Coprocessor z True</td><td colspan="5"><i>BCzT offset</i>
Compute a branch target address by adding the address of the instruction in the delay slot and the 16-bit <i>offset</i> (shifted left two bits and sign extendedly 32 bits). Branch to the target address (with a delay of one instruction) if coprocessor unit z's condition line is true.</td></tr>
<tr><td>Branch on Coprocessor z False</td><td colspan="5"><i>BCzF offset</i>
Compute a branch target address by adding the address of the instruction in the delay slot and the 16-bit <i>offset</i> (shifted left two bits and sign extended to 32 bits). Branch to the target address (with a delay of one instruction) if coprocessor unit z's condition line is false.</td></tr>
</table>

Table 2-19 Coprocessor Instruction Summary (ISA Extensions)

<table>
<tr><th>Instruction</th><th>Format and Description</th><th>COPz</th><th>sub</th><th>rt</th><th>rd</th><th>0</th></tr>
<tr><td>Doubleword Move From Coprocessor z</td><td colspan="6"><i>DMFCz rt,rd</i>
Move contents of coprocessor register <i>rd</i> of coprocessor unit z into CPU register <i>rt</i>.</td></tr>
<tr><td>Doubleword Move To Coprocessor z</td><td colspan="6"><i>DMTCz rt,rd</i>
Move contents of CPU register <i>rt</i> into coprocessor register <i>rd</i> of coprocessor unit z.</td></tr>
</table>

<table>
<tr><th>Instruction</th><th>Format and Description</th><th>op</th><th>base</th><th>rt</th><th>offset</th></tr>
<tr><td>Load Doubleword to Coprocessor z</td><td colspan="5"><i>LDCz rt,offset(base)</i>
Sign-extend 16-bit <i>offset</i> and add to contents of register <i>base</i> to form address. Load contents of addressed doubleword into coprocessor register. <i>rt</i> of coprocessor unit z</td></tr>
<tr><td>Store Doubleword from Coprocessor z</td><td colspan="5"><i>SDCz rt,offset(base)</i>
Sign-extend 16-bit <i>offset</i> and add to contents of register <i>base</i> to form address. Store contents of coprocessor register <i>rt</i> from coprocessor unit z at addressed memory word.</td></tr>
</table>

<table>
<tr><th>Instruction</th><th>Format and Description</th><th>COPz</th><th>BC</th><th>br</th><th>offset</th></tr>
<tr><td>Branch on Coprocessor z True Likely</td><td colspan="5"><i>BCzTL offset</i>
Compute a branch target address by adding the address of the instruction in the delay slot and the 16-bit <i>offset</i> (shifted left two bits and sign extended to 32 bits). Branch to the target address (with a delay of one instruction) if coprocessor unit z condition line is true. If conditional branch is not taken, the instruction in the branch delay slot is nullified.</td></tr>
<tr><td>Branch on Coprocessor z False Likely</td><td colspan="5"><i>BCzFL offset</i>
Compute a branch target address by adding the address of the instruction in the delay slot and the 16-bit <i>offset</i> (shifted left two bits and sign extended to 32 bits). Branch to the target address (with a delay of one instruction) if coprocessor unit z condition line is false. If conditional branch is not taken, the instruction in the branch delay slot is nullified.</td></tr>
</table>

System Control Coprocessor (CP0) Instructions

Coprocessor 0 instructions perform operations on the System Control Coprocessor (CP0) registers to manipulate the memory management and exception handling facilities of the processor. Table 2-20 summarizes the available instructions that work with CP0.

Table 2-20 System Control Coprocessor (CP0) Instruction Summary

<table>
<tr><th>Instruction</th><th>Format and Description</th><th>COP0</th><th>sub</th><th>rt</th><th>rd</th><th>0</th></tr>
<tr><td>Move To CP0</td><td colspan="6"><i>MTC0 rt,rd</i>
Load the contents of CPU register <i>rt</i> into register <i>rd</i> of CP0.</td></tr>
<tr><td>Move From CP0</td><td colspan="6"><i>MFC0 rt,rd</i>
Load the contents of CP0 register <i>rd</i> into CPU register <i>rt</i>.</td></tr>
<tr><th>Instruction</th><th>Format and Description</th><th>COP0</th><th>CO</th><th>0</th><th colspan="2">function</th></tr>
<tr><td>Read Indexed TLB Entry</td><td colspan="6"><i>TLBR</i>
Load <i>EntryHi, EntryLo0,</i> and <i>Entry Lo1</i> registers with TLB entry pointed to by the <i>Index</i> register.</td></tr>
<tr><td>Write Indexed TLB Entry</td><td colspan="6"><i>TLBWI</i>
Load TLB entry pointed to by the <i>Index</i> register with the contents of the <i>EntryHi, EntryLo0,</i> and <i>Entry Lo1</i> registers.</td></tr>
<tr><td>Write Random TLB Entry</td><td colspan="6"><i>TLBWR</i>
Load TLB entry pointed to by the <i>Random</i> register with the contents of the <i>EntryHi, EntryLo0,</i> and <i>Entry Lo1</i> registers.</td></tr>
<tr><td>Probe TLB for Matching Entry</td><td colspan="6"><i>TLBP</i>
Load the <i>Index</i> register with the address of the TLB entry whose contents match the <i>EntryHi, EntryLo0,</i> and <i>Entry Lo1</i> registers. If no TLB entry matches, set the high-order bit of the <i>Index</i> register.</td></tr>
<tr><td>Return from Exception</td><td colspan="6"><i>ERET</i>
Return from exception, interrupt, or error trap.</td></tr>
<tr><th>Instruction</th><th>Format and Description</th><th>CACHE</th><th>base</th><th>op</th><th colspan="2">offset</th></tr>
<tr><td>Cache Operation</td><td colspan="6"><i>CACHE op,offset(base)</i>
Virtual address is formed from addition of <i>offset</i> and <i>base</i>, and this virtual address is translated into a physical address using the TLB. Sub-opcode <i>op</i> specifies a cache operation for this address.</td></tr>
</table>

The R4000 Pipeline

3

This chapter describes the operation of the R4000 instruction execution pipeline. It first describes the basic operation of the pipeline. It then explains how the R4000 handles delay instructions; these are instructions that follow a branch or load instruction in the pipeline. A later section explains interruptions to the pipeline flow caused by interlocks and exceptions.

Basic Pipeline Operation

The R4000 processor has an eight-stage execution pipeline. Each pipeline stage takes one pcycle (one cycle of pclock, which runs at twice the frequency of MasterClock). The execution of each instruction thus takes at least eight pcycles (four MasterClock cycles). An instruction may take longer; for example, when the required data is not in the cache and must be retrieved from main memory. Once the pipeline has been completely filled, eight instructions are always being executed simultaneously.

The eight stages of the R4000 pipeline are listed below and are shown in Figure 3-1.

1. Instruction Fetch, Phase 1 (IF)
2. Instruction Fetch, Phase 2 (IS)
3. Register Fetch (RF)
4. Execution (EX)
5. Data Fetch, Phase 1 (DF)
6. Data Fetch, Phase 2 (DS)
7. Tag Check (TC)
8. Write Back (WB)

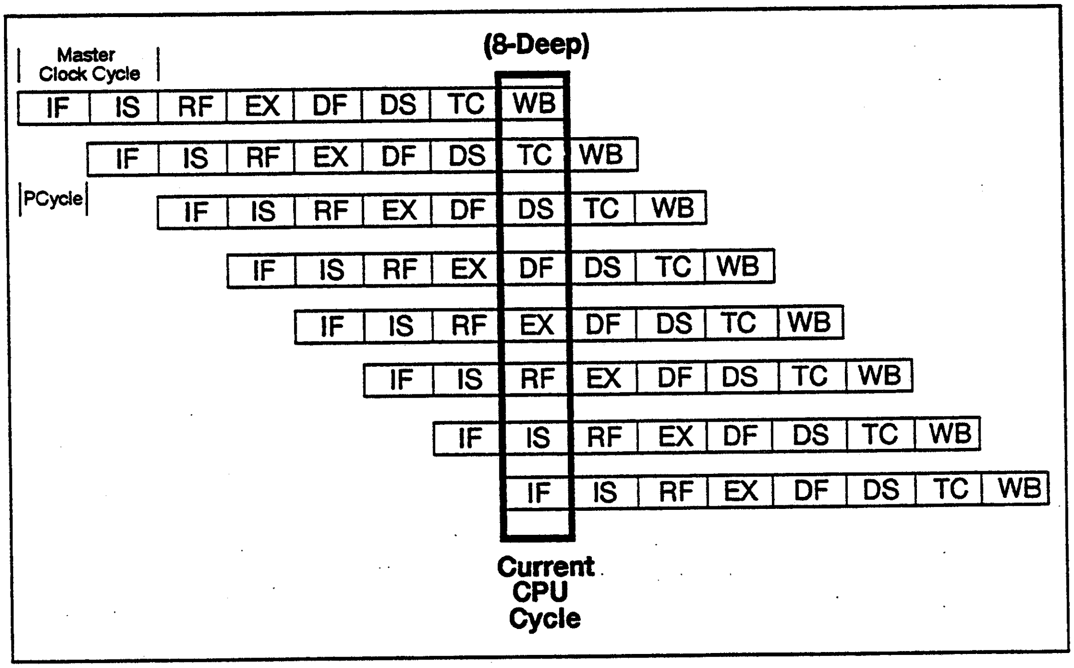
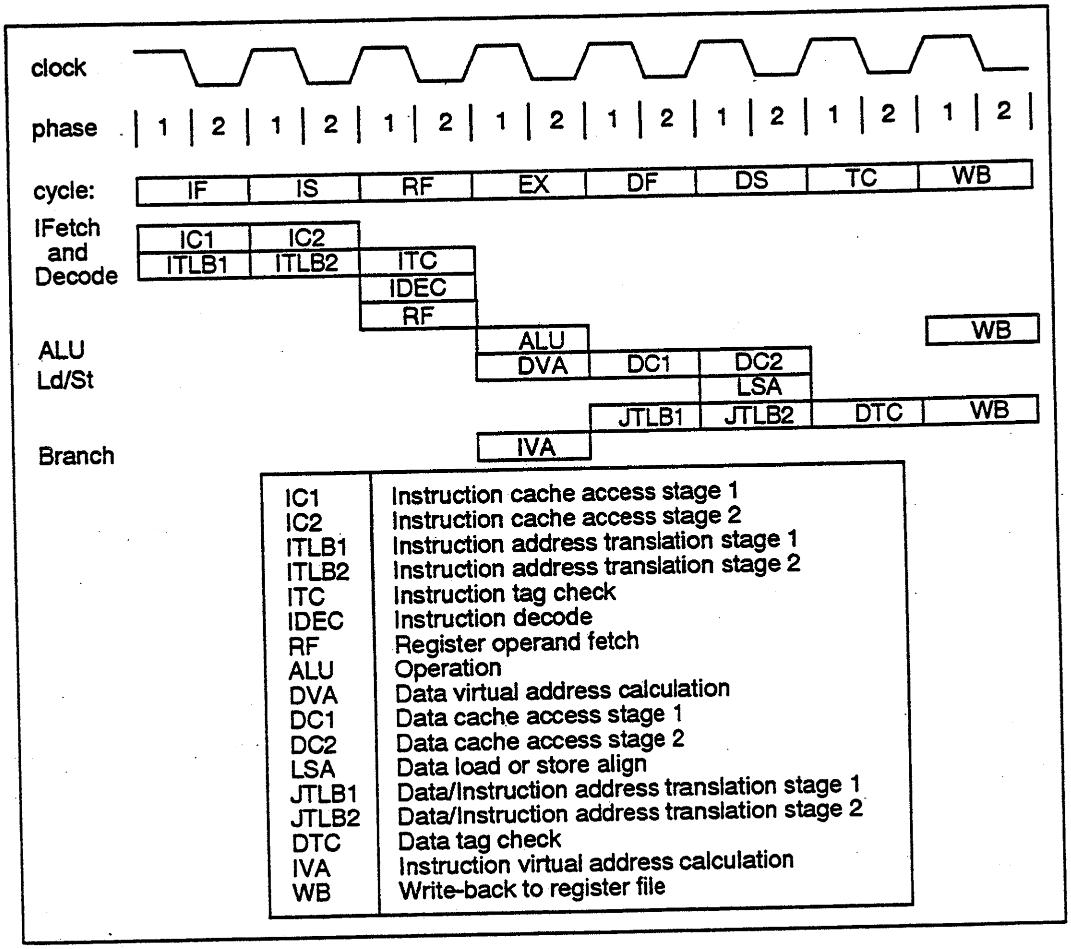


Figure 3-1 R4000 Pipeline and Instruction Overlapping

Figure 3-2 shows the activities occurring during each pipeline stage for ALU, load and store, and branch instructions. The subsections following Figure 3-2 describe the activities during each stage in more detail.



IC1	Instruction cache access stage 1
IC2	Instruction cache access stage 2
ITLB1	Instruction address translation stage 1
ITLB2	Instruction address translation stage 2
ITC	Instruction tag check
IDEC	Instruction decode
RF	Register operand fetch
ALU	Operation
DVA	Data virtual address calculation
DC1	Data cache access stage 1
DC2	Data cache access stage 2
LSA	Data load or store align
JTLB1	Data/Instruction address translation stage 1
JTLB2	Data/Instruction address translation stage 2
DTC	Data tag check
IVA	Instruction virtual address calculation
WB	Write-back to register file

Figure 3-2 R4000 Pipeline Activities

IF - Instruction Fetch, First Half

An instruction address is selected by the branch logic and the instruction cache fetch begins. The Instruction Translation Lookaside Buffer (ITLB) begins the virtual-to-physical address translation.

IS - Instruction Fetch, Second Half

The instruction cache fetch and the virtual-to-physical address translation are completed.

RF - Register Fetch

The instruction decoder (IDEC) decodes the instruction and checks for interlock conditions. The instruction cache tag is checked against the page frame number obtained from the ITLB. Any required operands are fetched from the register file.

EX - Execution

For register-to-register instructions, the ALU performs the arithmetic or logical operation. For load and store instructions, the ALU calculates the data virtual address. For branch instructions, the ALU determines whether the branch condition is true and calculates the virtual branch target address.

DF - Data Fetch, First Half

For load and store instructions, the data cache fetch and the data virtual-to-physical translation begin. For branch instructions, the branch instruction address translation and TLB update begin. Register-to-register instructions perform no operations during the DF, DS, and TC stages.

DS - Data Fetch, Second Half

For load and store instructions, the data cache fetch and data virtual-to-physical translation are completed. The Shifter aligns the data to the word or doubleword boundary. For branch instructions, the branch instruction address translation and TLB update are completed.

TC - Tag Check

For load and store instructions, the cache performs the tag check. The physical address from the TLB is checked against the cache tag to determine if there is a hit or a miss.

WB - Write Back

For register-to-register instructions, the instruction result is written back to the register file. Branch instructions perform no operation during this stage.".

Branch and Load Delay

The more finely grained pipeline of the R4000 results in a branch delay of three cycles and a load delay of two. The branch delay of three is easily observed by noting that the branch comparison logic operates during the EX pipestage of the branch, producing an instruction address which is available for IF stage of the fourth subsequent instruction. The branch delay is illustrated in the Figure 3-3.

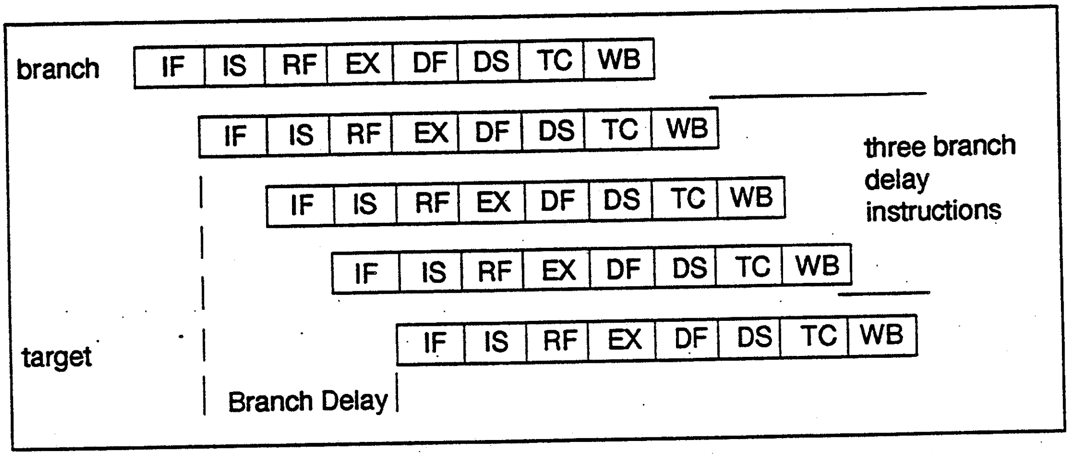


Figure 3-3 R4000 Pipeline Branch Delay

Similarly, the load delay of two is evident in that the completion of a load at the end of the DS pipestage of a load, produces an operand which is available for the EX pipestage of the third subsequent instruction. The load delay is illustrated in the Figure 3-4.

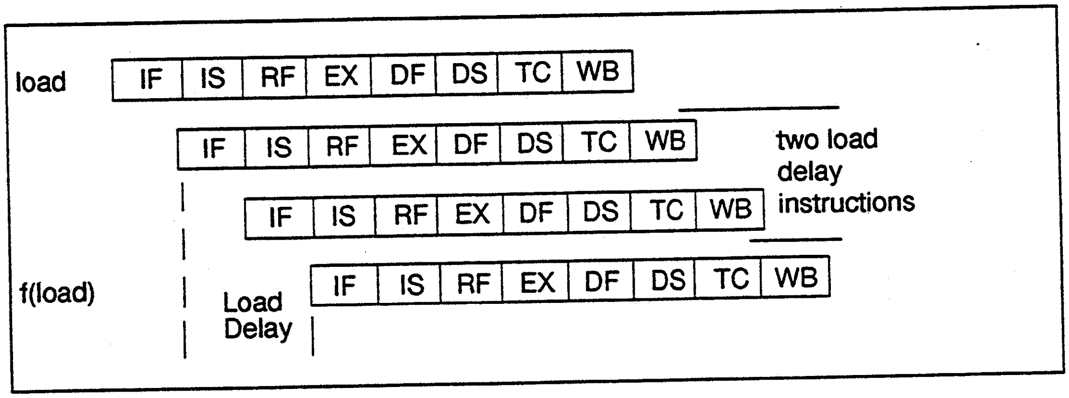


Figure 3-4 R4000 Pipeline Load Delay

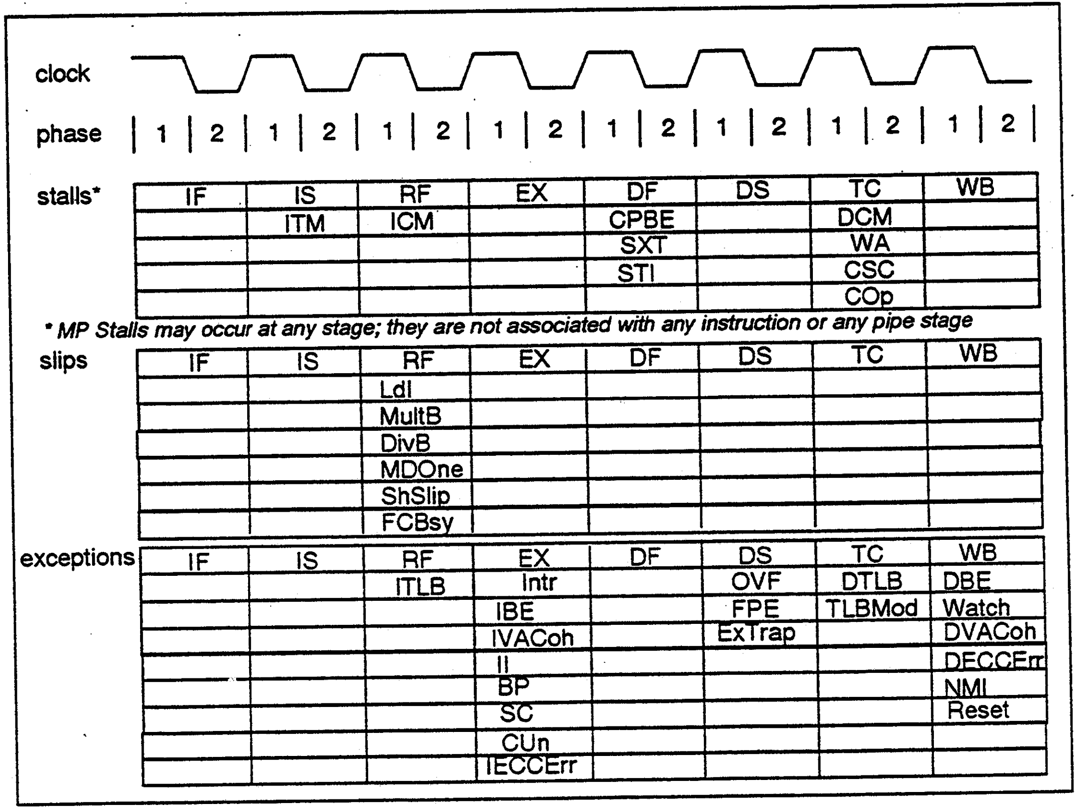
Interlock and Exception Timing

Smooth pipeline flow is interrupted when cache accesses miss, data dependencies are detected, or when exceptions occur. Interruptions that are handled by hardware, such as cache misses, are referred to as *interlocks*, while those that are handled using software are *exceptions*. Collectively, the cases of all interlock and exception conditions are referred to as *faults*.

Interlocks come in two varieties. Those interlocks which are resolved by simply stopping the pipeline are referred to as *stalls*, while those which require part of the pipeline to advance while holding up another part are *slips*.

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, the conditions can be referred back to particular instructions (see Figure 3-5).



phase	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2

stalls*	IF	IS	RF	EX	DF	DS	TC	WB
		ITM	ICM		CPBE		DCM	
					SXT		WA	
					STI		CSC	
							COp	

*** MP Stalls may occur at any stage; they are not associated with any instruction or any pipe stage*

slips	IF	IS	RF	EX	DF	DS	TC	WB
			Ldl					
			MultB					
			DivB					
			MDOne					
			ShSlip					
			FCBsy					

exceptions	IF	IS	RF	EX	DF	DS	TC	WB
			ITLB	Intr		OVF	DTLB	DBE
				IBE		FPE	TLBMod	Watch
				IVACoh		ExTrap		DVACoh
				II				DECCErr
				BP				NMI
				SC				Reset
				CUn				
				IECCErr				

Figure 3-5 Correspondence of Pipeline Stage to Interlock Condition

Figure 3-5 shows the correlation between the interlocks and exceptions shown in Figure 3-5.

Table 3-1 Correspondence of Pipeline Stage to Interlock Condition

Interlocks		Exceptions	
ITM	Instruction TLB Miss	ITLB	Instruction Translation or Address Exception
ICM	Instruction Cache Miss	Intr	External Interrupt
CPBE	Coprocessor Possible Exception	IBE	IBus Error
SXT	Integer Sign Extend	IVACoh	IVA Coherent
STI	Store Interlock	II	Illegal Instruction
DCM	Data Cache Miss	BP	Breakpoint
WA	Watch Address Exception	SC	System Call
LDI	Load Interlock	CUn	Coprocessor Unuseable
MultB	Multiply Unit Busy	IECCErr	Instruction ECC Error
DivB	Divide Unit Busy	OVF	Integer Overflow
MDOne	Mult/Div One Cycle Slip	FPE	FP Interrupt
ShSlip	Var Shift or Shift > 32 bits	ExTrap	EX Stage Traps
FCBsy	FP Busy	DTLB	Data Translation or Address Exception
		TLBMod	TLB Modified
		DBE	Data Bus Error
		Watch	Memory Reference Address Compare
		DVACoh	DVA Coherent
		DECCErr	Data ECC Error
		NMI	Non-maskable Interrupt
		Reset	Reset

When an exception condition occurs, the relevant instruction and all that follow it in the pipeline are cancelled. Accordingly, any stall conditions and any later exception conditions that are referenced to the same instruction are inhibited; there is no value in servicing stalls for a cancelled instruction. A new instruction stream is begun, starting execution at a predefined exception vector. System control coprocessor registers are loaded with information that will identify the type of exception and any necessary auxiliary information, such as the virtual address at which translation exceptions occur.

When a stall condition is detected, all eight instructions, each in a different stage of the pipeline, are frozen at once. Often, the stall condition is only detected after parts of the pipeline have advanced using incorrect data; we refer to this occurrence as pipeline overrun. When in the stalled state, no pipeline stages advance until the interlock condition is resolved. After the interlock is removed, the restart sequence begins two cycles before resuming execution. The restart sequence reverses the pipeline overrun condition by inserting the correct information into the pipeline.

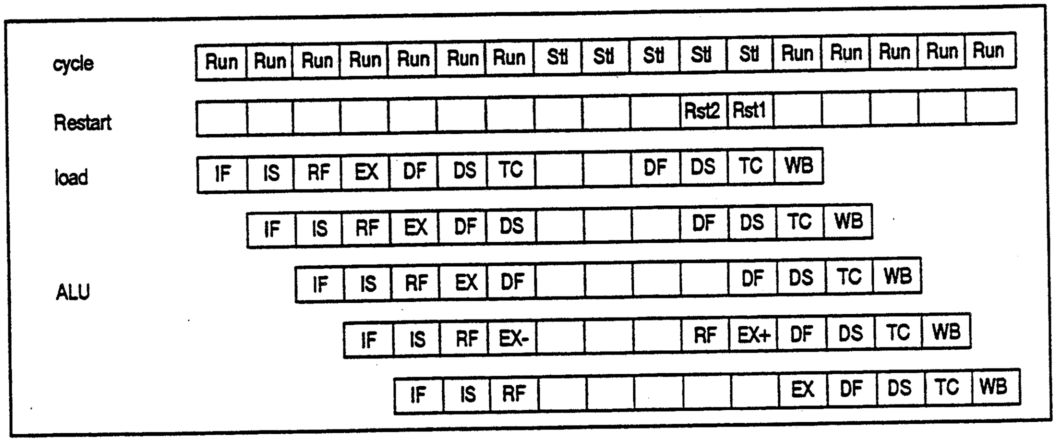
When a slip condition is detected, the pipeline stages which must advance in order to resolve the dependency continue to be retired while the dependent stages are held until the necessary data is available.

Another class of interlocks exists which, since they originate external to the processor, are not referenced to a particular pipeline stage. These interlocks are referred to as *external* stalls and are unaffected by the occurrence of exceptions.

In order to prevent interlock and exception handling from adversely affecting the processor cycle time, the R4000 uses both logic and circuit pipelining techniques to reduce critical timing paths. Logical pipelining of interlock and exception handling has the following two principal effects:

1. the processor pipeline must be backed up in some cases to recover from interlocks, and
2. in some cases interlocks will be serviced for instructions which will be aborted due to an exception.

An example of the former happens in the case of data cache misses, where the late detection of the miss causes a subsequent instruction to compute an incorrect result. Not only must the cache miss be serviced but the EX stage of the dependent instruction must be redone before the pipeline can be restarted. Figure 3-6 below illustrates this phenomena. A minus (-) following a pipestage descriptor indicates that the operation performed produced an incorrect result while a plus (+) indicates the successful re-execution of that operation.



cycle	Run	Run	Run	Run	Run	Run	Run	Stl	Stl	Stl	Stl	Stl	Run	Run	Run	Run	Run
Restart											Rst2	Rst1					
load	IF	IS	RF	EX	DF	DS	TC			DF	DS	TC	WB				
		IF	IS	RF	EX	DF	DS				DF	DS	TC	WB			
ALU			IF	IS	RF	EX	DF					DF	DS	TC	WB		
				IF	IS	RF	EX-				RF	EX+	DF	DS	TC	WB	
					IF	IS	RF						EX	DF	DS	TC	WB

Figure 3-6 Pipeline Overrun

An example of a case in which interlocks are serviced for instructions which will subsequently be aborted is the interaction between integer overflow and instruction cache miss. In this case, pipelining the overflow exception handling into the DF pipestage will allow an instruction cache miss to occur on the immediately subsequent instruction. This is illustrated in Figure 3-7. Aborted instructions are denoted with an asterisk (*).

Ignoring the fact that the line brought in by the instruction cache could be replaced by a line of the exception handler, it can be argued that no performance loss occurs since the instruction cache miss would have otherwise been serviced after returning from the exception handler anyway. A more legitimate argument for handling the exception in this fashion however is that the frequency of exceptions is relatively low by definition. If this were not the case the processor would spend most of its time in the exception handler and no progress would be made.

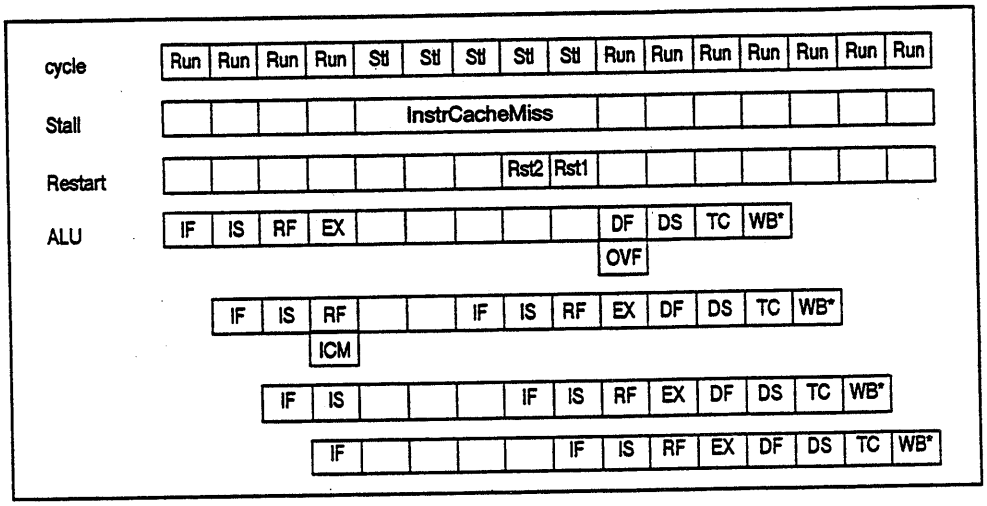


Figure 3-7 Instruction Cache Miss

Circuit pipelining of interlock and exception handling is accomplished by pipelining the logical resolution of the possible fault conditions with the buffering and distribution of the pipeline control signals. In particular, a half clock period is provided for the buffering and distribution of the *run* control signal and during this time the logic evaluation to produce run for the next cycle is begun. This process is illustrated in Figure 3-8 for a sequence of loads.

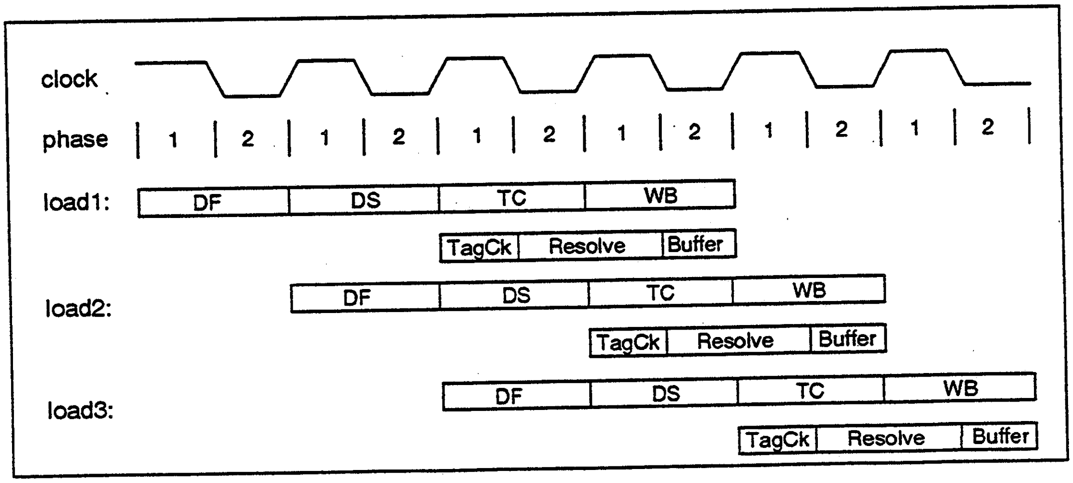


Figure 3-8 Circuit Pipelining of Interlock and Exception Handling

In the general case, resolving whether or not the pipeline advances in any particular cycle is done in the same three step sequence:

1. Individually evaluate all possible fault causing events such as cache misses, translation exceptions, load interlocks, etc.
2. Resolve which fault is to be serviced based on a predefined priority determined by the pipestage of the asserted faults.
3. Buffer and distribute the pipeline advance control signals.

This process is illustrated in Figure 3-9.

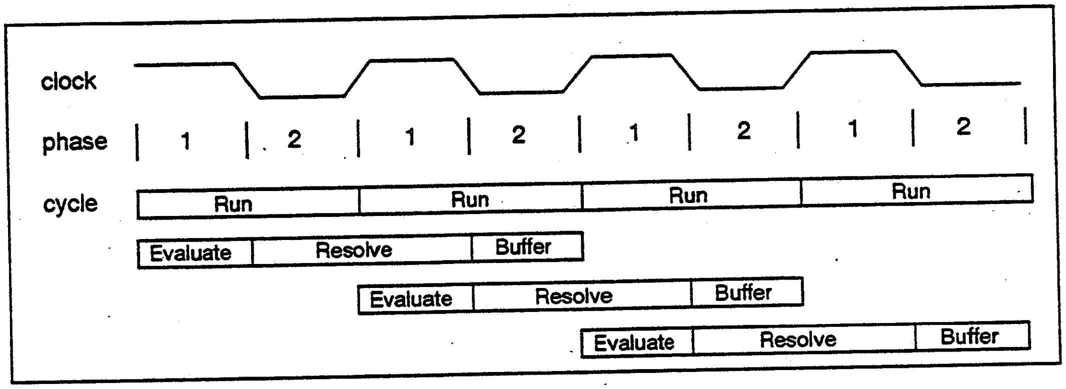


Figure 3-9 Pipeline Advance Resolution

Special Cases

In some instances, the pipeline control state machine is bypassed. This bypassing occurs due to either:

- performance considerations, or
- correctness considerations.

An example of the former occurs in the case of cache misses on loads. By bypassing the pipeline state machine in this instance it is possible to eliminate up to two cycles from the load miss latency. In this case, it is relatively straightforward to perform the bypass since sending the cache miss address to the secondary cache has no negative impact even if an exception later nullifies the effect of the cache access. The bypassing of the potential cache miss address is referred to as address acceleration. It is noted that an argument could be put forward that some power is wasted when the miss is inhibited by some fault, but this will be a minor effect. Another technique used in the R4000 to reduce miss latency is the automatic increment and driveout of instruction miss addresses following an instruction cache miss. This form of latency reduction is referred to as address prediction as the subsequent instruction miss address is being predicted to be a simple increment of the previous miss address. Figure 3-10 illustrates a cache miss where the cache miss address is shown changing based simply on detection of the miss.

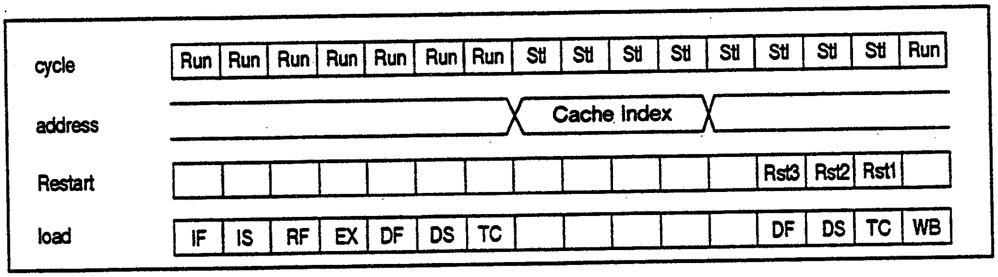


Figure 3-10 Load Address Bypassing

An example of a case where bypassing is necessary to guarantee correctness is cache writes.

Memory Management System

4

The MIPS R4000 processor provides a full-featured memory management unit (MMU) that uses an on-chip Translation Lookaside Buffer (TLB) to translate virtual addresses into physical addresses.

The MMU provides very fast virtual memory translation. This chapter describes the operation of the TLB and the CP0 registers that provide the software interface to the TLB. The memory mapping scheme, which translates virtual addresses to physical addresses, is also described in detail.

Memory System Architecture

The virtual memory system extends the address space available to programs by translating addresses composed in a large virtual address space into physical memory space.

The R4000 physical address space is 64 Gigabytes using a 36-bit address. The virutal address is either 64 or 32 bits wide depending on whether the processor is operating is 32- or 64-bit mode. In 32-bit mode, addresses are 32-bits wide and the maximum user process size is 2 Gigabytes (2^{31}). In 64-bit mode, addresses are 64-bits wide and the maximum user process is 1 Terabyte (2^{40})

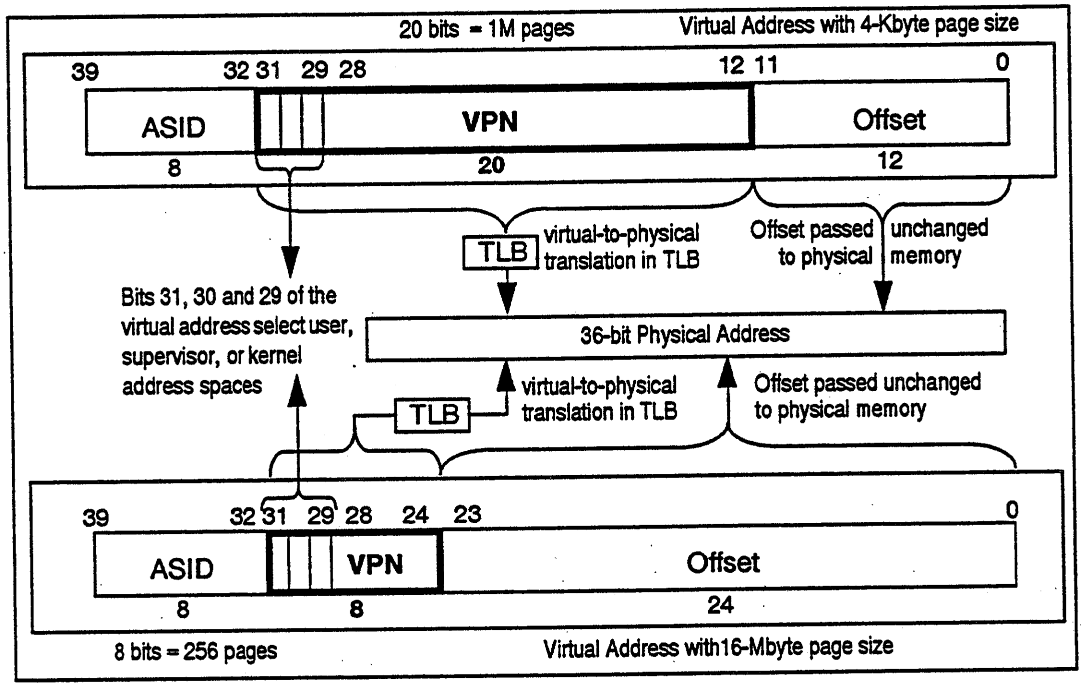

20 bits = 1M pages
Virtual Address with 4-Kbyte page size
39
32 31 29 28
12 11
0
ASID
VPN
Offset
8
20
12
TLB
virtual-to-physical translation in TLB
Offset passed to physical
unchanged memory
Bits 31, 30 and 29 of the virtual address select user, supervisor, or kernel address spaces
36-bit Physical Address
TLB
virtual-to-physical translation in 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 with16-Mbyte page size

Figure 4-1 R4000 32-bit Virtual Address Format

The virtual address is extended with an Address Space Identifier (ASID) to reduce the frequency of TLB flushing when switching context. The size of the ASID field is 8 bits. The ASID is contained in the CP0 *EntryHi* register. The CP0 *EntryHi* register is described in this chapter.

Operating Modes

This section describes the three operating modes of the R4000 for 32- and 64-bit operation:

- User mode
- Supervisor mode
- Kernel mode

Two of these modes are provided by all MIPS R-Series processors: Kernel mode, which is analogous to the "supervisory" mode provided by many machines, and User mode, in which nonsupervisory programs are executed. The R4000 provides a third, intermediate mode, called Supervisor mode. This mode can be used to more easily build secure operating systems.

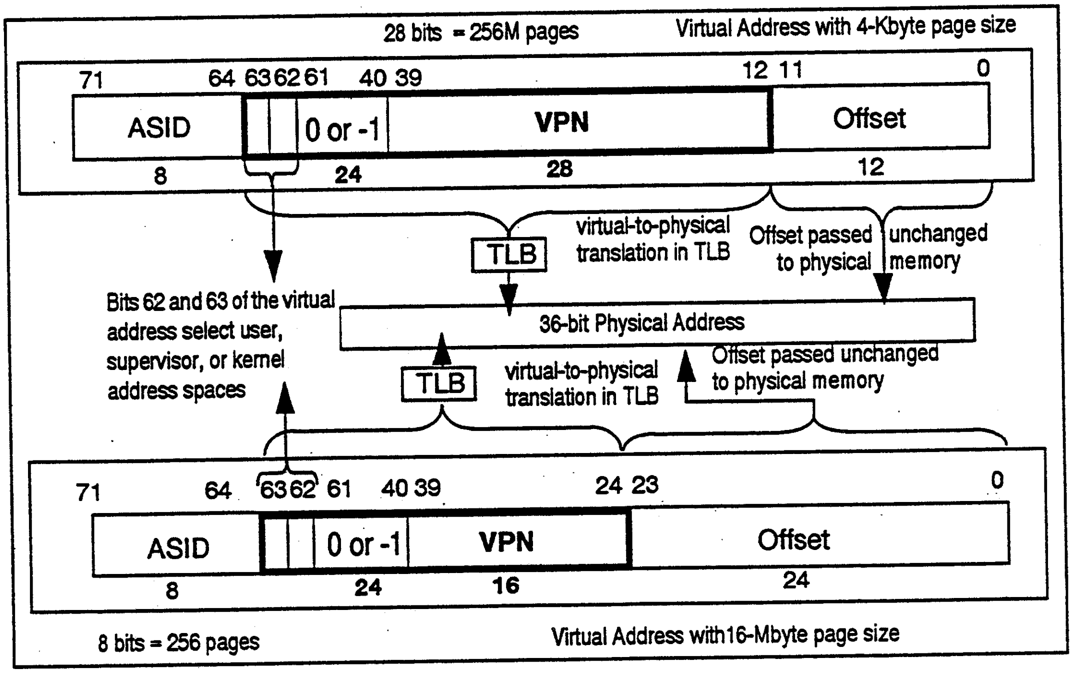

28 bits = 256M pages
Virtual Address with 4-Kbyte page size
71
64
63
62
61
40
39
12
11
0
ASID
0 or -1
VPN
Offset
8
24
28
12
TLB
virtual-to-physical translation in TLB
Offset passed unchanged to physical memory
Bits 62 and 63 of the virtual address select user, supervisor, or kernel address spaces
36-bit Physical Address
TLB
virtual-to-physical translation in 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
8 bits = 256 pages
Virtual Address with16-Mbyte page size

Figure 4-2 R4000 64-bit Virtual Address Format

The CPU 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 taking the exception.

User Mode Virtual Addressing

In User mode, a single, uniform virtual address space (*useg*) of 2 GBytes (2^{31} bytes) in 32-bit mode or 1 Terabyte (2^{40} bytes) in 64-bit mode is available, as shown in Figure 4-3. Figure 4-1 and Figure 4-2 show that the virtual address is extended with an 8-bit Address Space Identifier (ASID) field during virtual to physical address translation to form unique virtual addresses for up to 256 user processes. By assigning each process an ASID, the system is able to maintain the TLB contents across context switches. All references to *useg* are mapped through the TLB, and the cacheability of a reference is determined by bit settings within the TLB entry for the page.

The User segment starts at address zero, 0x0000 0000. The TLB maps all references to *useg* identically from all modes, and controls cache accessibility. (The C-bits in a TLB entry determine whether the reference is cached; see Figure 4-1.) The current user process resides in *useg*. Figure 4-3 shows User mode address space.

When bits *KSU* equals 10, bit *EXL* equals 0, and bit *ERL* equals 0 in the *Status* register (see Chapter 5 for a description of the *Status* register), the processor is executing in User mode. The *UX* bit in the *Status* register selects 32- or 64-bit addressing.

- *useg*. When *UX* = 0 in the Status Register, user-mode addressing is compatible with 32-bit addressing shown in Figure 4-3. All valid User mode virtual addresses have the most-significant bit cleared to 0; any attempt to reference an address with the most-significant bit set while in the User mode causes an Address Error exception. (See Chapter 5). The TLB refill exception vector is used for TLB misses.
- *xuseg*. When *UX* =1 in the Status Register, user-mode addressing is exteneded to 64-bit addressing shown in Figure 4-3. The R4000 provides a single, uniform address space of 2^{40} bytes for user processes. All valid user-mode virtual addresses have bits 63..40 equal to zero; an attempt to reference an address with bits 63..40 not equal to zero causes an Address Error exception (See Chapter 5). The Extended addressing TLB refill exception vector is used for TLB misses.

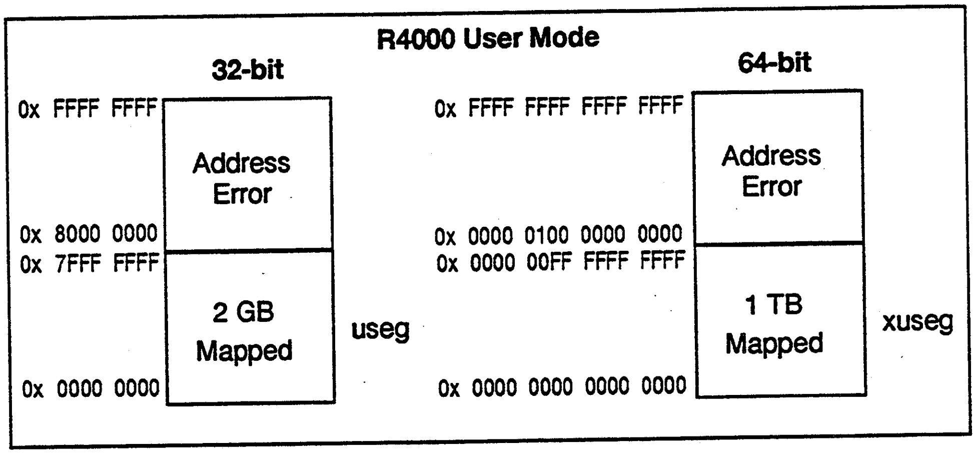


Figure 4-3 MIPS User Mode Virtual Address Space

Supervisor Mode Virtual Addressing

In the following discussion, please refer to Figure 4-4. Supervisor mode is intended for those layered operating system implementations where a "true kernel" runs in R4000 Kernel mode, and the rest of the operating system runs in Supervisor mode. When bits *KSU* equals 01, bit *EXL* equals 0, and bit *ERL* equals 0 in the *Status* register (see Chapter 5 for a description of the *Status* register), the processor is executing in Supervisor mode. The *SX* bit in the *Status* register selects 32- or 64-bit addressing.

- *suseg*. When *SX* = 0 in the *Status* register and the most-significant bit of the 32-bit virtual address is set to 0, the virtual address space, named *suseg*, covers the full 2^{31} bytes (2 Gbytes) of the current user address space. The virtual address is extended with the contents of the ASID field to form unique virtual addresses. This mapped space starts at virtual address 0x0000 0000 and runs up through 0x7FFF FFFF.
- *sseg*. When *SX* = 0 in the *Status* register and the most-significant three bits of the 32-bit virtual address are 110, the virtual address space selected is the current 2^{29}-byte (512-Mbyte) supervisor virtual space labelled *sseg*. The virtual address is extended with the contents of the ASID field to form unique virtual addresses. This mapped space begins at virtual address 0xC000 0000 and runs up through 0xDFFF FFFF.

- *xsuseg*. When *SX* = 1 in the *Status* register and bits 63..62 of the virtual address are set to 00, the virtual address space, named *xsuseg*, covers the full 2^{40} bytes (1 Terabyte) of the current user address space. The virtual address is extended with the contents of the ASID field to form unique virtual addresses. This mapped space starts at virtual address 0x0000 0000 0000 0000 and runs up through 0x0000 00FF FFFF FFFF.
- *xsseg*. When *SX* = 1 in the *Status* register and bits 63..62 of the virtual address are set to 01, the virtual address space selected is the current supervisor virtual space labelled *xsseg*. The virtual address is extended with the contents of the ASID field to form unique virtual addresses. This mapped space begins at virtual address 0x4000 0000 0000 0000 and runs up through 0x4000 00FF FFFF FFFF.
- *csseg*. When *SX* = 1 in the *Status* register and bits 63..62 of the virtual address are set to 11, the virtual address space selected is the current supervisor virtual space labelled *csseg*. Addressing of the *csseg* is compatible with supervisor addressing in 32-bit mode. This mapped space begins at virtual address 0xFFFF FFFF C000 0000 and runs up through 0xFFFF FFFF DFFF FFFF.

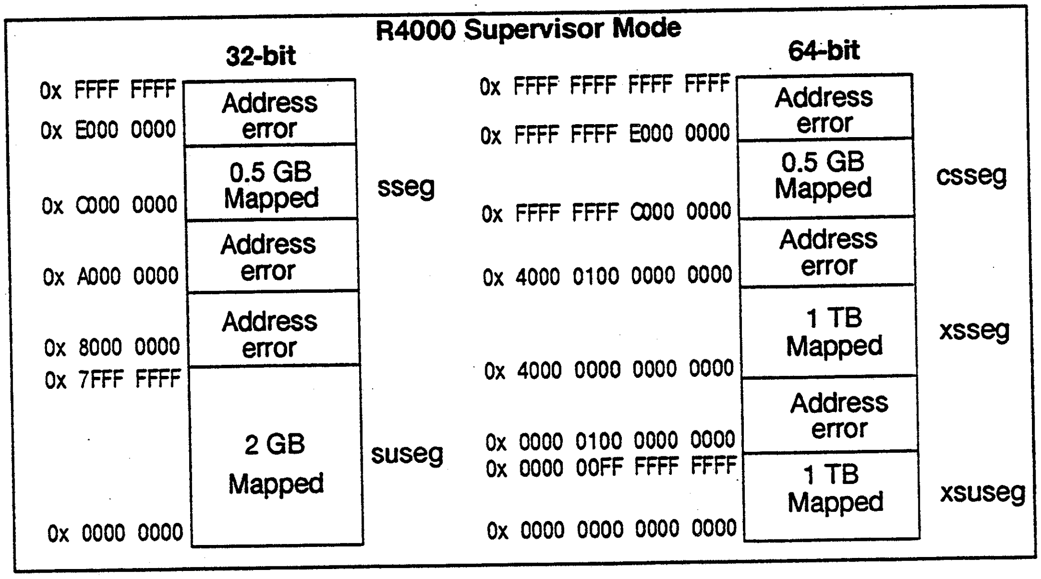


Figure 4-4 MIPS R4000 Supervisor Mode Address Space

Kernel Mode Virtual Addressing

When the processor is operating in Kernel mode (bits *KSU* equals 00, or bit *EXL* equals 1, or bit *ERL* equals 1, in the *Status* register) the virtual address space is divided into regions, differentiated by the high-order bits of the virtual address. The *KX* bit in the *Status* register selects 32- or 64-bit addressing:

- *kuseg*. When *KX* = 0 in the *Status* register and the most-significant bit of the virtual address is cleared, the 32-bit virtual address space selected covers the full 2^{31} bytes (2 Gbytes) of the current user address space labelled *kuseg*. The virtual address is extended with the contents of the ASID field to form unique virtual addresses.
- *kseg0*. When *KX* = 0 in the *Status* register and the most-significant three bits of the virtual address are 100, the 32-bit virtual address space selected is the current 2^{29}-byte (512-Mbyte) kernel physical space labelled *kseg0*. References to *kseg0* are not mapped through the TLB; the physical address selected is defined by subtracting 0x8000 0000 from the virtual address. Cacheability and coherency are controlled by the K0 field of the Config register described in Chapter 5 *Exception Processing*.
- *kseg1*. When *KX* = 0 in the *Status* register and the most-significant three bits of the 32-bit virtual address are 101, the virtual address space selected is the current 2^{29}-byte (512-Mbyte) kernel physical space labelled *kseg1*. 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.
- *ksseg*. When *KX* = 0 in the *Status* register and the most-significant three bits of the 32-bit virtual address are 110, the virtual address space selected is the current 2^{29}-byte (512-Mbyte) supervisor virtual space labelled *ksseg*. The virtual address is extended with the contents of the ASID field to form unique virtual addresses.
- *kseg3*. When *KX* = 0 in the *Status* register and the most-significant three bits of the 32-bit virtual address are 111, the virtual address space selected is the current 2^{29}-byte

(512-Mbyte) kernel virtual space labelled *kseg3*. The virtual address is extended with the contents of the ASID field to form unique virtual addresses.

- *xkuseg*. When *KX* = 1 in the *Status* register and bits 63..62 of the 64-bit virtual address are 00, the virtual address space selected covers the current user address space labelled *xkuseg*. The virtual address is extended with the contents of the ASID field to form unique virtual addresses. As a special feature for the ECC handler, if the *ERL* bit of the *Status* register is set, the user address region becomes a 2^{31}-byte unmapped, uncached space. The allows the ECC exception code to operate uncached using R0 as a base register.
- *xksseg*. When *KX* = 1 in the *Status* register and bits 63..62 of the 64-bit virtual address are 01, the virtual address space selected is the current supervisor virtual space labelled *xksseg*. The virtual address is extended with the contents of the ASID field to form unique virtual addresses.
- *xkphys*. When *KX* = 1 in the *Status* register and bits 63..62 of the 64-bit virtual address are 10, the virtual address space selected is a set of eight 2^{36}-byte kernel physical spaces labelled *xkphys*. Addresses with bits 58..36 not equal to zero cause an address error. References to this space are not mapped; the physical address selected is taken directly from bits 35..0 of the virtual address. The cachebility and coherence algorithm is specified by bits 61..59 of the virtual address (see *EntryLo* for the cache algorithm values).

Value	Cache Algorithm	Starting Address
0	reserved	0x8000 0000 0000 0000
1	reserved	0x8800 0000 0000 0000
2	uncached	0x9000 0000 0000 0000
3	cacheable, non-coherent	0x9800 0000 0000 0000
4	cacheable, coherent exclusive	0xA000 0000 0000 0000
5	cacheable, coherent exclusive on write	0xA800 0000 0000 0000
6	cacheable, coherent update on write	0xB000 0000 0000 0000
7	reserved	0xB800 0000 0000 0000

- *xkseg*. When *KX* = 1 in the *Status* register and bits 63..62 of the 64-bit virtual address are 11, the virtual address space selected is the current supervisor virtual space labelled *xkseg*. The virtual address is extended with the contents of the ASID field to form unique virtual addresses.
- *ckseg0*. When *KX* = 1 in the *Status* register and bits 63..62 of the 64-bit virtual address are 11, the 64-bit virtual address space selected is an unmapped region compatible with the 32-bit address model *kseg0* when bits 61..31 of the virtual address equal -1. Cacheability and coherency are controlled by the K0 field of the Config register described in Chapter 5 *Exception Processing*.
- *ckseg1*. When *KX* = 1 in the *Status* register and bits 63..62 of the 64-bit virtual address are 11, the 64-bit virtual address space selected is an unmapped and uncached region compatible with the 32-bit address model *kseg1* when bits 61..31 of the virtual address equal -1.
- *cksseg*. When *KX* = 1 in the *Status* register and bits 63..62 of the 64-bit virtual address are 11, the 64-bit virtual address space selected is the current supervisor virtual space compatible with the 32-bit address model *ksseg* when bits 61..31 of the virtual address equal -1.
- *ckseg3*. When *KX* = 1 in the *Status* register and bits 63..62 of the 64-bit virtual address are 11, the 64-bit virtual address space selected is kernel virtual space compatible with the 32-bit address model *kseg3* when bits 61..31 of the virtual address equal -1.

Figure 4-5 shows the boundaries of the segments defined in this mode.

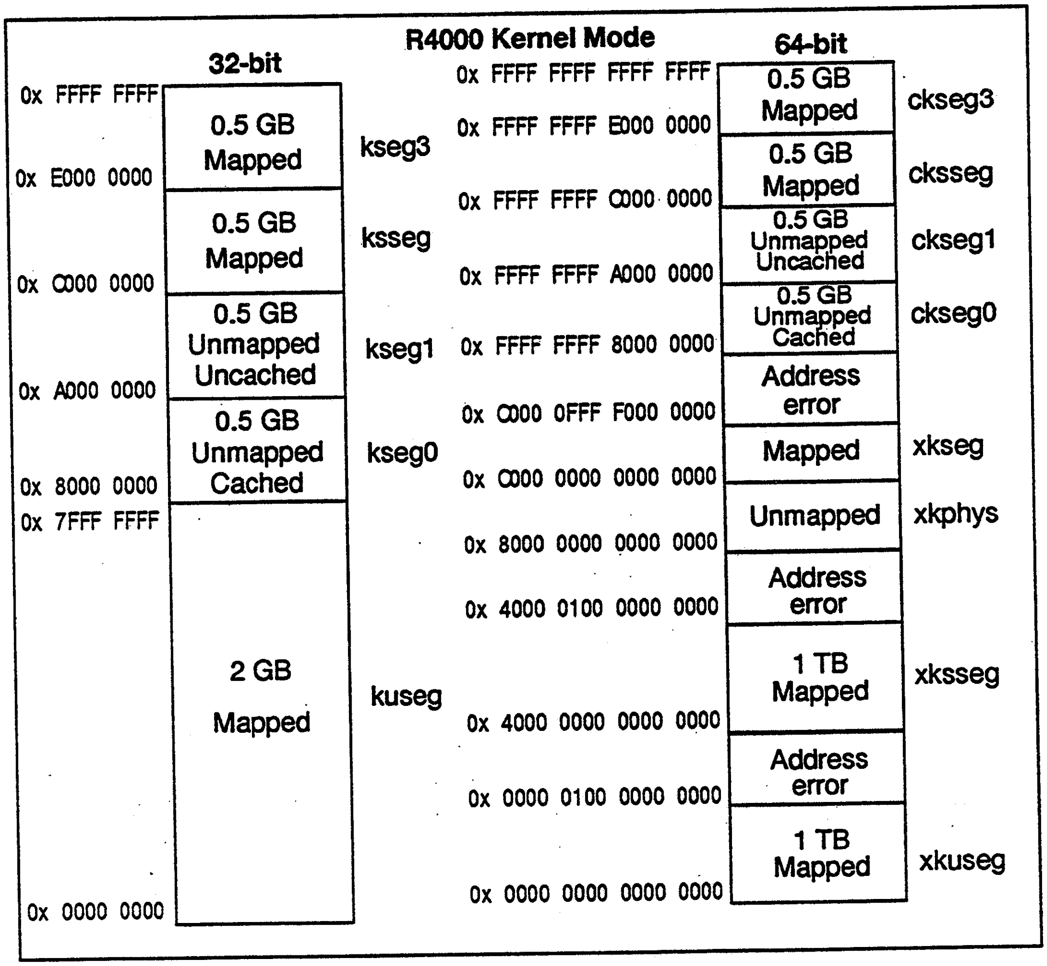


Figure 4-5 MIPS R4000 Kernel Mode Address Space

Virtual Memory and the TLB

Mapped virtual addresses are translated into physical addresses using an on-chip TLB. The TLB is a fully-associative memory that holds 48 entries that provide mapping to 48 odd/even page pairs.The address range mapped by a page can range in size from 4 Kbytes to 16 Mbytes (increasing by multiples of 4: i.e., 4K, 16K, 64K, 256K, 1M, 4M, 16M). When address mapping is indicated, each TLB entry is simultaneously checked for a match with the virtual address extended by the current ASID stored in the *EntryHi* register.

If there is a match (a hit), the physical page number is extracted from the TLB and concatenated with the offset to form the physical address.

If no match occurs (a miss), an exception is taken and software refills the TLB from a Page Table resident in memory. Software can write over a selected TLB entry or use a hardware mechanism to write into a random entry.

If more than one entry in the TLB matches the virtual address being translated, the operation is undefined and the TLB may be shut down. The *TLB-Shutdown (TS)* bit in the *Status* register is set to 1 if the TLB is disabled.

System Control Coprocessor

The system control coprocessor (CP0) is implemented as an integral part of the CPU. CP0 supports address translation, exception handling, and other privileged operations. CP0 also contains the registers shown in Figure 4-6 plus a 48-entry TLB. The sections that follow describe how each of the TLB-related registers are used.

NOTE: CP0 functions and registers associated with exception handling are described in Chapter 5, *Exception Processing*.

The numeral accompanying each register refers to the register number, as described in Chapter 2, *CPU Instruction Set Summary*.

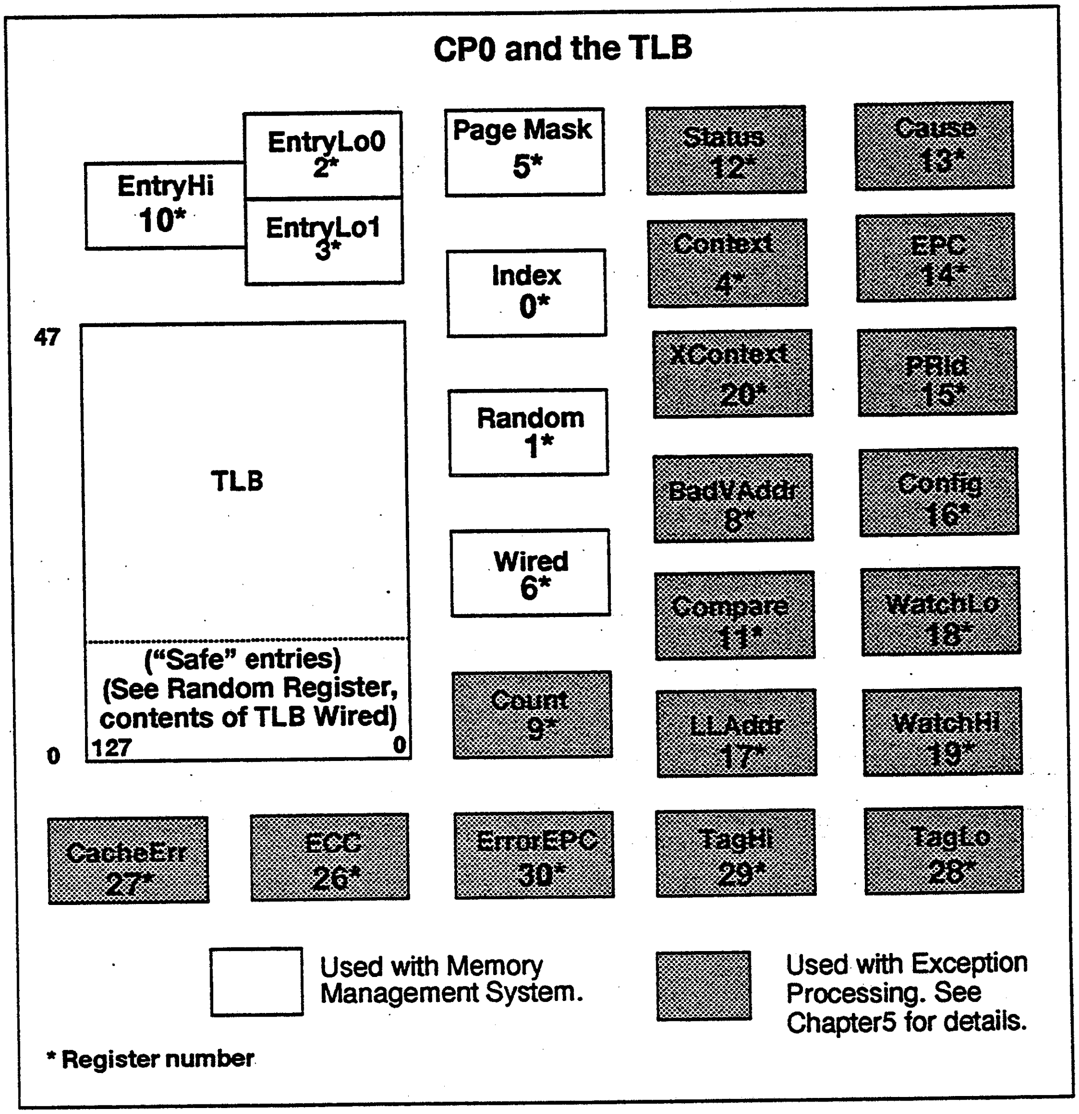


Figure 4-6 The R4000 CP0 Registers and the TLB

TLB Entry Format

Figure 4-7 shows the TLB entry format for 32- and 64-bit addressing. Each field of an entry has a corresponding field in the *EntryHi, EntryLo0, EntryLo1*, or *PageMask* registers, as shown in Figure 4-8.

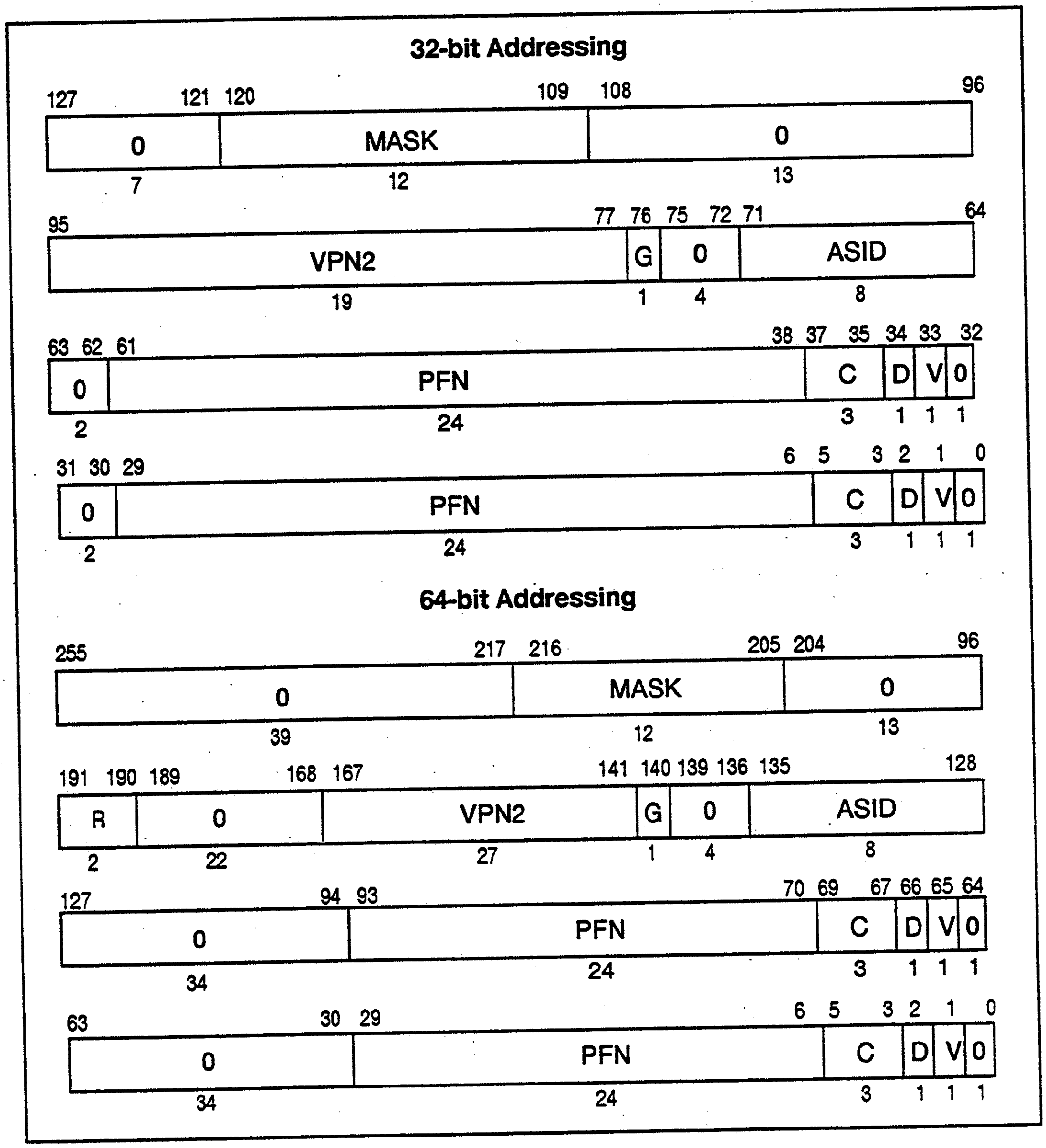
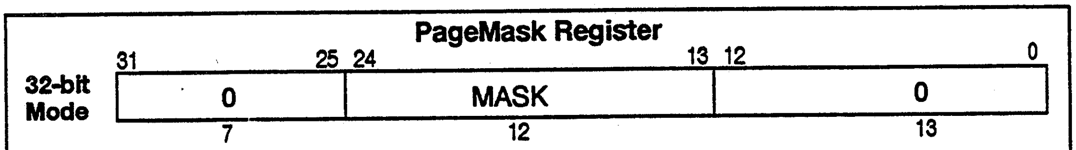
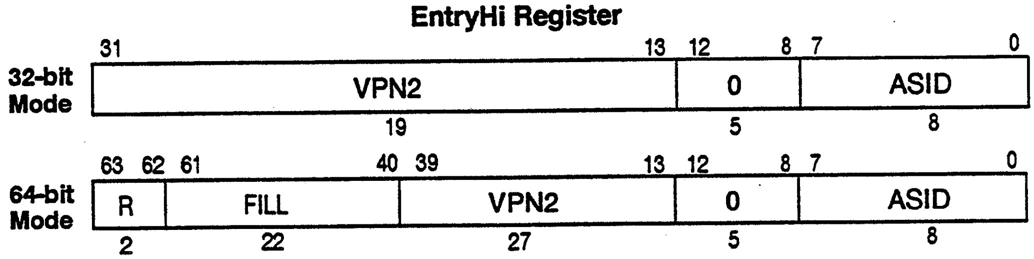


Figure 4-7 Format of an R4000 TLB Entry

The format of the *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers are nearly the same as the 48-bit TLB entry. There is one exception, the TLB does use the *Global* field (bit 76) which is reserved in the *EntryHi* register.



MASK Page comparison mask

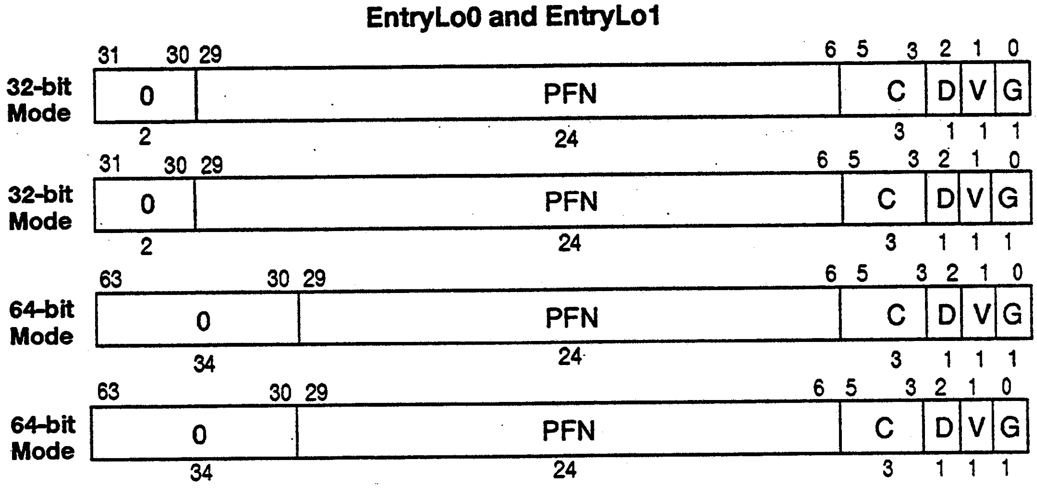


VPN2 Virtual Page Number divided by two (maps to two pages).

ASID Address Space ID field. An 8-bit field which lets multiple processes share the TLB while each process has a distinct mapping of otherwise identical virtual page numbers. This is the same ASID described at the beginning of this chapter.

R Region. (00 → user, 01 → supervisor, 11 → kernel) used to match vAddr$_{63..62}$.

Fill Reserved. Must be the same as bit 63 of the register when written.



PFN Page Frame Number. Upper bits of the physical address.

C Specifies the cache algorithm to be used; see Table 4-1.

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 ignore the ASID during TLB lookup.

0 Reserved. Must be written as zeroes, returns zeroes when read.

Figure 4-8 Fields of an R4000 TLB Entry Registers

The cache algorithm (C) bits specify whether references to the page should be cached; if cached, the algorithm selects between several cache coherency algorithms. Table 4-1 shows the algorithms selected by decoding the C bits.

Table 4-1 Cache Algorithmn Bit Values

C Bit Value	Algorithm
0	reserved
1	reserved
2	uncached
3	cacheable noncoherent (noncoherent)
4	cacheable coherent exclusive (exclusive)
5	cacheable coherent exclusive on write (sharable)
6	cacheable coherent update on write (update)
7	reserved

EntryHi, EntryLo0, EntryLo1, and PageMask Registers

These registers provide the data pathway through which the TLB is read, written, or probed. When address translation exceptions occur, these registers are loaded with relevant information about the address that caused the exception.

EntryHi Register (CPU Register 10)

The *EntryHi* register is a read/write register used to access the TLB. In addition, the *EntryHi* register contains the current ASID value for the processor. This is used to match the virtual address with a TLB entry when virtual addresses are presented for translation.

The *EntryHi* register holds the contents of the high-order bits of a TLB entry when performing TLB read and write operations. When either a TLB refill, TLB invalid, or TLB modified exception occurs, the *EntryHi* register is loaded with the Virtual Page Number (VPN) and the ASID of the virtual address that failed to have a matching TLB entry. For more information on TLB exceptions, see Chapter 5, *Exception Processing.*

EntryHi is accessed by the TLBP, TLBW, TLBWI, and TLBR instructions. Figure 4-8 shows the format of this register.

EntryLo0 (2), and EntryLo1 (3) Registers

EntryLo consists of two registers: *EntryLo0* for even virtual pages and *EntryLo1* for odd virtual pages. The *EntryLo0* and *EntryLo1* registers are read/write registers used to access a TLB. *EntryLo0* and *EntryLo1* 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 4-8 shows the format of these registers.

PageMask Register (5)

The *PageMask* register is a read/write register for reading from or writing to the TLB; it implements a variable page size by holding a per-entry comparison mask. TLB read and write operations use this register as a source or destination; when virtual addresses are presented for translation, the corresponding bits in the TLB specify which of the virtual address bits 24..13 participate in the comparison. Figure 4-8 shows the format of the *PageMask* register.

Table 4-2 gives MASK values for the full range of page sizes. When MASK is not one of these values, the operation of the TLB is undefined.

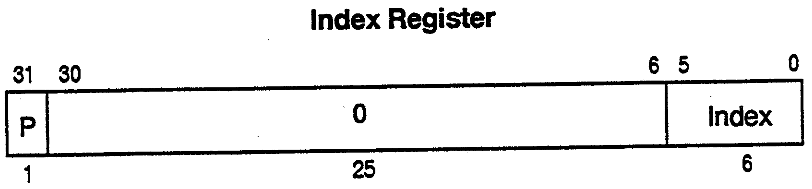
Table 4-2 MASK Values for Page Sizes

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

Index Register (0)

The *Index* register is a 32-bit, read/write register containing six bits that index an entry in the TLB. The high-order bit of the register shows the success or failure of a TLB Probe (TLBP) instruction (described at the end of this chapter).

The *Index* register also specifies the TLB entry that is affected by the TLB Read (TLBR) and TLB Write Index (TLBWI) instructions. Figure 4-9 shows the format of the *Index* register.



P Probe failure. Set to 1 when the last TLBProbe (TLBP) instruction was unsuccessful.

Index Index to the TLB entry that will be affected by the TLBRead and TLBWrite instructions.

0 Reserved. Must be written as zeroes, returns zeroes when read.

Figure 4-9 The Index Register

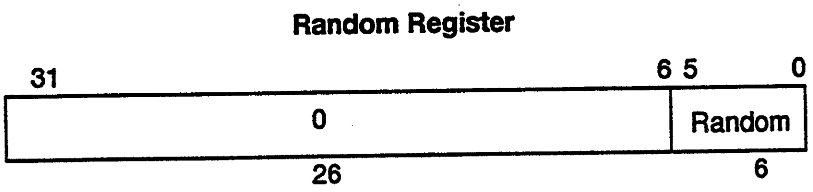
Random Register (1)

The *Random* register is a read-only register of which six bits are used to index an entry in the TLB. This register decrements for each instruction executed. The values range between:

- a lower bound set by the number of TLB entries reserved for exclusive use by the operating system (the contents of the *Wired* register), and
- an upper bound set by the total number of TLB entries. (47 maximum.)

The *Random* register specifies the entry in the TLB affected by the TLB Write Random instruction, TLBWR. 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. The format of the *Random* register is shown in Figure 4-10.



Random TLB Random Index

0 Reserved. Must be written as zeroes, returns zeroes when read.

Figure 4-10 The Random Register

Wired Register (6)

The *Wired* register is a read/write register that specifies the boundary between the wired (fixed, nonreplaceable entries that cannot be overwritten by a TLBWR operation) and random entries of the TLB (see Figure 4-11).

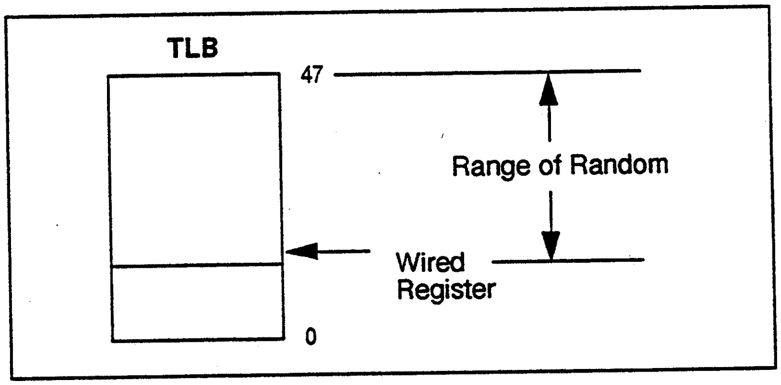
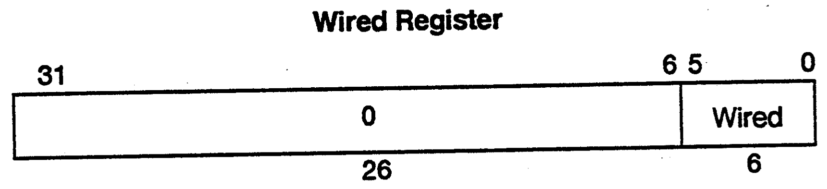


Figure 4-11 Wired Register Location

The *Wired* register is set to zero upon system reset. Writing this register also sets the *Random* register to the value of its upper bound (see *Random* Register, above). Figure 4-12 shows the format of the *Wired* register.



Wired TLB Wired boundary

0 Reserved. Must be written as zeroes, returns zeroes when read.

Figure 4-12 The Wired Register

Virtual Address Translation

During virtual-to-physical address translation, the CPU compares the ASID and, depending upon the page size, the highest 7-to-19 bits in 32-bit mode (VPN) and the highest 15-to-27 bits in 64-bit mode (VPN) of the virtual address to the contents of the TLB. Figure 4-13 illustrates the TLB address translation process.

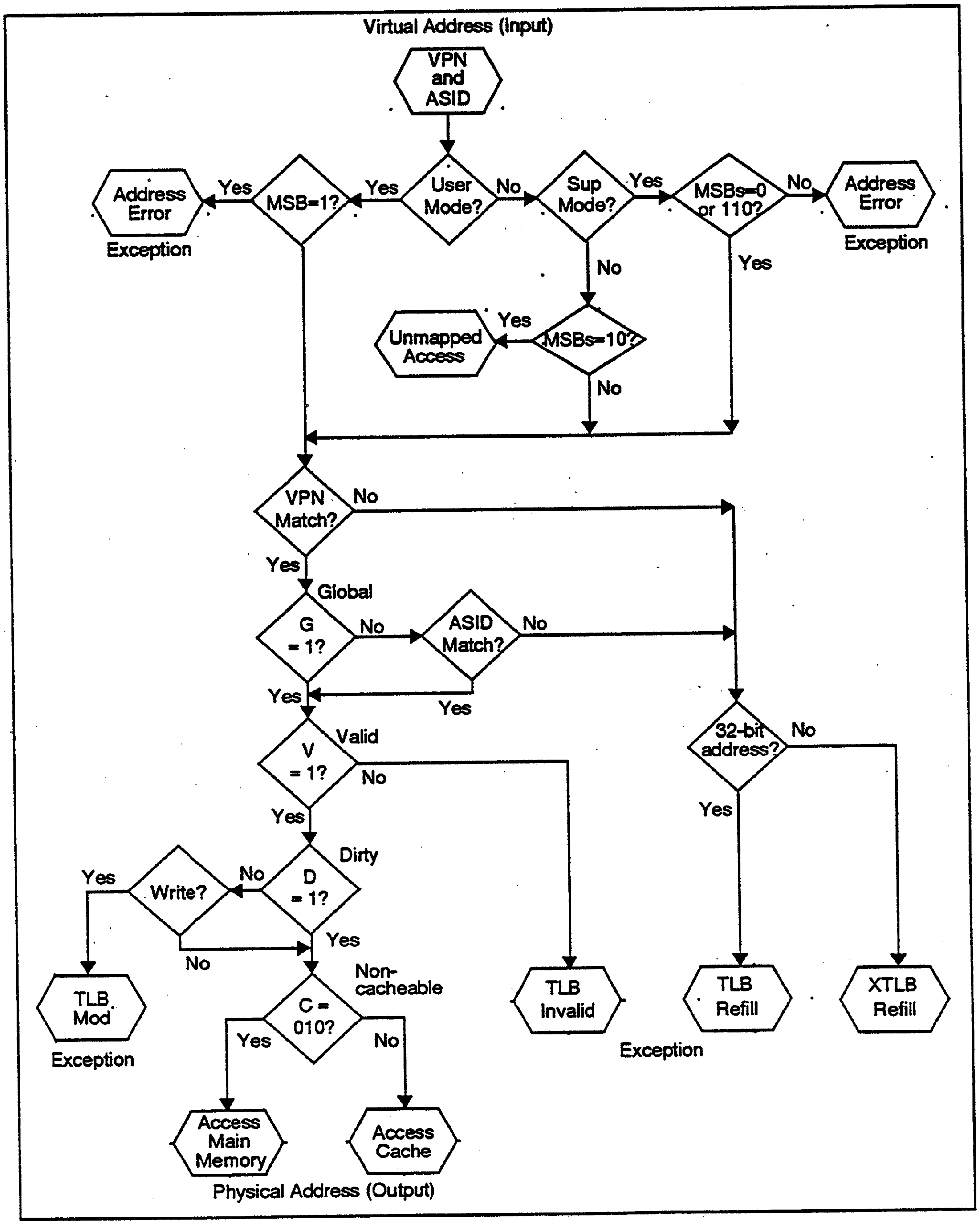


Figure 4-13 R4000 TLB Address Translation

A virtual address matches a TLB entry when the VPN field of the virtual address equals the VPN field of the entry, and either the *G* bit of the TLB entry is set or the ASID field of the virtual address (the current ASID is held in the *EntryHi* register) matches the ASID field of the 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.

If a TLB entry matches, the physical address and access control bits (*C*, *D*, and *V*) are retrieved from the matching TLB entry. Otherwise, a TLB miss exception occurs. 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 binary 010, the physical address that is retrieved is used to access main memory, bypassing the cache.

TLB Instructions

The instructions that the CPU provides for working with the TLB are listed in Table 4-3, and are described briefly below.

Table 4-3 TLB Instructions

Op Code	Description
TLBP	Translation Lookaside Buffer Probe
TLBR	Translation Lookaside Buffer Read
TLBWI	Translation Lookaside Buffer Write Index
TLBWR	Translation Lookaside Buffer Write Random

Translation Lookaside Buffer Probe (TLBP). The *Index* register is loaded with the address of the TLB entry whose contents match the contents of the *EntryHi* register. If no TLB entry matches, the high-order bit of the *Index* register is set. An instruction occurring immediately after a TLBP instruction and causing a memory data reference produces undefined results. Results are also undefined if a TLB reference produces more than one hit in the TLB.

Translation Lookaside Buffer Read (TLBR). This instruction loads the *EntryHi* and *EntryLo0, EntryLo1* registers with the contents of the TLB entry specified by the contents of the *Index* register.

Translation Lookaside Buffer Write Index (TLBWI). This instruction loads the specified TLB entry with the contents of the *EntryHi* and *EntryLo0, EntryLo1* registers. The contents of the *Index* register specify the TLB entry.

Translation Lookaside Buffer Write Random (TLBWR). This instruction loads a pseudo-randomly-specified TLB entry with the contents of the *EntryHi* and *EntryLo0, EntryLo1* registers. The contents of the *Random* register specify the TLB entry.

Exception Processing

5

This chapter describes the exception processing capabilities and hardware of the R4000. It presents an overview of the CPU exception handling process and describes the format and use of each CPU exception handling register. This chapter also describes how the R4000 handles each kind of exception.

For a description of FPU Exceptions, please refer to Chapter 6, *Floating-Point Exceptions.*

Exception Handling Operation

The R4000 processes exceptions from a number of sources, including TLB misses, arithmetic overflows, I/O interrupts, and system calls. When the CPU detects an exception, the normal sequence of instruction execution is suspended; the processor exits the current mode and enters Kernel mode. The processor then disables interrupts and forces execution of a software handler located at a fixed address. The handler saves the context of the processor, including the contents of the program counter, the current operating mode (User or Supervisor), and the status or the interrupts (enabled or disabled). This context must be restored when the exception has been handled.

When an exception occurs, the CPU loads the Exception Program Counter (*EPC*) with a restart location where execution may resume after the exception has been serviced. The restart location in the *EPC* 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.

The Exception Handling Registers

This section describes the CP0 registers that are used in exception processing. Software examines the CP0 registers during exception processing to determine the cause of an exception and the state of the CPU at the time of an exception. Each of these registers is described in detail in the sections that follow.

Table 5-1 CP0 Registers

Register Name	CP0 No.
Context	4
BadVAddr (Bad Virtual Address)	8
Count	9
Compare register	11
Status	12
Cause	13
EPC (Exception Program Counter)	14
PRId (Processor Revision Identifier)	15
Config	16
LLAdr (Load Linked Address)	17
WatchLo (Memory Reference Trap Address Low)	18
WatchHi (Memory Reference Trap Address High)	19
XContext	20
ECC	26
CacheErr (Cache Error and Status)	27
TagLo (Cache Tag)	28
TagHi (Cache Tag)	29
ErrorEPC (Error Exception Program Counter)	30

Two other CP0 registers that are part of the virtual memory management system and contain important information about exception handling are the Index Register (CP0 register 0) and the Random Register (CP0 register 1). These two registers are described in Chapter 4.

Context Register (CP0 Register 4)

The Context Register is a read/write register containing a pointer to an entry in the Page Table Entry (PTE) array. This array is an operating system data structure which stores virtual to physical address translations. When there is a TLB miss, operating system software handles the miss by loading the TLB with the missing translation from the PTE array. The Context register is intended for use by the TLB refill handler that loads entries for references to a 32-bit address space.

The *Context* register duplicates some of the information provided in the *BadVAddr* register, but the information is in a form that is more useful for a software TLB exception handler.

The *Context* register can be used by the operating system to hold a pointer into the PTE array. 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*. This register is included solely for use of the operating system.

For all addressing exceptions except bus errors, this register holds the Virtual Page Number/2 (VPN2) from the most recent virtual address for which the translation was invalid. Figure 5-1 shows the format of the *Context* register.

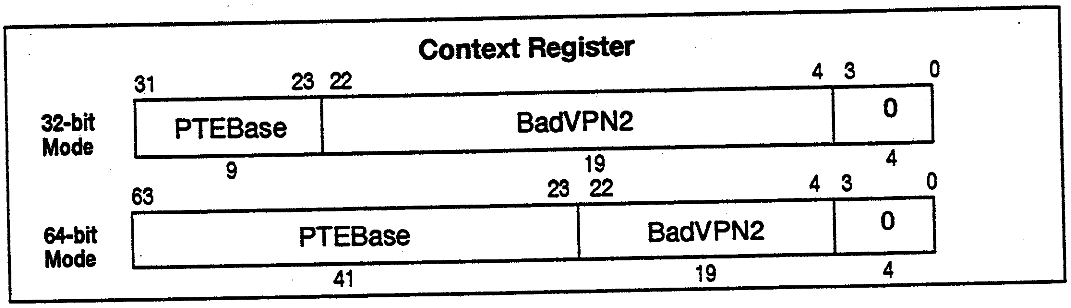


Figure 5-1 Context Register Format

Bit-field descriptions of the *Context* register are:

- The BadVPN2 field is written by hardware on a miss. It contains the VPN of the most recently translated virtual address that did not have a valid translation.
- The PTEBase is a read/write field for use by the operating system. It is normally written with a value that allow the operating system to use the *Context* register as a pointer into the current PTE array in memory.

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. This format can be used directly as an address in a table of pairs of 8-byte PTEs, for a 4-Kbyte page size. For other page and PTE sizes, shifting and masking this value produces an appropriate address.

Bad Virtual Address Register (BadVAddr) (8)

The Bad Virtual Address register (*BadVAddr*) is a read-only register that displays the most recently translated virtual address that failed to have a valid translation.

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

Figure 5-2 shows the format of the *Bad Virtual Address* register.

Note: The *Bad Virtual Address* register does not save any information for bus errors because they are not addressing errors.

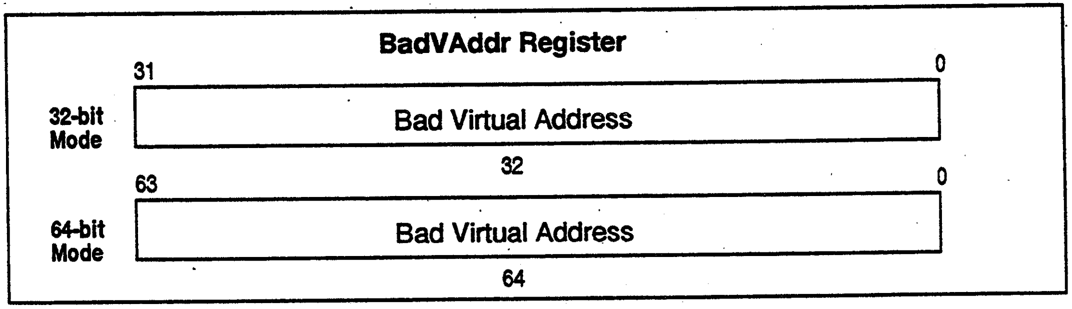


Figure 5-2 BadVAddr Register Format

Count Register (9)

The *Count* register acts as a timer, incrementing at a constant rate whether or not an instruction is executed, retired, or any forward progress is made. This register increments at half the maximum instruction issue rate.

This register can be read or written; it can be written for diagnostic purposes or system initialization to synchronize two processors operating in lock-step.

Figure 5-3 shows the format of the *Count* register.

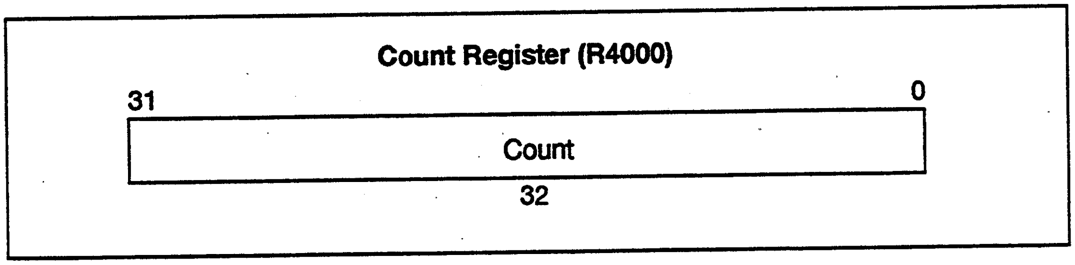


Figure 5-3 The Count Register

Compare Register (11)

The *Compare* register implements a timer service (see also the *Count* register) which maintains a stable value and does not change on its own. When the value of the *Count* register equals the value of the *Compare* register, interrupt bit *IP*$_7$ in the *Cause* register is set. This causes an interrupt to be taken on the next execution cycle in which 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 read/write. In normal use however, the *Compare* register is only written. Figure 5-4 shows the format of the *Compare* register.

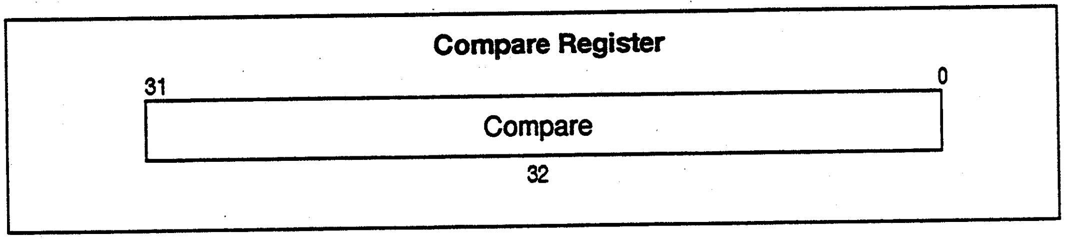


Figure 5-4 Compare Register Format

Status Register (12)

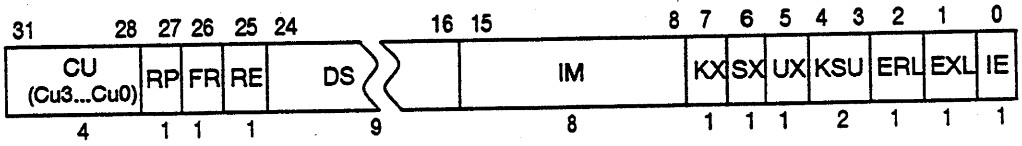
The *Status* register (SR) is a read/write register that contains the operating mode, interrupt enabling, and the diagnostic states of the processor. The following list describes *Status* register fields that are used in all R-Series processors; the format of the register is shown in Figure 5-5 and Figure 5-6:

- The Interrupt Mask (IM) field is an 8-bit field that controls the enabling of eight interrupt conditions. 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. For more information, refer to the Interrupt Pending (IP) field of the *Cause* register.
- The Coprocessor Usability (CU) field is a 4-bit field that controls the usability of four possible coprocessors. Regardless of the *CU0* bit setting, CP0 is *always* considered usable in Kernel mode.
- The Diagnostic Status (*DS*) field is a 9-bit field used for self-testing and checking the cache and virtual memory system.

The Reverse Endian *(RE)* bit, bit 25, is used to reverse the endianness of the machine in User mode. R-Series processors are configured as either Little-endian or Big-endian at system reset. This selection is used in Kernel and Supervisor modes, and in the User mode when the *RE* bit is 0; setting this bit to 1 inverts the selection in User mode.

Figure 5-5 shows the formats of the *Status* register. Additional information on the Diagnostic Status (DS) field is found in Figure 5-6.

The Status Register



CU Controls the usability of each of the four coprocessor unit numbers (1 → usable; 0 → unusable).
CP0 is always usable when in Kernel mode, regardless of the setting of the CU_0 bit.

RP Enables reduced-power operation by reducing the internal clock frequency (0 → full speed; 1→ reduced clock). The clock divisor is programmable at boot time.

FR Enables additional floating-point registers (0 → 16 registers; 1 → 32 registers).

RE Reverse Endian 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 (0 → disabled, 1→ enabled). The Interrupt Mask (IM) field is an 8-bit field that controls the enabling of eight interrupt conditions. 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.

KX Enables 64-bit addressing in kernel mode. The Extended addressing TLB refill exception is used for TLB misses on kernel addresses. (0 → 32–bit, 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. (0 → 32–bit, 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 supervisor addresses. (0 → 32–bit, 1 → 64–bit)

KSU Mode (10 → User, 01 → Supervisor, 00 → Kernel)

ERL Error Level (0 → normal, 1 → error)

EXL Exception Level (0 → normal, 1 → exception)

IE Interrupt Enable (0 → disable, 1 → enable)

0 Reserved. Must be written as zeroes, returns zeroes when read.

Figure 5-5 The Status Register

The *Status* register contains a base mode (*KSU*), base interrupt enable (*IE*), and two modifier bits (*EXL* and *ERL*). These bits allow support for Supervisor mode as well as rapid TLB refill exceptions for the kernel address space.

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

- *IE* is set to 1
- *EXL* is cleared to 0

- *ERL* is cleared to 0

If these conditions are met, interrupts are recognized according to the setting of the IM bits.

Processor Modes. The following R4000 *Status* register bit settings are required for User, Kernel, and Supervisor modes.

- The processor is in User mode when *KSU* is equal to 10, and *EXL* is cleared to 0, and *ERL* is cleared to 0.
- The processor is in Supervisor mode when *KSU* is equal to 01, and *EXL* is cleared to 0, and *ERL* is cleared to 0.
- The processor is in Kernel mode when *KSU* is equal to 00, or *EXL* is set to 1, or *ERL* is set to 1.

32- and 64-bit Modes. The following R4000 *Status* register bit settings select 32- or 64-bit operation for User, Kernel, and Supervisor 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 mode may be set independently.

- 64-bit addressing is enabled for kernel mode when *KX* is set to 1. 64-bit operations are always valid in kernel mode.
- 64-bit addressing and operations are enabled for supervisor mode when *SX* is set to 1.
- 64-bit addressing and operations areenabled for user mode when *UX* is set to 1.

Kernel Address Space Accesses. Access to the Kernel address space is allowed when the processor is in kernel mode:

- *KSU* is equal to 00.
- *EXL* is set to 1.
- *ERL* is set to 1.

Supervisor Address Space Accesses. Access to the Supervisor address space is allowed when the processor is in kernel or supervisor mode:

- *KSU* is not equal to 10 (not in User mode).
- *EXL* is set to 1.
- *ERL* is set to 1

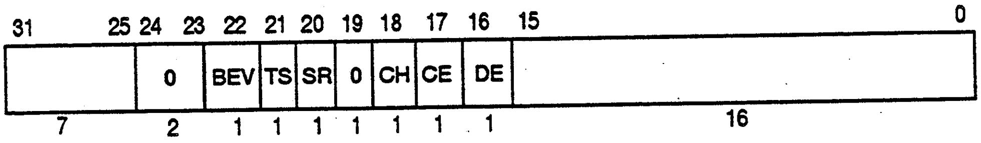
User Address Space Accesses. Access to the User address space is always allowed.

Reset. The contents of the *Status* register are undefined at reset, except for the following bits in the *Diagnostic Status Field*:

- *TS* is cleared to 0
- *ERL* and *BEV* are set to 1
- *SR* distinguishes between Reset, and Soft Reset (Nonmaskable Interrupt [NMI]).

Figure 5-6 shows the format of the diagnostic status (DS) field, along with bit descriptions. All bits in the DS field are read and write, except *TS*.

The Diagnostic Status Fields



BEV Controls the location of TLB refill and general exception vectors. (0 → normal; 1→ bootstrap)

TS TLB shutdown has occurred (read-only).

SR A soft reset 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.

CE Contents of the *ECC* register are used to set or modify the check bits of the caches when CE equals 1; see the *ECC* register description.

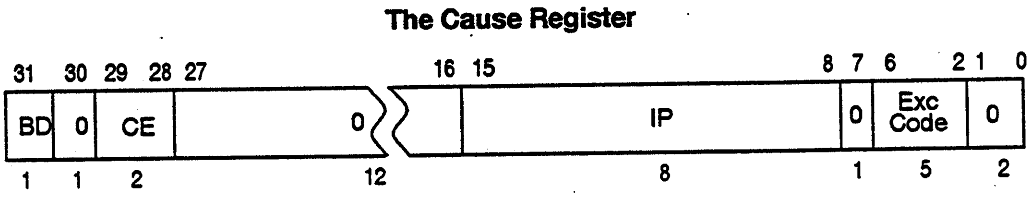
DE Specifies that cache parity or ECC errors are not to cause exceptions.

0 Reserved. Must be written as zeroes, returns zeroes when read

Figure 5-6 R4000 Status Register DS Field

Cause Register (13)

The *Cause* register is a 32-bit read/write register. The Cause Register's contents describe the cause of the most recent exception. A 5-bit exception code (*ExcCode*) indicates the cause as listed in Table 5-2. The remaining fields contain detailed information specific to certain exceptions. All bits in the register, with the exception of the *IP(1..0)* bits, are read-only. *IP(1..0)* bits are used for software interrupts. Table 5-2 shows a decoding of the 5-bit Exception Code field, and Figure 5-7 shows the format of the Cause Register.



BD Indicates whether or not the last exception was taken while executing in a branch delay slot. (1→ delay slot; 0 → normal).

CE Indicates the coprocessor unit number referenced when a Coprocessor Unusable exception is taken.

IP Indicates whether an interrupt is pending.

ExcCode This is the exception code field.

0 Reserved. Must be written as zeroes, returns zeroes when read.

Figure 5-7 Cause Register Format

Table 5-2 The ExcCode Field of Cause Register

Exception Code Value	Mnemonic	Description
0	Int	Interrupt
1	Mod	TLB modification exception
2	TLBL	TLB exception (load or instruction fetch)
3	TLBS	TLB exception (store)
4	AdEL	Address error exception (load or instruction fetch)
5	AdES	Address error exception (store)
6	IBE	Bus error exception (instruction fetch)
7	DBE	Bus error exception (data reference: 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	VCEI	Virtual Coherency Exception Instruction
15	FPE	Floating-Point exception
16–22	–	Reserved
23	WATCH	Reference to *WatchHi/WatchLo* address
24–30	–	Reserved
31	VCED	Virtual Coherency Exception Data

The R4000 processor has eight interrupts, *IP(7:0)*, which are used as follows:

- *IP(7..2)*: Reading the *Cause* register returns the inclusive OR of two internal registers for interrupts *IP(6..2)*. One of the internal registers is latched each cycle from the interrupt pins on the R4000; the other register is read and written by commands on the system interface port. On reset, *IP(7)* is configured as either a sixth external interrupt, or an internal interrupt that is set when the *Count* register is equal to the *Compare* register.
- *IP(1..0)* are software-only interrupts, and can be written to set or reset software interrupts.

Floating-point exceptions use a separate exception code contained in the Floating-Point Control and Status Registers (see Chapter 6).

Exception Program Counter (EPC) Register (14)

The Exception Program Counter (*EPC*) is a read-write register that contains the address where 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 *EPC* register is read/write.

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

The format of the *EPC* register is shown in Figure 5-8.

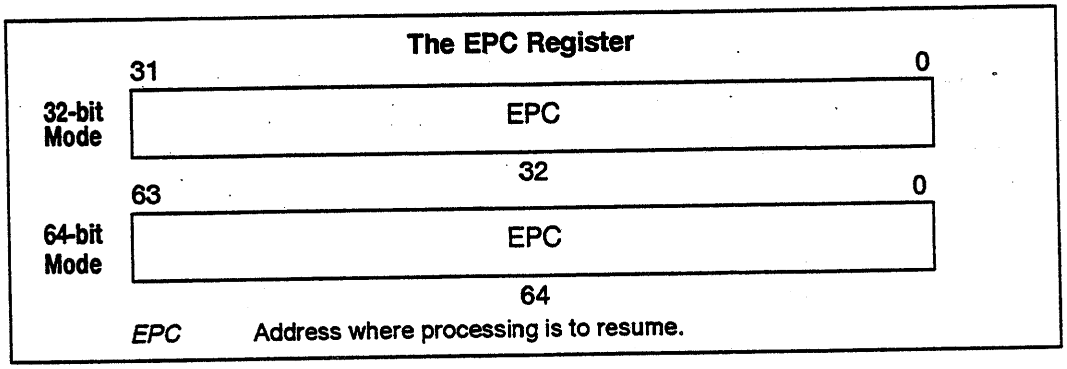
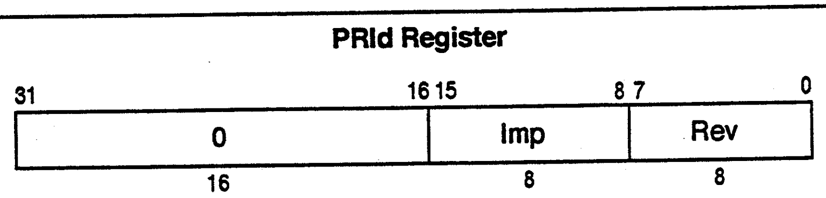


Figure 5-8 EPC Register Format

Processor Revision Identifier (PRId) Register (15)

The Processor Revision Identifier (*PRId*) is a 32-bit, read-only register that contains information identifying the implementation and revision level of the CPU and CP0. Figure 5-9 shows the format of the *PRId* register.



Imp Implementation number.

Rev Revision number.

0 Reserved. Must be written as zeroes, returns zeroes when read.

Figure 5-9 Processor Revision Identifier Register Format

The low-order byte (bits 7..0) of the *PRId* register is interpreted as a coprocessor unit revision number, and the second byte (bits 15..8) is interpreted as a coprocessor unit implementation number. The R4000 coprocessor implementation number is 0x04. The contents of the high-order halfword of the register are reserved.

The revision number is a value of 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, MIPS does not guarantee that changes to its chips 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 selected on R4000 processors. Some configuration options, as defined by *Config* bits 31..6, are set by the hardware during reset, and are included in this register as read-only status for software. Other configuration options are read / write (defined by *Config* bits 5..0) and controlled by software; on reset these fields are undefined.

The *Config* register should be initialized by software before caches are used. The caches should be completely written back to memory before changing block sizes, and reinitialized after any change is made. Figure 5-10 shows the format of the *Config* register and Table 5-3 lists the field and bit definitions for the *Config* Register.

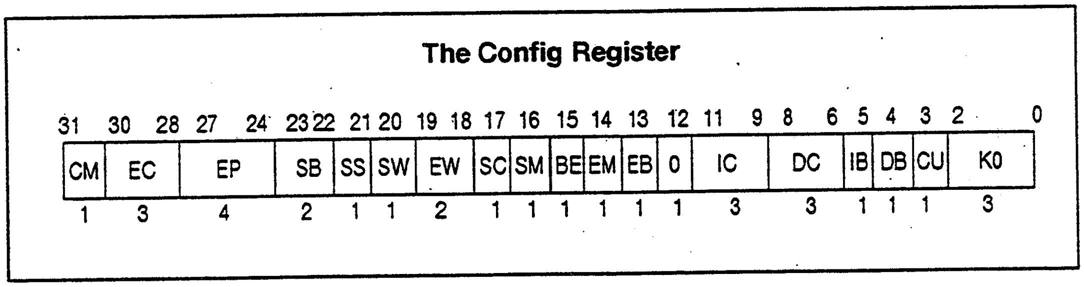


Figure 5-10 Config Register Format

Table 5-3 Config Register Field and Bit Definitions

Field/Bit Name	Description
CM	Master-Checker Mode (1 → Master-Checker Mode is enabled) This bit is automatically 0 on a Soft Reset.
EC	System clock ratio: 0 → processor clock frequency divided by 2 1 → processor clock frequency divided by 3 2 → processor clock frequency divided by 4
EP	Transmit data pattern (pattern for write-back data): 0 → D Doubleword every cycles 1 → DDx 2 Doublewords every 3 cycles 2 → DDxx 2 Doublewords every 4 cycles 3 → DxDx 2 Doublewords every 4 cycles 4 → DDxxx 2 Doublewords every 5 cycles 5 → DDxxxx 2 Doublewords every 6 cycles 6 → DxxDxx 2 Doublewords every 6 cycles 7 → DDxxxxx 2 Doublewords every 7 cycles 8 → DxxxDxxx 2 Doublewords every 8 cycles
SB	Secondary Cache block size: 0 → 4 words 1 → 8 words 2 → 16 words 3 → 32 words
SS	Split Secondary Cache Mode (0 → instruction and data mixed in secondary cache; 1 → instruction and data separated by SCAddr$_{17}$)
SW	Secondary Cache port width (0 → 128-bit data path to SCache; 1 → 64-bit)
EW	System Port width (0 → 64-bit; 1 → 32-bit)
SC	Secondary Cache present (0 → SCache present; 1 → no SCache present)
SM	Dirty Shared coherency state; 1 → then Dirty Shared state is disabled; 0 → enabled
BE	BigEndianMem (1 → then kernel and memory are Big Endian, 0 → Little Endian)
EM	ECC mode enable (0 → ECC mode enabled;1 → parity mode enabled)
EB	Block ordering (0 → then sequential,1 → sub-block)
0	Reserved. Must be written as zeroes, returns zeroes when read.
IC	Primary ICache Size (ICache size = 2^{12+IC} bytes
DC	Primary DCache Size (DCache size = 2^{12+DC} bytes)
IB	Primary ICache line size (1 → 32 bytes; 0 → 16 bytes)
DB	Primary DCache line size (1 → 32 bytes; 0 → 16 bytes)
CU	Update on Store Conditional (0 → Store Conditional uses coherency algorithm specified by TLB; 1 → SC uses cacheable coherent update on write)
K0	*kseg0* coherency algorithm (see *EntryLo0* and *EntryLo1* Registers)

Load Linked Address (LLAddr) Register (17)

The Load Linked Address (*LLAddr*) register is an R4000 read/write coprocessor register that contains the physical address read by the most recent Load Linked instruction. This register is used only for diagnostic purposes, and serves no function during normal operation. Figure 5-11 shows the format of the *LLAddr* register; *PAddr* represents bits 35..4 of the physical address.

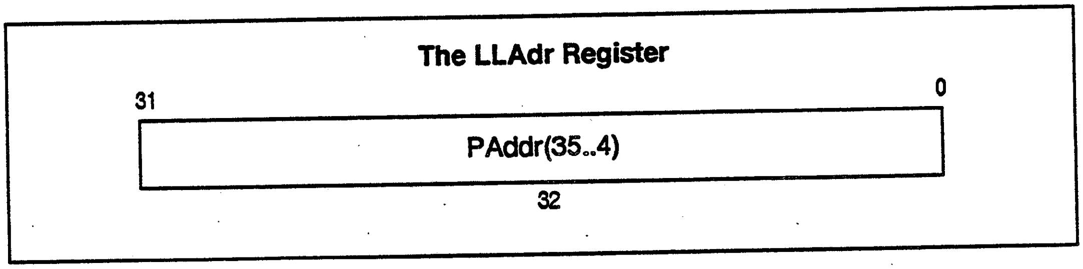


Figure 5-11 LLAdr Register Format

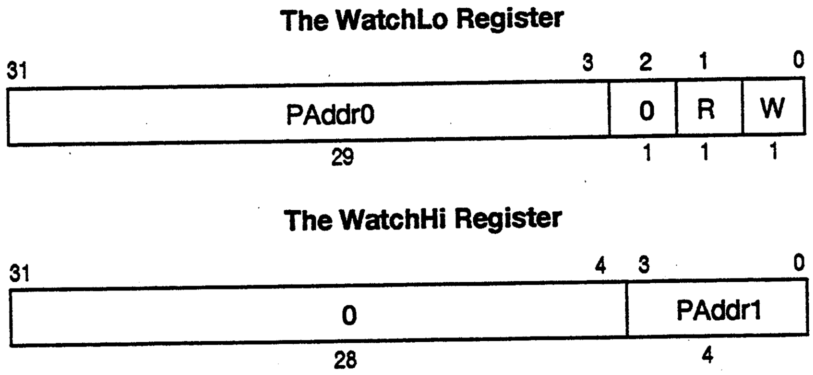
WatchLo (18) and WatchHi (19) Registers

R4000 processors provide a debugging feature to detect references to a selected physical address; load and store operations to the location specified by the *WatchLo* and *WatchHi* registers cause a Watch exception (described later in this chapter). Figure 5-12 shows the format of the *WatchLo* and *WatchHi* registers.

XContext Register (CP0 Register 20)

The XContext Register is a read/write register containing a pointer to an entry in the Page Table Entry (PTE) array. This array is an operating system data structure which stores virtual to physical address translations. When there is a TLB miss, operating system software handles the miss by loading the TLB with the missing translation from the PTE array. The *XContext* register is intended for use with the XTLB refill handler, which loads TLB entries for references to a 64-bit address space.

The *XContext* register duplicates some of the information provided in the *BadVAddr* register, but the information is in a form that is more useful for a software TLB exception handler.



PAddr1 Bits 35..32 of the physical address.
PAddr0 Bits 31..3 of the physical address.
R Trap on load references if set to1.
W Trap on store references if set to1.
0 Reserved. Must be written as zeroes, returns zeroes when read.

Figure 5-12 WatchLo and WatchHi Register Formats

The *XContext* register can be used by the operating system to hold a pointer into the PTE array. 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*. This register is included solely for use of the operating system.

For all addressing exceptions except bus errors, this register holds the Virtual Page Number/2 (VPN2) from the most recent virtual address for which the translation was invalid. Figure 5-13 shows the format of the *XContext* register.

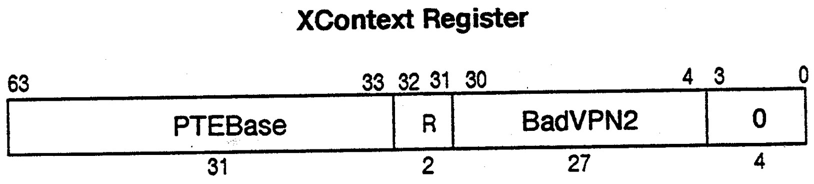


Figure 5-13 XContext Register Format

Bit-field descriptions of the *XContext* register are:

- The BadVPN2 field is written by hardware on a miss. It contains the VPN of the most recently translated virtual address that did not have a valid translation.
- R is the Region: 00→user, 01→supervisor, 11→kernel. These are bits 63..62 of the virtual address.
- The PTEBase is a read/write field 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.

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

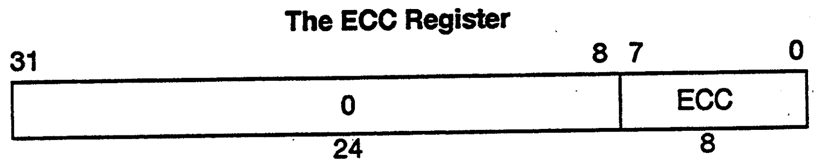
Error Correction Code (ECC) Register (26)

The Error Correction Code (*ECC*) register is an 8-bit read/write register; it reads and writes either secondary-cache data ECC bits or primary-cache data parity bits, for cache initialization, cache diagnostics, or cache error handling. (Tag ECC and parity are loaded from and stored to the *TagLo* register.)

The *ECC* register is loaded by the CACHE operation Index Load Tag. It 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 CACHE operation Fill, or
- XORed into the computed ECC for the secondary cache for the following primary data cache CACHE operations: Index Write Back Invalidate, Hit Write Back, and Hit Write Back Invalidate.

Figure 5-14 shows the format of the *ECC* register.



ECC An 8-bit field specifying the ECC bits read from or written to a secondary cache, or the even byte parity bits to be read from or written to a primary cache.

0 Reserved. Must be written as zeroes, returns zeroes when read.

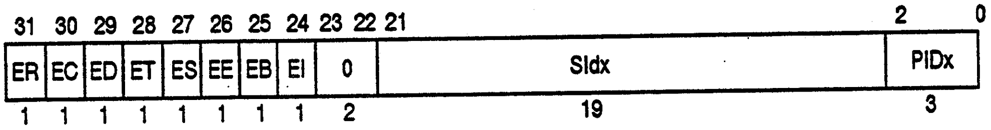
Figure 5-14 ECC Register Format

Cache Error Register (27)

The *CacheErr* register is a 32-bit read-only register that handles ECC errors in the secondary cache and parity errors in the primary cache. Parity errors cannot be corrected. All single- and double-bit ECC errors in the secondary cache tag and data are detected and single-bit errors in the tag are automatically corrected. Single-bit ECC errors in the secondary cache data are not automatically corrected.

The *CacheErr* register provides cache index and status bits which indicate the source and nature of the error; it is loaded when a Cache Error exception is taken. Figure 5-15 shows the format of the *CacheErr* register.

The CacheErr Register



ER Type of reference (0 → instruction, 1 → data).

EC Cache level of the error (0 → primary, 1 → secondary).

ED Indicates whether a data field error occurred (0 → no error, 1 → error).

ET Indicates whether a tag field error occurred (0 → no error, 1 → error).

ES Indicates that the error occurred while accessing primary or secondary cache in response to an external request (0 → internal reference, 1 → external reference).

EE Set if the error occurred on the SysAD bus.

EB Set if a data error occurred in addition to the instruction error (indicated by the remainder of the bits), which requires flushing the data cache after fixing the instruction error.

EI Set on a secondary data cache ECC error while refilling the primary cache on a store miss. The ECC handler must first do an Index Store Tag to invalidate the incorrect data from the primary data cache.

SIdx Bits pAddr(21..3) of the reference that encountered the error (which is not necessarily the same as the address of the doubleword in error, but is sufficient to locate that doubleword in the secondary cache).

PIdx Bits vAddr(14..12) of the doubleword in error (used with SIdx to construct a virtual index for the primary caches).

0 Reserved. Must be written as zeroes, returns zeroes when read.

Figure 5-15 CacheErr Register Format

Cache Tag (TagLo, and TagHi) (28) (29) Registers

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 handling. 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.

Figure 5-16 shows the format of these registers for primary cache operations.

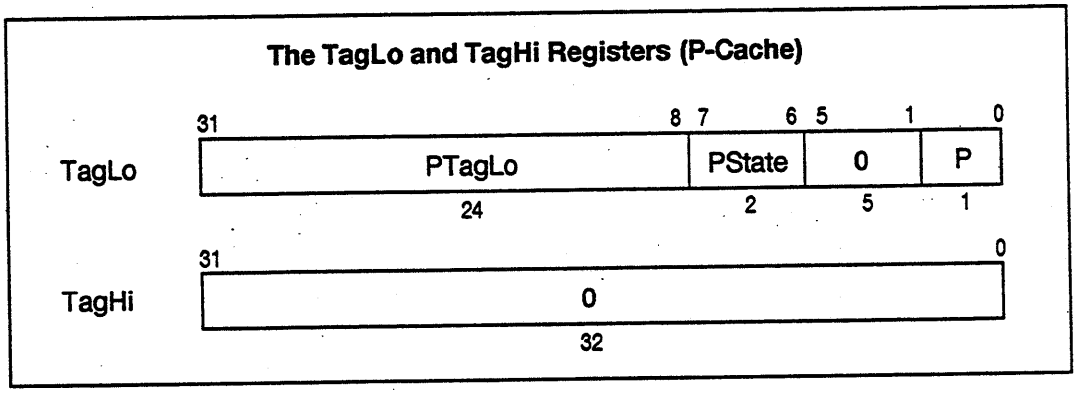


Figure 5-16 TagLo and TagHi Register (P-Cache) Formats

Figure 5-17 shows the format of these registers for secondary cache operations.

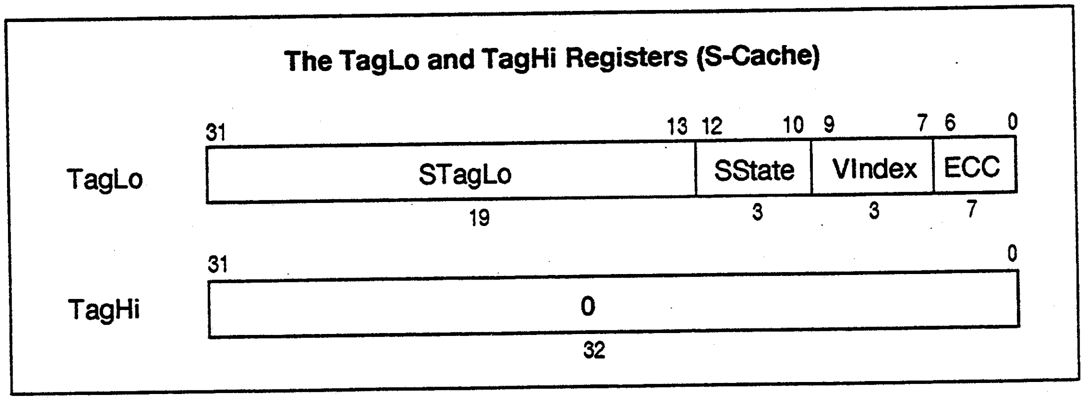


Figure 5-17 TagLo and TagHi Register (S-Cache) Formats

Bit definitions of the *TagLo* and *TagHi* registers are given in Table 5-4.

Table 5-4 The Cache Tag Field and Bit Definitions

Bit Name	Description
PTagLo	A 24-bit field specifying the physical address bits 35..12.
PState	A 2-bit field specifying the primary cache state.
P	A 1-bit field specifying the primary tag even parity bit.
STagLo	A 19-bit field specifying the physical address bits 35..17.
SState	A 3-bit field specifying the secondary cache state.
VIndex	A 3-bit field specifying the virtual index of the associated primary cache line, vAddr(14..12).
ECC	ECC for the STag, SState, and VIndex fields.
0	Reserved. Must be written as zeroes, returns zeroes when read.

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

The *ErrorEPC* register is similar to the *EPC* register, but is used on ECC and parity error exceptions. It is also used to store the PC on Reset, Soft Reset, and NMI exceptions. The read/write *ErrorEPC* register contains the virtual address at which instruction processing can resume after servicing an error. The address may be either:

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

There is no branch delay slot indication for the *ErrorEPC* register. Figure 5-18 shows the format of the *ErrorEPC* register.

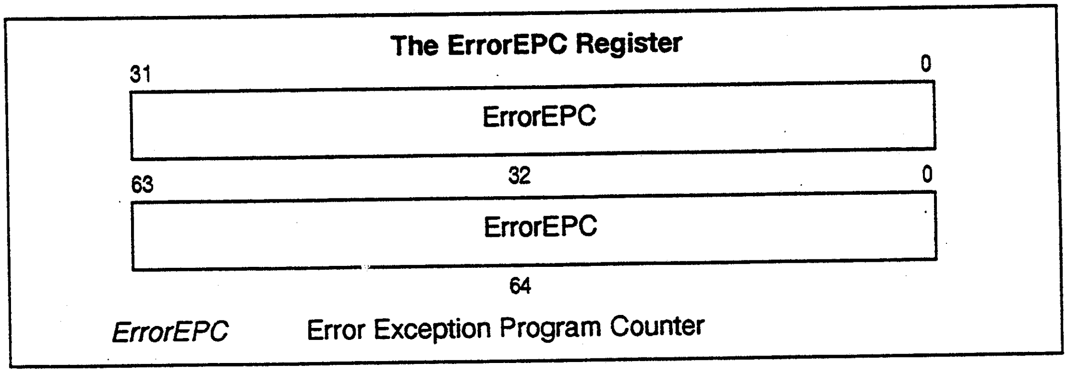


Figure 5-18 ErrorEPC Register Format

Exception Description Details

This section describes each of the R4000 exceptions — its cause, handling, and servicing.

Exception Operation

To handle an exception, the processor forces execution of a handler, at a fixed address in Kernel mode, with interrupts disabled. To resume normal operation, the Program Counter (PC), operating mode, and interrupt enable must be restored; thus it is *this* context that must be saved when an exception is taken.

When an exception occurs, the *EPC* register is loaded with the restart location at which execution can resume after servicing the exception. The *EPC* register contains the address of the instruction that caused the exception; or, if the instruction was executing in a branch delay slot, the *EPC* register contains the address of the branch instruction immediately preceding.

R4000 processors use the following mechanisms for saving and restoring the operating mode and interrupt status:

- A single interrupt enable bit (*IE*) located in the Status Register.
- A base operating mode (User, Supervisor, Kernel) located in *KSU* of the Status Register.
- An exception level (normal, exception) located in *EXL* of the Status Register.
- An error level (normal, error) located in *ERL* of the Status Register.

Interrupts are enabled by setting the *IE* bit to 1 and both levels (exception and error) to *normal.*

When the *EXL* bit in the *Status* register is zero, the *User* or *Supervisor* operating mode is specified by the *KSU* bits in the *Status* register. When the *EXL* bit is one, the processor is in *Kernal* mode and exceptions set the *EXL* bit to one. The exception handler typically resets the *EXL* bit to zero after saving the appropriate state, and then sets the *EXL* bit back to one while restoring the state and restarting. Returning from an exception (See the ERET instruction in Appendix A), resets the *EXL* bit to zero.

Figure 5-19 shows the R4000 reset exception.

```
T: undefined
   Random ← TLBENTRIES–1
   Wired ← 0
   Config <- CM || EC || EP || SB || SS || SW || EW || SC || SM || BE || EM || EB || 0 || 001
            || 001 || undefined$^6$
   ErrorEPC ← PC
   SR ← $SR_{31..23}$ || 1 || 0 || 0 || $SR_{19..3}$ || 1 || $SR_{1..0}$
   PC ← 0xBFC0 0000
```

Figure 5-19 R4000 Reset Exception

Figure 5-20 shows the R4000 Soft Reset and NMI exception.

```
T: ErrorEPC ← PC
   SR ← $SR_{31..23}$ || 1 || 0 || 1 || $SR_{19..3}$ || 1 || $SR_{1..0}$
   PC ← 0xBFC0 0000
```

Figure 5-20 R4000 Soft Reset and NMI Exception

Figure 5-21 shows the R4000 exceptions except Reset, Soft Reset, NMI, and Cache Error.

```
T: Cause ← BD || 0 || CE || $0^{12}$ || $Cause_{15..8}$ || 0 || ExcCode || $0^2$
   if $SR_1$ = 0 then
      EPC ← PC
   endif
   SR ← $SR_{31..2}$ || 1 || $SR_0$
   if $SR_{22}$ = 1 then
      PC ← 0xBFC0 0200 + vector
   else
      PC ← 0x8000 0000 + vector
   endif
```

Figure 5-21 R4000 Exceptions (Except Reset, Soft Reset, NMI, and Cache Error)

Figure 5-22 shows the Cache Error exception.

```
T: ErrorEPC ← PC
   CacheErr ← ER || EC || ED || ET || ES || EE || EB || EI || $0^2$ || SIdx || PIdx
   SR ← $SR_{31..3}$ || 1 ||$SR_{1..0}$
   if $SR_{22}$ = 1 then
      PC ← 0xBFC0 0200 + 0x100
   else
      PC ← 0xA000 0000 + 0x100
   endif
```

Figure 5-22 R4000 Cache Error Exception

Exception Vector Locations

The Reset, Soft Reset, and NMI exceptions are always vectored to location 0xBFC0 0000 in 32-bit mode or 0xFFFF FFFF BFC0 0000 in 64-bit mode. Addresses for other exceptions are a combination of a *vector offset* and a *base address,* determined by the *BEV* bit of the *Status* register. Table 5-5 shows the Vector Base addresses, and Table 5-6 shows the Vector Offset to these addresses.

Table 5-5 Exception Vector Base Addresses

BEV	R4000 Vector Base	
	32-bit mode	64-bit mode
0	0x8000 0000	0xFFFF FFFF 8000 0000
1	0x BFC0 0200	0xFFFF FFFF BFC0 0200

The vector base for the Cache Error exception is in *kseg1* (0xA000 0000 in 32-bit mode, 0xFFFF FFFF A000 0000 in 64-bit mode) instead of *kseg0* (0x8000 0000 in 32-bit mode, 0xFFFF FFFF 8000 0000 in 64-bit mode) when *BEV* is 0. This indicates that the caches are initialized and that the vector may be cached.

When *BEV* is set to a 1, vector base for the Cache Error exception is 0xBFC0 0200 in 32-bit mode and 0xFFFF FFFF BFC0 0200 in 64-bit mode which is uncached and unmapped. This vector does not rely on proper cache operation.

Table 5-6 Exception Vector Offset Addresses

Exception	R4000 Vector Offset
TLB refill, EXL = 0	0x000
XTLB refill, EXL = 0	0x080
Cache Error	0x100
Others	0x180

Priority of Exceptions

While more than one exception can occur for a single instruction, only one exception is reported, with priority given in the order shown in Table 5-7.

Table 5-7 Exception Priority Order

Reset
Soft Reset
NMI
Address error — Instruction fetch
TLB refill — Instruction fetch
TLB invalid — Instruction fetch
Cache error — Instruction fetch
Virtual Coherency — 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
Watch
Virtual Coherency — Data access
Bus error — Data access
Interrupt

Reset Exception

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

Handling. The CPU provides a special interrupt vector (0xBFC0 0000) for this exception. The Reset vector resides in unmapped and uncached CPU address space; therefore the hardware need not initialize the TLB or the cache to handle this exception. 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:

- In the *Status* register, *SR* and *TS* are cleared to 0, and *ERL* and *BEV* are set to 1. Other bits are undefined.
- The *Random* register is initialized to the value of its upper bound (see the *Random* register for more information).
- The *Wired* register is initialized to 0.

Servicing. The Reset exception is serviced by initializing all processor registers, coprocessor registers, caches, and the memory system; by performing diagnostic tests; and by bootstrapping the operating system.

The Reset exception vector is located in the uncached, unmapped memory space of the machine so that instructions can be fetched and executed while the cache and virtual memory system are still in an undefined state.

Soft Reset Exception

Cause. The Soft Reset exception occurs in response to the Reset* input signal, and execution begins at the Reset vector when Reset* is deasserted. This exception is not maskable.

Handling. The Reset exception vector (0xBFC0 0000) is used for this exception, located within unmapped and uncached address space so that the cache and TLB need not be initialized to handle this exception. The *SR* bit of the *Status* register is set to distinguish this exception from a Reset exception.

The primary purpose of the Soft Reset exception is to reinitialize the processor after a fatal error such as a Master/Checker mismatch. Unlike an NMI, all cache and bus state machines are reset by this exception; like Reset, it can be used on the processor in any state. The caches, TLB, and normal exception vectors need not be properly initialized.

The contents of all registers are preserved when this exception occurs, except for:

- The *ErrorEPC* register, which contains the restart PC.
- The *ERL* bit of the *Status* register, which is set to 1.
- The *SR* bit of the *Status* register, which is set to 1.
- The *BEV* bit of the *Status* register, which is set to 1.

Because the Soft Reset can abort cache and bus operations, cache and memory state is undefined when this exception occurs.

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

NonMaskable Interrupt (NMI) Exception

Cause. The NonMaskable Interrupt (NMI) exception occurs in response to the falling edge of the NMI pin, or an external write to the **Int*[6]** bit of the *Interrupt* register. As the name describes, this exception is not maskable; it occurs regardless of the settings of the *EXL, ERL,* and the *IE Status* register bits.

Handling. The Reset exception vector (0xBFC0 0000) is also used for this exception. This vector is located within unmapped and uncached address space so that the cache and TLB need not be initialized to handle an NMI interrupt. The *SR* bit of the *Status* register is set to differentiate this exception from a Reset exception.

Because an NMI could occur in the midst of another exception, in general it is not possible to continue program execution after servicing an NMI.

Unlike Reset and Soft Reset, but like other exceptions, NMI is taken only at instruction boundaries. The state of the caches and memory system are preserved by this exception.

The contents of all registers are preserved when this exception occurs, except for:

- The *ErrorEPC* register, which contains the restart PC.
- The *ERL* bit of the *Status* register, which is set to 1.
- The *SR* bit of the *Status* register, which is set to 1.
- The *BEV* bit of the *Status* register, which is set to 1.

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

Address Error Exception

Cause. The Address Error exception occurs when an attempt is made to:

- 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.
- Load or store a doubleword that is not aligned on a doubleword boundary.
- Reference the kernel address space from User or Supervisor mode.
- Reference the Supervisor address space from User mode.

This exception is not maskable.

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

When this exception occurs, the *BadVAddr* register retains the virtual address that was not properly aligned or which 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 points at the instruction that caused the exception, unless this instruction is in a branch delay slot. If in a branch delay slot, the *EPC* register points at 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 UNIX SIGSEGV (segmentation violation) signal. This error is usually fatal to the process incurring the exception.

TLB Exceptions

There are three different types of TLB exceptions than can occur:

- **TLB Refill** occurs when there is no TLB entry to match a 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/writable.

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.

Handling. Two special exception vectors are provided 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 are 32-bit or 64-bit spaces. All references use these vectors when the *EXL* bit is set to 0 in the *Status* register.

The *TLBL* or *TLBS* code in the *Cause* register is set. 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 points at the instruction that caused the exception, unless this instruction is in a branch delay slot, in which case the *EPC* points at 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 *EntryLo0*/*EntryLo1* register, and 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 is handled by allowing a TLB refill exception in the TLB refill handler. This second exception goes instead 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.

Handling. The common exception vector is used for this exception. The *TLBL* or *TLBS* code in the *Cause* register is set. This code 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 are undefined.

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

Servicing. The valid bit of a TLB entry is typically cleared when:

- 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 this exception, the TLB entry is located with TLBP (TLB Probe), and replaced by an entry with its 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 which is marked valid but is not dirty/writable. This exception is not maskable.

Handling. 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 are undefined.

The *EPC* register points at the instruction that caused the exception unless this instruction is in a branch delay slot, in which case the *EPC* points at 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* has occurred.

If write accesses are permitted, the page frame is marked dirty/ writable by the kernel in its own data structures. The TLBP instruction is used to place 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 secondary cache ECC error or primary cache parity error is detected. This exception is not maskable (however error detection can be disabled by the *DE* bit of the *Status* register).

Handling. The processor sets the *ERL* bit in the *Status* register, saves the exception restart address in *ErrorEPC* register, and then transfers to a special vector in uncached space: 0xA000 0100 in 32-bit mode and 0xFFFF FFFF A000 0100 in 64-bit mode if the *BEV* bit is 0, otherwise 0xBFC0 0300 in 32-bit mode and 0xFFFF FFFF BFC0 0300 in 64-bit mode.

No other registers are changed.

Servicing. All errors should be logged.

Single-bit ECC errors in the secondary cache can be corrected, using the CACHE instruction, and execution resumed through ERET.

Cache parity errors and non-single-bit ECC errors in unmodified cache blocks can be corrected by using the CACHE instruction to invalidate the cache block, then overwriting the old data through a cache miss and resuming execution with ERET. Other errors are not correctable, and are likely to be fatal to the current process.

Virtual Coherency Exception

Cause. The Virtual Coherency exception occurs when a primary cache miss hits in the secondary cache, but bits 14..12 of the virtual address were not equal to the corresponding bits of the PIdx field of the secondary cache tag, and the cache algorithm for the page (from the C field in the TLB) specifies that the page is cached. This exception is not maskable.

Handling. The common exception vector is used for this exception. The *VCEI* or *VCED* code in the *Cause* register is set for instruction and data cache misses respectively. The *BadVAddr* register holds the virtual address that caused the exception.

Servicing. The CACHE instruction can determine the old virtual index, remove the data from the primary caches at the old virtual index, and write the PIdx field of the secondary cache with the new virtual index. At this point, the program can be continued.

Software can avoid the cost of this trap by using consistent virtual primary cache indexes to access the same physical data.

Bus Error Exception

Cause. The Bus Error exception occurs when signaled 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.

Bus Error occurs only when a cache miss refill, uncached reference, or unbuffered write occurs synchronously; a Bus Error resulting from a buffered write transaction must be reported using the general interrupt mechanism.

Handling. The common interrupt vector is used for a Bus Error exception. The *IBE* or *DBE* code in 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 points at the instruction that caused the exception, unless it is in a branch delay slot, in which case the *EPC* points at 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 system control coprocessor 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 which caused the exception is located at the virtual address contained in the *EPC* register (or four plus 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 UNIX SIGBUS (bus error) signal, which is usually fatal.

Integer Overflow Exception

Cause. The 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.

Handling. The common exception vector is used for this exception. The *OV* code in the *Cause* register is set.

The *EPC* register points at the instruction that caused the exception unless the instruction is in a branch delay slot, in which case the *EPC* points at 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 UNIX SIGFPE/FPE_INTOVF_TRAP (floating-point exception/integer overflow) signal. This error is usually fatal to the current process.

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.

Handling. The common exception vector is used for this exception, and the *Tr* code in the *Cause* register is set.

The *EPC* register points at the instruction causing the exception unless the instruction is in a branch delay slot, in which case the *EPC* points at 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 UNIX SIGFPE/FPE_INTOVF_TRAP (floating-point exception/integer overflow) signal. This error is usually fatal.

System Call Exception

Cause. The System Call exception occurs on an attempt to execute the SYSCALL instruction. This exception is not maskable.

Handling. The common exception vector is used for this exception. The *Sys* code in the *Cause* register is set.

The *EPC* register points at the SYSCALL instruction unless it is in a branch delay slot, in which case the *EPC* points at 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 is not re-executed; this is accomplished by adding a value of four to the *EPC* register before returning. If a SYSCALL instruction is in a branch delay slot, a more complicated algorithm is required.

Breakpoint Exception

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

Handling. The common exception vector is used for this exception, and the *BP* code in the *Cause* register is set.

The *EPC* register points at the BREAK instruction unless it is in a branch delay slot, in which case the *EPC* points at 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 from the unused bits of the BREAK instruction (bits 25..6), and from loading the contents of the instruction at which the *EPC* register points. (A value of four must be added to the contents of the *EPC* register to locate the instruction if it resides in a branch delay slot.)

To resume execution, the *EPC* register must be altered so that the BREAK instruction is not re-executed; this is accomplished by adding the value of four to the *EPC* register 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 an attempt is made to execute an instruction whose major opcode (bits 31..26) is undefined, or a SPECIAL instruction whose minor opcode (bits 5..0) is undefined. This exception also occurs on a REGIMM instruction whose minor opcode (bits 20..16) is undefined. A Reserved Instruction exception can also occur if the processor attempts to execute 64-bit operations in 32-bit mode and operating is 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.

Handling. The common exception vector is used for this exception, and the *RI* code in the *Cause* register is set.

The *EPC* register points at the reserved instruction unless it is in a branch delay slot, in which case the *EPC* points at the preceding branch instruction.

Servicing. In current systems, no instructions in the MIPS ISA are interpreted. The process executing at the time of this exception is handed a UNIX SIGILL/ILL_RESOP_FAULT (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 is executing in User mode.

This exception is not maskable.

Handling. 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 coprocessor of the four was referenced.

The *EPC* register points at the unusable coprocessor instruction unless it is in a branch delay slot, in which case the *EPC* points at the preceding branch instruction.

Servicing. The coprocessor unit to which an attempted reference was made is identified by the Coprocessor Usage Error field. Results are one of the following:

- If the process is entitled to access, the coprocessor is marked usable and the corresponding user state is restored to the coprocessor.
- If the process is entitled to access 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 the coprocessor instruction.
- If the process is not entitled to access the coprocessor, the process executing at the time is handed a UNIX SIGILL/ ILL_PRIVIN_FAULT (illegal instruction/privileged instruction fault) signal. This error is usually fatal.

Floating-Point Exception

Cause. The Floating-Point exception is used by the R4000 floating-point coprocessor. The Floating-Point exception is not maskable.

Handling. 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.

Watch Exception

Cause. The Watch exception occurs when a load or store instruction references the physical address specified in the *WatchLo* / *WatchHi* system control coprocessor registers. The *WatchLo* register specifies whether a load or store initiated this exception.

The CACHE instruction never causes a Watch exception.

The Watch exception is postponed while the *EXL* bit is set in the *Status* register, and Watch is only maskable by setting *EXL* in the *Status* register.

Handling. The common exception vector is used for this exception, and the *Watch* code in the *Cause* register is set.

Servicing. The Watch exception is a debugging aid; typically the exception handler transfers control to a debugger, allowing the user to examine the situation. To continue, the Watch exception must be disabled to execute the faulting instruction, and then the Watch exception must be reenabled. The faulting instruction can be executed either by interpretation or by setting breakpoints.

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 *IEc* bit of the *Status* register.

Handling. 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 the current interrupt requests. It is possible that more than one of the bits will be simultaneously set (or even *no* bits may be set) if an 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.

Floating-Point Unit

6

Functional Overview

The MIPS Floating-Point Unit (FPU) operates as a coprocessor for the CPU and extends the CPU instruction set to perform arithmetic operations on values in floating-point representations. The FPU, with associated system software, fully conforms to the requirements of 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. Figure 6-1 illustrates the functional organization of the FPU.

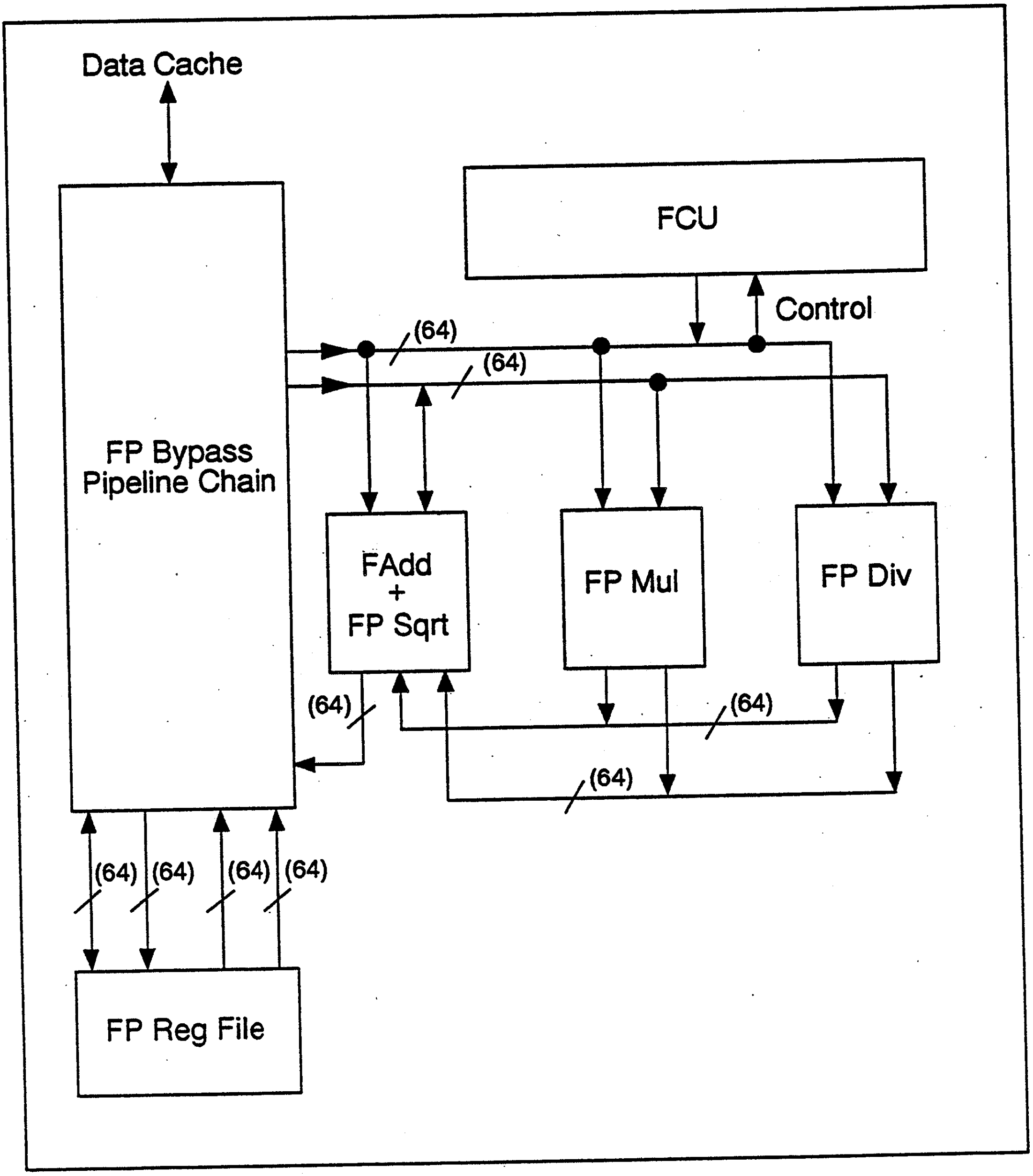


Figure 6-1 FPU Functional Block Diagram

FPU Features

- **Full 64-bit operation.** When the *FR* bit in the *Status* register equals zero, the FPU contains thirty-two 32-bit registers that hold single- or, when used in pairs, double-precision values. When the *FR* bit in the *Status* register equals one, the FPU registers are expanded to 64 bits wide. Each register can hold single- or double-precision values. The FPU also includes a 32-bit *Status/Control* register that provides access to all IEEE-Standard exception handling capabilities.
- **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. Floating-point operations are started in a single cycle and their execution is overlapped with other fixed-point or floating-point 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.

FPU Programming Model

This section describes the organization of data in registers and the set of general registers available. This section also gives a summary description of the FPU registers.

Floating-Point General Registers (FGRs)

The FPU has a set of *Floating-Point General-Purpose* registers (*FGRs*) and two control registers: the *Control/Status* and *Implementation/ Revision* registers. The general registers can be accessed in three different ways:

- As thirty-two general-purpose registers, each 32 bits wide (32 FGRs) when the *FR* bit in the *Status* register equals zero or 64-bits wide when *FR* equals one. The CPU accesses the general registers as FGRs through move, load, and store instructions.

- When the *FR* bit in the *Status* register equals zero: as sixteen floating-point registers, each 64-bit-wide FPR holds floating-point values during floating-point operations. The FPRs hold values in either single- or double-precision floating-point format. The FPU accesses the general registers as FPRs. Each FPR corresponds to adjacently numbered FGRs as shown in Figure 6-2. Only even numbers are used to address FPRs; odd FPR register numbers are invalid. During single-precision floating-point operations, only the even numbered (*least*, as shown in Figure 6-2) general registers are used, and during double-precision operations, the general registers are accessed in double pairs.
- When the *FR* bit in the *Status* register equals one: as thirty-two floating-point registers, each 64-bit-wide FPR holds floating-point values during floating-point operations. The FPRs hold values in either single- or double-precision floating-point format. The FPU accesses the general registers as FPRs. Each FPR corresponds to an FGR as shown in Figure 6-2. Both even and odd are valid to address FPRs. During single-precision floating-point operations, the low-order words of the general registers are used; during double-precision operations the general registers are accessed as 64-bit registers.

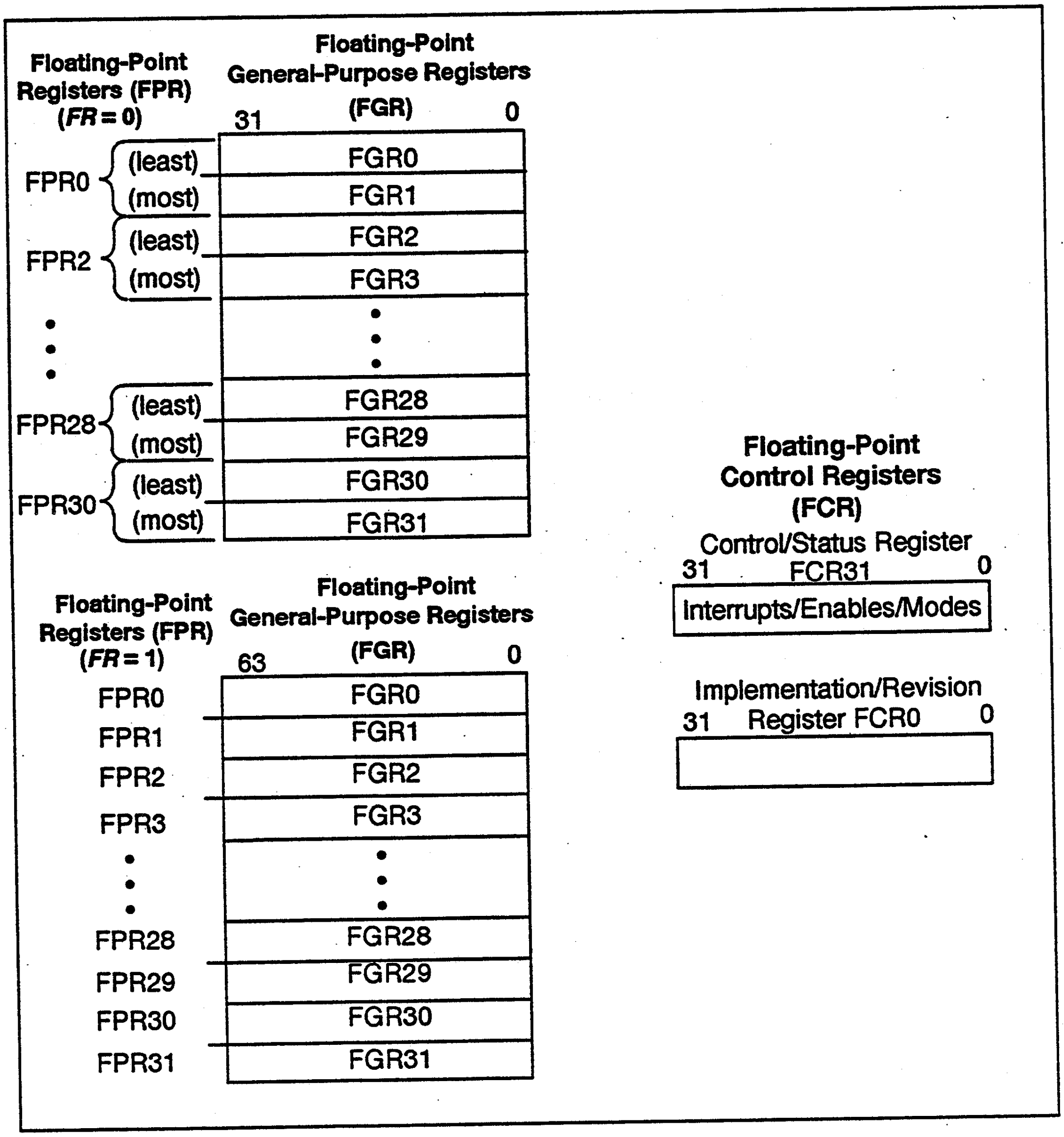


Figure 6-2 FPU Registers

Floating-Point Registers

The FPU provides 16 *Floating-Point* registers (*FPRs*) when the *FR* bit in the *Status* register equals zero or 32 *Floating-Point* registers (*FPRs*) when the *FR* bit in the *Status* register equals one. These 64-bit registers hold floating-point values during floating-point operations and are physically formed from the *General-Purpose* registers (*FGRs*).When the *FR* bit in the *Status* register equals one, the *FPR* references a single 64-bit *FGR*.

The *FPRs* hold values in either single- or double-precision floating-point format. Only even numbers are used to address *FPRs*; odd *FPR* register numbers are invalid unless the *FR* bit is set to a one. When this bit is set, all *FPR* register numbers are valid. During single-precision floating-point operations when the *FR* bit is not set, only the even-numbered (*least*, as shown in Figure 6-2) general registers are used, and during double-precision floating-point operations the general registers are accessed in double pairs. Thus, in a double-precision operation, selecting *Floating-Point Register 0* (*FPR0*) addresses adjacent *Floating-Point General-Purpose* registers *FGR0* and *FGR1*.

Floating-Point Control Registers

The R4000 FPU has 32 control registers. The following *Floating-Point Control* registers (*FCRs*) can be accessed only by move operations. The registers are described below:

- The *Control/Status* register *(FCR31)* controls and monitors exceptions, holds the result of compare operations, and establishes rounding modes.
- The *Implementation/Revision* register *(FCR0)* holds revision information about the FPU.

Table 6-1 lists the assignments of the *FCRs*.

Table 6-1 Floating-Point Control Register Assignments

FCR Number	Use
FCR0	Coprocessor implementation and revision register
FCR1–30	Reserved
FCR31	Rounding mode, cause, trap enables, and flags

Control/Status Register FCR31 (Read and Write)

The *Control/Status* register, *FCR31*, contains control and status data and can be accessed by instructions in either Kernel or User mode. It controls the arithmetic rounding mode and the enabling of User-mode traps. It also identifies exceptions that occurred in the most recently executed instruction and any exceptions that may have occurred without being trapped. Figure 6-3 shows the bit assignments of *FCR31*.

The Control/Status Register (FCR31)

31 25	24	23	22 18	17 12	11 7	6 2	1 0
0	FS	C	0	Cause EVZOUI	Enables VZOUI	Flags VZOUI	RM
7	1	1	5	6	5	5	2

Where:

FS When set, denormalized results are flushed to zero instead of causing an unimplemented operation exception.

C Condition bit. See description below.

Cause Cause bits. See Figure 6-4 and the description of *Control/Status Register Cause, Flag,* and *Enable* Bits.

Enables Enable bits. See Figure 6-4 and the description of *Control/Status Register Cause, Flag,* and *Enable* Bits.

Flags Flag bits. See Figure 6-4 and the description of *Control/Status Register Cause, Flag,* and *Enable* Bits.

RM Rounding mode bits. See Table 6-2 and the section *Control/Status Register Rounding Mode Control Bits.*

Figure 6-3 FP Control/Status Register Bit Assignments

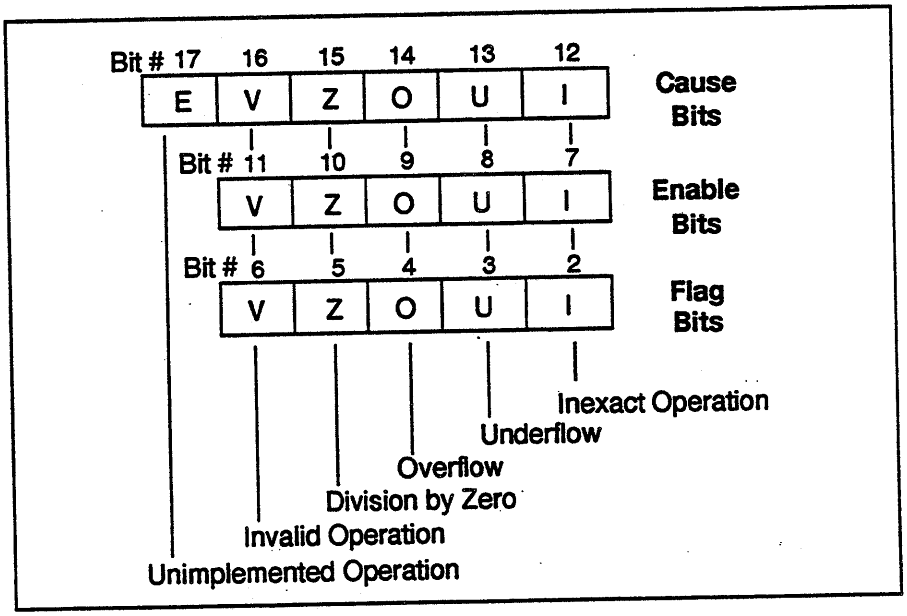


Figure 6-4 Control/Status Register Cause/Flag/Enable Bits

When the *Control/Status* register is read using 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 exception is taken and the CFC1 instruction can be 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. This register must only be written to when the FPU is not actively executing floating-point operations. This can be ensured by first 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 optionally 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. These flag bits implement IEEE 754 exception status flags, and the cause and enable bits implement exception handling.

Control/Status Register FS Bit. Bit 24 of the *Control/Status* register is the *FS* bit. When this bit is set, denormalized results are flushed to zero instead of causing an unimplemented operation exception.

Control/Status Register Condition Bit. Bit 23 of the *Control/Status* register is the *Condition* bit. When a floating-point Compare operation takes place, the result is stored at bit 23 in order that the state of the condition line can be saved or restored. The C bit is set to1 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.

Control/Status Register Cause, Flag, and Enable Bits

Figure 6-4 illustrates the Cause, Flag, and Enable bit assignments in the *Control/Status* register.

Bits 17..12 in the *Control/Status* register contain Cause bits, as shown in Figure 6-4, which reflect the results of the most recently executed instruction. The Cause 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.

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

A floating-point exception is generated any time a Cause bit and the corresponding Enable bit are both 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.

When a floating-point exception is taken, no results are stored, and the only states affected are the Cause and Flag bits. Exceptions caused by an immediately previous floating-point operation can be determined by reading the Cause field.

Before returning from a floating-point exception, or doing a CTC1, software must first clear the enabled Cause bits 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.

The appropriate Flag bits are set by the operation when a User-mode exception handler is invoked. This is not implemented in hardware; floating-point exception software is responsible for setting these bits before invoking a user handler.

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.

Figure 6-4 shows the meanings of each bit in the Cause field. If more than one exception occurs on a single instruction, each appropriate bit will be set.

The Flag bits are cumulative and indicate that an exception was raised on some operation since they were explicitly reset. Flag bits are set to 1 if an IEEE 754 exception is raised, and remain unchanged otherwise. The Flag bits are never cleared as a side effect of floating-point operations, but 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 comprise the Rounding Mode (RM) field. These bits specify the rounding mode that the FPU uses for all floating-point operations as shown in Table 6-2.

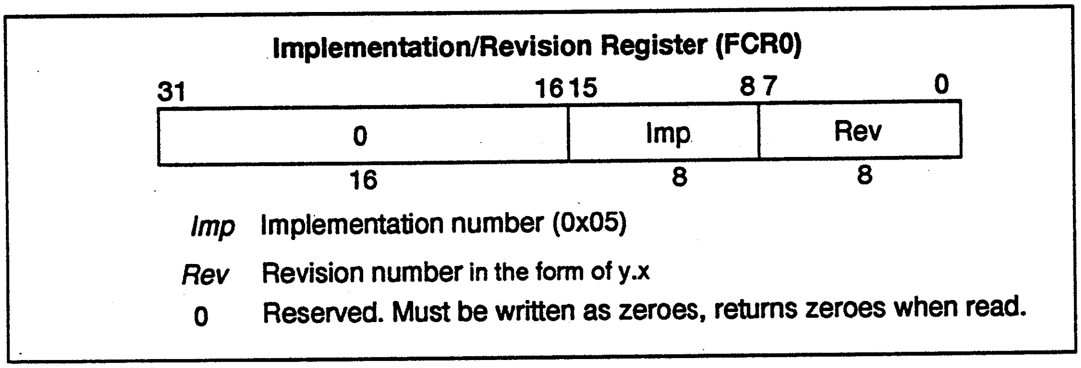
Table 6-2 Rounding Mode Bit Decoding

Rounding Mode	Mnemonic	Description
0	RN	Round result to nearest representable value; round to value with least-significant bit zero when the two nearest representable values are equally near.
1	RZ	Round toward zero: 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.

Implementation and Revision Register FCR0 (Read Only)

The *Implementation and Revision* register specifies the implementation and revision number of the FPU. This information can be used to determine the coprocessor revision and performance level, and can also used by diagnostic software.

Figure 6-5 shows the layout of the register.



Imp Implementation number (0x05)

Rev Revision number in the form of y.x

0 Reserved. Must be written as zeroes, returns zeroes when read.

Figure 6-5 Implementation/Revision Register

The revision number is a value of the form *y.x*, where *y* is a major revision number held in bits 7..4, and *x* is a minor revision number held in bits 3..0. The revision number can distinguish 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.

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 6-6.

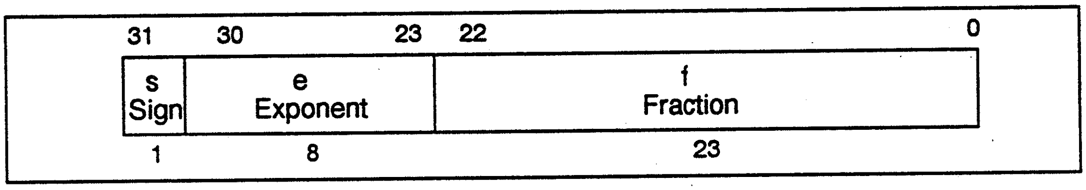


Figure 6-6 Single-Precision Floating-Point Format

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 6-7.

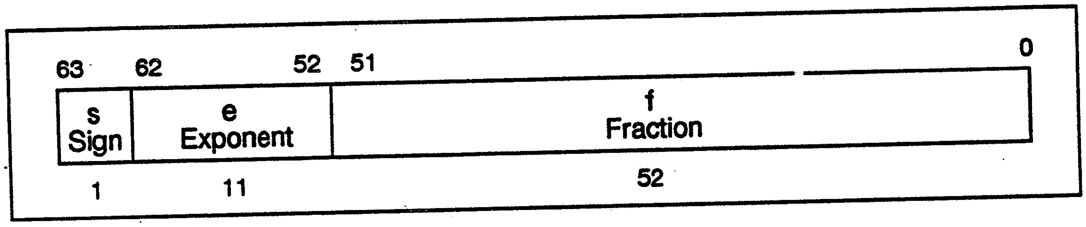


Figure 6-7 Double-Precision Floating-Point Format

Numbers in these floating-point formats are composed of three fields:

- 1-bit sign: 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 two values E_{min} and E_{max} inclusive, and also two other reserved values: E_{min} -1 (to encode +0 and denormalized numbers), and E_{max} +1 (to encode +∞ and NaNs [Not a Number]). For single- and double-precision formats, each représentable nonzero 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 6-3.

Table 6-3 Equations for Calculating Values in Single and Double-Precision Floating-Point Format

(1)	if $E = E_{max}+1$ and $f \neq 0$, then v is NaN, regardless of s.
(2)	if $E = E_{max}+1$ and $f = 0$, then $v = (-1)^s \infty$.
(3)	if $E_{min} \leq E \leq E_{max}$, then $v = (-1)^s 2^E (1.f)$.
(4)	if $E = E_{min}-1$ and $f \neq 0$, then $v = (-1)^s 2^{E_{min}} (0.f)$.
(5)	if $E = E_{min}-1$ and $f = 0$, then $v = (-1)^s 0$.

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 6-4 defines the values for the format parameters.

Table 6-4 Floating-Point Format Parameter Values

Parameter	Format	
	Single	Double
f	24	53
E_{max}	+127	+1023
E_{min}	–126	–1022
exponent *bias*	+127	+1023
exponent width in bits	8	11
integer bit	hidden	hidden
fraction width in bits	24	53
format width in bits	32	64

Minimum and maximum floating-point values are given in Table 6-5.

Table 6-5 Minimum and Maximum Floating-Point Values

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

Binary Fixed-Point Format

Binary fixed-point values are held in 2's-complementary format. Unsigned fixed-point values are not directly provided by the floating-point instruction set. Binary fixed-point format is shown in Figure 6-8.

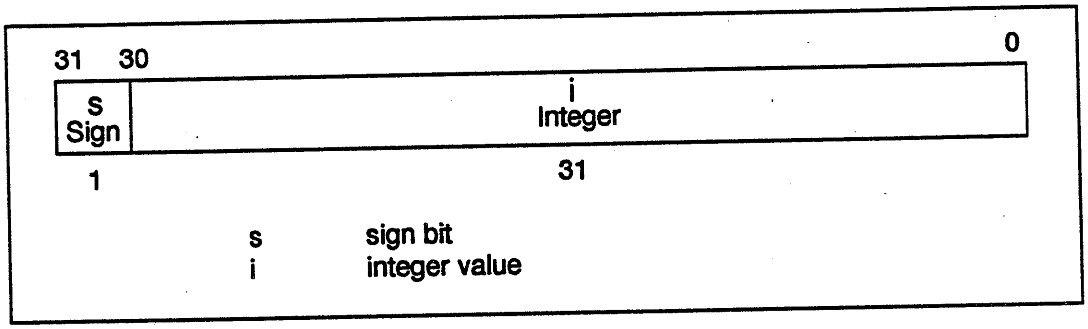


Figure 6-8 Binary Fixed-Point Format

Instruction Set Overview

All FPU instructions are 32 bits long, aligned on a word boundary, and 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 condition bit based on the results.
- **Branch on FPU Condition** instructions perform a branch to the specified target if the specified coprocessor condition is met.

Table 6-6 lists the instruction set of the FPU. A complete description of each instruction is provided in Appendix B.

Table 6-6 FPU Instruction Summary

OP	Description
	Load/Store/Move Instructions
LWC1	Load Word to FPU
SWC1	Store Word from FPU
LDC1	Load Doubleword to FPU
SDC1	Store Doubleword 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
	Conversion Instructions
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 Single Fixed Point
ROUND.w.fmt	Floating-point Round
TRUNC.w.fmt	Floating-point Truncate
CEIL.w.fmt	Floating-point Ceiling
FLOOR.w.fmt	Floating-point Floor
	Computational Instructions
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
	Compare Instructions
C.cond.fmt	Floating-point Compare
	Branch on FP Condition
BC1T	Branch on FPU True
BC1F	Branch on FPU False
BC1TL	Branch on FPU True Likely
BC1FL	Branch on FPU False Likely
.fmt	format specifier
.cond	condition specifier

Load, Store, and Move Instructions

All movement of data between the FPU and memory is accomplished by:

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

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

Data can also be moved directly between the FPU and the CPU by Move To Coprocessor 1 (MTC1), Move From Coprocessor 1 (MFC1), Doubleword Move To Coprocessor 1 (DMTC1), Doubleword Move From Coprocessor 1 (DMFC1)instructions. Like the floating-point load and store operations, these operations perform no format conversions and never cause floating-point exceptions.

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

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 two bits of the address must always be zero.
- For doubleword loads and stores, the access type is always DOUBLEWORD, and the low-order three bits of the address must always be zero.

Regardless of byte-numbering order (endianness), 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.

Table 6-7 summarizes the load, store, and move instructions.

Table 6-7 FPU Load, Store and Move Instruction Summary

Instruction	Format and Description	op	base	ft	offset
Load Word to FPA (coprocessor 1)	*LWC1 ft,offset(base)* Sign-extend 16-bit *offset* and add to contents of CPU register *base* to form address. Load contents of addressed word into FPU general register *ft.*				
Store Word from FPA (coprocessor 1)	*SWC1 ft,offset(base)* Sign-extend 16-bit *offset* and add to contents of CPU register *base* to form address. Store the contents of FPU general register *ft* at addressed location.				
Load Double-word to FPA (coprocessor 1)	*LDC1 ft,offset(base)* Sign-extend 16-bit *offset* and add to contents of CPU register *base* to form address. Load contents of addressed doubleword into FPU general registers *ft* and *ft+1* (FR=0), or FPU general register *ft* (FR=1).				
Store Double-word from FPA (coprocessor 1)	*SDC1 ft,offset(base)* Sign-extend 16-bit *offset* and add to contents of CPU register *base* to form address. Store the 64-bit contents of FPU general registers *ft* and *ft+1* (FR=0), or FPU general register *ft* (FR=1) at addressed location.				

Instruction	Format and Description	COP1	sub	rt	fs	0
Move Word to FPA (coprocessor 1)	*MTC1 rt,fs* Move contents of CPU register *rt* into FPU general register *fs.*					
Move Word from FPA (coprocessor 1)	*MFC1 rt,fs* Move contents of FPU general register *fs* into CPU register *rt.*					
Move Control Word to FPA (coprocessor 1)	*CTC1 rt,fs* Move contents of CPU register *rt* into FPU control register *fs.*					
Move Control Word from FPA (coprocessor 1)	*CFC1 rt,fs* Move contents of FPU control register *fs* into CPU register *rt.*					
Doubleword Move to FPA (coprocessor 1)	*DMTC1 rt,fs* Move contents of CPU register *rt* into FPU general register *fs.*					
DoubleWord Move from FPA (coprocessor 1)	*DMFC1 rt,fs* Move contents of FPU general register *fs* into CPU register *rt.*					

Floating-Point Conversion Instructions

Conversion instructions perform conversions between the various data formats such as single- or double-precision fixed- or floating-point formats. Table 6-8 lists the conversion instructions and their formats.

Table 6-8 FPU Conversion Instruction Summary

Instruction	**Format and Description** COP1 fmt 0 fs fd function
Floating-Point Convert to Single FP Format	*CVT.S.fmt fd,fs* Interpret the contents of FPU register *fs* in the specified format (*fmt*) and arithmetically convert to single binary floating-point format. Place the rounded result in FPU register *fd*.
Floating-Point Convert to Double FP Format	*CVT.D.fmt fd,fs* Interpret the contents of FPU register *fs* in the specified format (*fmt*) and arithmetically convert to the double binary floating-point format. Place the rounded result in FPU register *fd*.
Floating-Point Convert to Single Fixed-Point Format	*CVT.W.fmt fd,fs* Interpret the contents of FPU register *fs* in the specified format (*fmt*) and arithmetically convert to the single fixed-point format. Place the result in FPU register *fd*.
Floating-point Round	*ROUND.W.fmt fd,fs* Interpret the contents of FPU register *fs* in the specified format (*fmt*) and arithmetically convert to the single fixed-point format. Place the result in FPU register *fd*.
Floating-point Truncate	*TRUNC.W.fmt fd,fs* Interpret the contents of FPU register *fs* in the specified format (*fmt*) and arithmetically convert to the single fixed-point format. Place the result in FPU register *fd*.
Floating-point Ceiling	*CEIL.W.fmt fd,fs* Interpret the contents of FPU register *fs* in the specified format (*fmt*) and arithmetically convert to the single fixed-point format. Place the result in FPU register *fd*.
Floating-point Floor	*FLOOR.W.fmt fd,fs* Interpret the contents of FPU register *fs* in the specified format (*fmt*) and arithmetically convert to the single fixed-point format. Place the result in FPU register *fd*.

Floating-Point Computational Instructions

Computational instructions perform arithmetic operations on floating-point values in registers. There are two categories of computational instructions, as shown in Table 6-8 and listed below:

- **3-Operand Register-Type** instructions, which perform floating-point addition, subtraction, multiplication, division, and square root operations.
- **2-Operand Register-Type** instructions, which perform floating-point absolute value, move, and negate operations.

Table 6-9 FPU Computational Instruction Summary

Instruction	**Format and Description** COP1 fmt ft fs fd function
Floating-Point Add	ADD.fmt fd,fs,ft Interpret the contents of FPU registers *fs* and *ft* in the specified format(*fmt*) and add arithmetically. Place the rounded result in FPU register *fd*.
Floating-Point Subtract	SUB.fmt fd,fs,ft Interpret the contents of FPU registers *fs* and *ft* in the specified format(*fmt*) and arithmetically subtract. Place the rounded result in FPU register *fd*.
Floating-Point Multiply	MUL.fmt fd,fs,ft Interpret the contents of FPU registers *fs* and *ft* in the specified format(*fmt*) and arithmetically multiply. Place the rounded result in FPU register *fd*.
Floating-Point Divide	DIV.fmt fd,fs,ft Interpret the contents of FPU registers *fs* and *ft* in the specified format (*fmt*) and arithmetically divide *fs* by *ft*. Place the rounded result in FPU register *fd*.
Floating-Point Absolute Value	ABS.fmt fd,fs Interpret the contents of FPU register *fs* in the specified format (*fmt*) and take arithmetic absolute value. Place the result in FPU register *fd*.
Floating-Point Move	MOV.fmt fd,fs Interpret the contents of FPU register *fs* in the specified format (*fmt*) and copy into FPU register *fd*.
Floating-Point Negate	NEG.fmt fd,fs Interpret the contents of FPU register *fs* in the specified format (*fmt*) and take arithmetic negation. Place the result in FPU register *fd*.
Floating-Point Square root	SQRT.fmt fd,fs Interpret the contents of FPU register *fs* in the specified format (*fmt*) and take the positive arithmetic square root. Result is rounded then placed in the FPU register *fd*.

In the instruction formats shown in Table 6-8 and Table 6-9 the *fmt* term appended to the instruction opcode is the data format specifier:

s specifies single-precision binary floating-point, *d* specifies double-precision binary floating-point, and *w* specifies binary fixed-point.

For example, an ADD.D specifies that the operands for the addition operation are double-precision binary floating-point values.

When *fmt* is single-precision or binary fixed-point, the odd register of the destination is undefined.

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 6-8 summarizes the floating-point Compare instructions and Table 6-11 lists the conditions that can be specified for the compare operation.

Table 6-10 FPU Compare Instruction Summary

Instruction	Format and Description | COP1 | fmt | ft | fs | 0 | function |
Floating-Point Compare	*C.cond.fmt* *fs,ft* Interpret the contents of FPU registers *fs* and *ft* in the specified format (*fmt*) and compare arithmetically. The result is determined by the comparison and the specified condition (*cond*). After a 1-instruction delay, the condition is available for testing by the CPU with the Branch on Floating-Point Coprocessor Condition (*BC1T*, *BC1F*) instructions.

Table 6-11 Relational Mnemonic Definitions

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

Branch on FPU Condition Instructions

Table 6-12 summarizes the four Branch on FPU (coprocessor unit 1) Condition instructions that can be used to test the result of the FPU Compare (C.cond) instructions. In this table, *delay slot* refers to the instruction immediately following the branch instruction. Refer to Chapter 2 for a discussion of the branch delay slot.

Table 6-12 Branch on FPU Condition Instructions

Instruction	**Format and Description** COP1 | BC | br | offset
Branch on FPU True	*BC1T offset* Compute a branch target address by adding the address of the instruction in the delay slot and the 16-bit *offset* (shifted left two bits and sign extended). Branch to the target address (with a delay of one instruction) if the FPU condition line is true.
Branch on FPU False	*BC1F offset* Compute a branch target address by adding the address of the instruction in the delay slot and the 16-bit *offset* (shifted left two bits and sign extended). Branch to the target address (with a delay of one instruction) if the FPU condition line is false.
Branch on FPU True Likely	*BC1TL offset* Compute a branch target address by adding the address of the instruction in the delay slot and the 16-bit *offset* (shifted left two bits and sign extended). Branch to the target address (with a delay of one instruction) if the FPU condition line is true. If conditional branch is not taken, theinstruction in the branch delay slot is nullified.
Branch on FPU False Likely	*BC1FL offset* Compute a branch target address by adding the address of the instruction in the delay slot and the 16-bit *offset* (shifted left two bits and sign extended). Branch to the target address (with a delay of one instruction) if the FPU condition line is false. If conditional branch is not taken, the instruction in the branch delay slot is nullified.

FPU Instruction Pipeline

The FPU provides an instruction pipeline that parallels the CPU instruction pipeline. It shares the same 8-stage pipeline architecture with the CPU, as described in Chapter 2, *Instruction Set Summary.*

Instruction Execution

Figure 6-9 illustrates how the eight instructions overlap in the FPU pipeline.

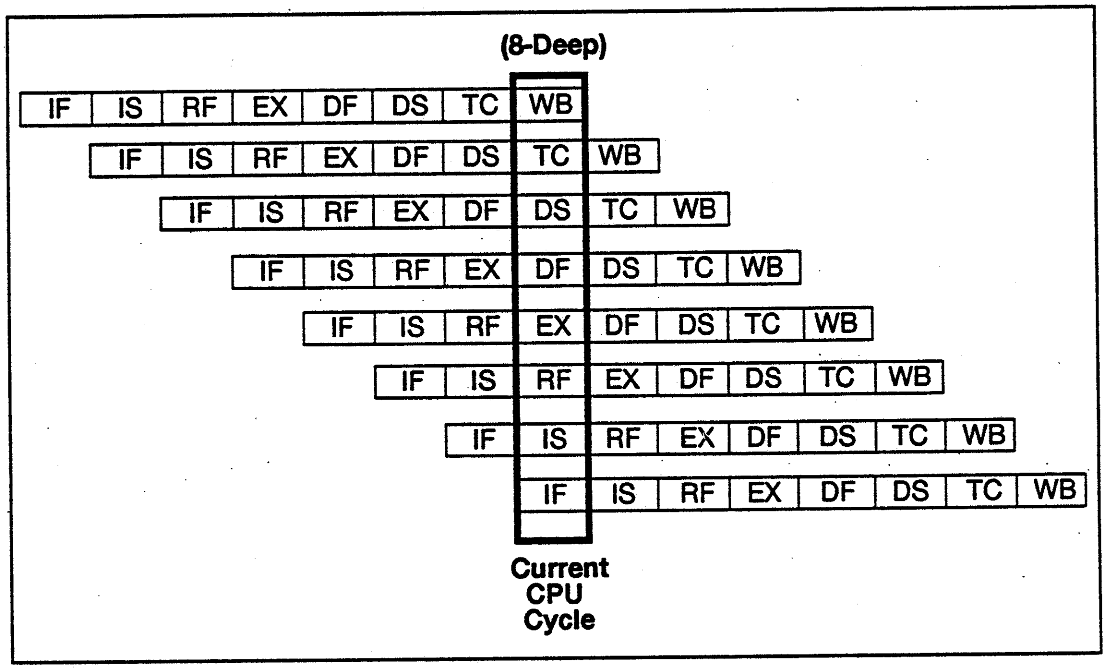
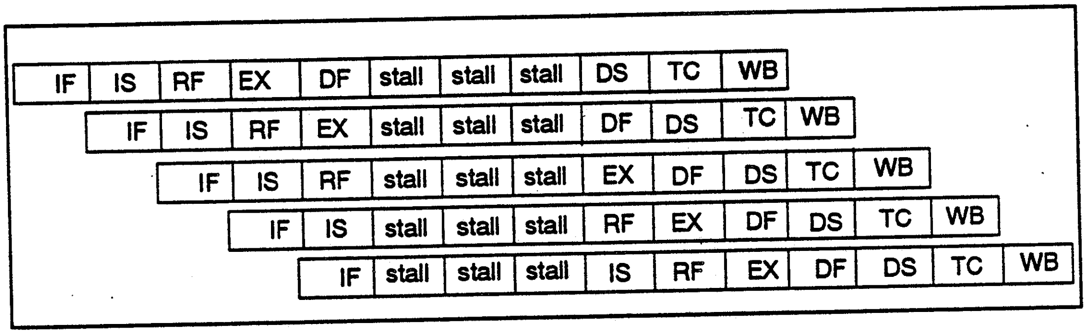


Figure 6-9 FPU Instruction Pipeline

Figure 6-9 assumes that one instruction is completed every pcycle. Most FPU instructions, however, require more than one cycle in the EX stage. Therefore, the FPU must stall the pipeline if an instruction execution cannot proceed because of register or resource conflicts. Figure 6-10 illustrates the effect of a three-cycle stall on the FPU pipeline.



IF	IS	RF	EX	DF	stall	stall	stall	DS	TC	WB				
	IF	IS	RF	EX	stall	stall	stall	DF	DS	TC	WB			
		IF	IS	RF	stall	stall	stall	EX	DF	DS	TC	WB		
			IF	IS	stall	stall	stall	RF	EX	DF	DS	TC	WB	
				IF	stall	stall	stall	IS	RF	EX	DF	DS	TC	WB

Figure 6-10 FPU Pipeline Stall

To mitigate the performance impact that would result from stalling the instruction pipeline, the FPU allows instructions to overlap so that instruction execution can proceed as long as there are no resource conflicts, data dependencies, or exception conditions. The following sections describe the timing and overlapping of FPU instructions.

Instruction Execution Times

Unlike the CPU, which executes almost all instructions in a single cycle, the time required to execute FPU instructions operates within a larger range.

Table 6-13 gives the minimum latency, in processor pipeline cycles, of each floating-point operation for the currently implemented configurations. These latency calculations assume the result of the operation is immediately used in a succeeding operation.

Table 6-13 Floating-Point Operation Latencies

	Pipeline cycles		
Operation	**Single**	**Double**	**Word**
ADD.fmt	4	4	(b)
SUB.fmt	4	4	(b)
MUL.fmt	7	8	(b)
DIV.fmt	23	36	(b)
SQRT.fmt	54	112	(b)
ABS.fmt	2	2	(b)
MOV.fmt	1	1	(b)
NEG.fmt	2	2	(b)
ROUND.W.fmt	4	4	(b)
TRUNC.W.fmt	4	4	(b)
CEIL.W.fmt	4	4	(b)
FLOOR.W.fmt	4	4	(b)
CVT.S.fmt	(b)	4	6
CVT.D.fmt	2	(b)	5
CVT.W.fmt	4	4	(b)
C.fmt.cond	3(a)	3(a)	(b)
BC1T	(c)	1	(c)
BC1F	(c)	1	(c)
BC1TL	(c)	1	(c)
BC1FL	(c)	1	(c)
LWC1	(c)	3	(c)
SWC1	(c)	1	(c)
LDC1	(c)	3	(c)
SDC1	(c)	1	(c)
MTC1	(c)	3(a)	(c)
MFC1	(c)	3	(c)
CTC1	(c)	3(a)	(c)
CFC1	(c)	2	(c)

(a) Software *must* schedule operations so that an FPU register that is the target of a floating-point load or move is not read until at least two instructions later. Software *must* also schedule a floating-point branch instruction two or more instructions after a floating-point compare instruction.

(b) These operations are illegal.

(c) These operations are undefined.

Scheduling FPU Instructions

The floating-point architecture permits the pipelining of operations and the overlapping of floating-point operations with floating-point load, store, and move instructions and with other processor operations.

The FPU coprocessor implements three separate operation (op) units: multiply, divide, and an adder for remaining operations.

Multiplies and divides can overlap with adder operations; however, they use the adder on their final cycles, which imposes some limitations.

The multiply unit can begin a new double-precision multiply every four cycles, and a new single-precision multiply every three cycles. The adder generally begins a new operation one cycle before the previous cycle completes; therefore, a floating-point add or subtract can start every three cycles.

The FPU coprocessor pipeline is fully bypassed and interlocked.

FPU Pipeline Overlapping

The FPU has three operational (op) units: adder, divider, and multiplier. Each op unit is controlled by an FPU resource scheduler, which issues instructions under certain constraints, as described in the following section.

Table 6-14 lists the pipe stages used in the op units (although not all stages are used by each unit).

Table 6-14 FPU Operational Unit Pipe Stages

Stage	Description
A	FPU Adder Mantissa Add stage
E	FPU Adder Exception Test stage
EX	CPU EX stage
M	FPU Multiplier 1st stage
N	FPU Multiplier 2nd stage
R	FPU Adder Result Round stage
S	FPU Adder Operand Shift stage
U	FPU Unpack stage

Instruction Scheduling Constraints

The FPU resource scheduler is kept from issuing instructions to FPU operation units (adder, multiplier, and divider) by the limitations in their micro-architectures listed below. If any of the following constraints are violated, the operation unit assumes the outstanding instruction in its pipe is discarded, and then continues operation on the most recently issued instruction.

FPU Divider Constraints

Handles only one non-overlapped divide instruction in its pipe at any one time.

FPU Multiplier Constraints

Allows up to two pipelined MUL.[S,D] instructions to be processed as long as the following constraints are met:

- Two idle cycles are required after MUL.S (shown in Figure 6-11)
- Three idle cycles are required after MUL.D (shown in Figure 6-12).

These figures are not meant to imply that back-to-back multiplies are allowed. Rather, as shown in Figure 6-11, I2 and I3 are illegal and I5, I6, I7, and I8 are successive stages of I4, referenced to I1. Figure 6-12 is similar, in that I6, I7, and I8 are successive stages of I5.

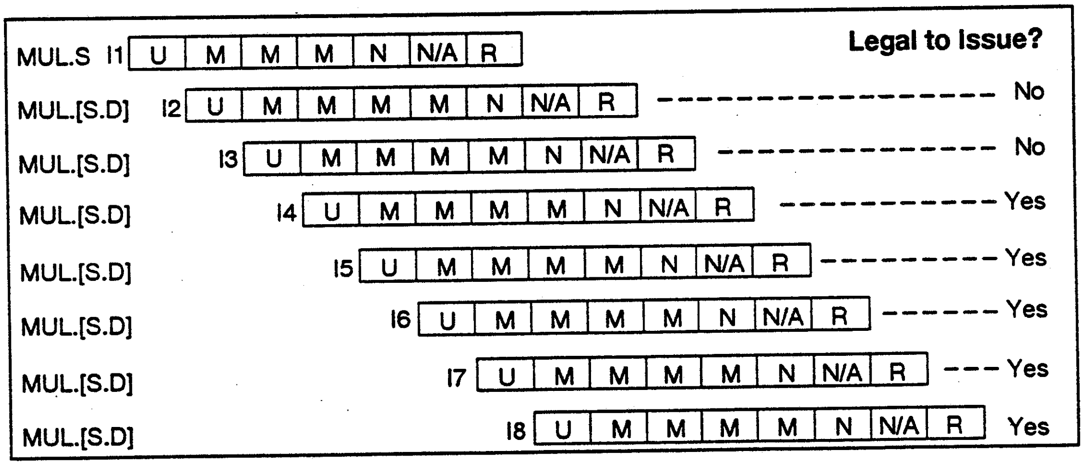

Legal to Issue?
MUL.S I1 U M M M N N/A R
MUL.[S.D] I2 U M M M M N N/A R No
MUL.[S.D] I3 U M M M M N N/A R No
MUL.[S.D] I4 U M M M M N N/A R Yes
MUL.[S.D] I5 U M M M M N N/A R Yes
MUL.[S.D] I6 U M M M M N N/A R Yes
MUL.[S.D] I7 U M M M M N N/A R Yes
MUL.[S.D] I8 U M M M M N N/A R Yes

Figure 6-11 MUL.S Instruction Scheduling in R4000 FPU Multiplier

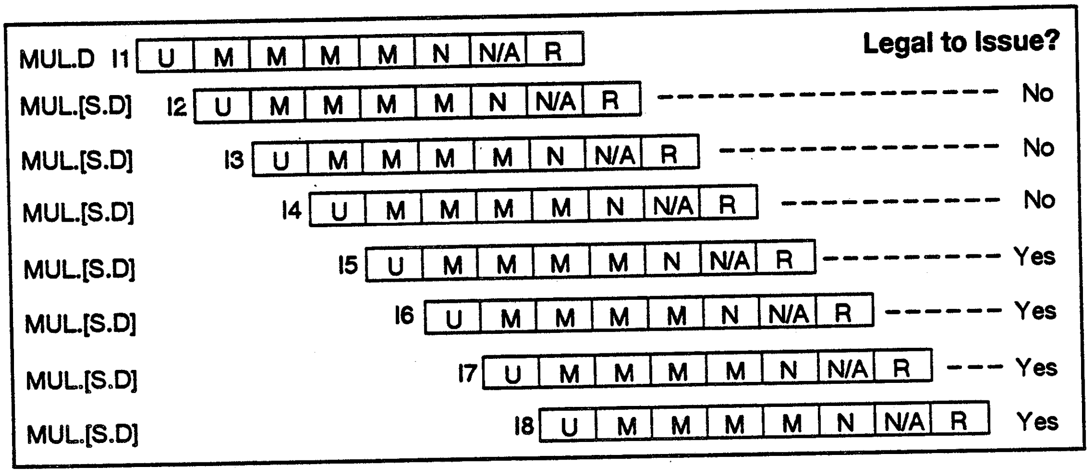

Legal to Issue?
MUL.D I1 U M M M M N N/A R
MUL.[S.D] I2 U M M M M N N/A R No
MUL.[S.D] I3 U M M M M N N/A R No
MUL.[S.D] I4 U M M M M N N/A R No
MUL.[S.D] I5 U M M M M N N/A R Yes
MUL.[S.D] I6 U M M M M N N/A R Yes
MUL.[S.D] I7 U M M M M N N/A R Yes
MUL.[S.D] I8 U M M M M N N/A R Yes

Figure 6-12 MUL.D Instruction Scheduling in R4000 FPU Multiplier

FPU Adder Constraints

The following constraints must be met in the FPU adder op unit:

- The adder op unit allows one clock cycle overlap between each newly-issued instruction and the instruction being completed (as shown in Figure 6-13).

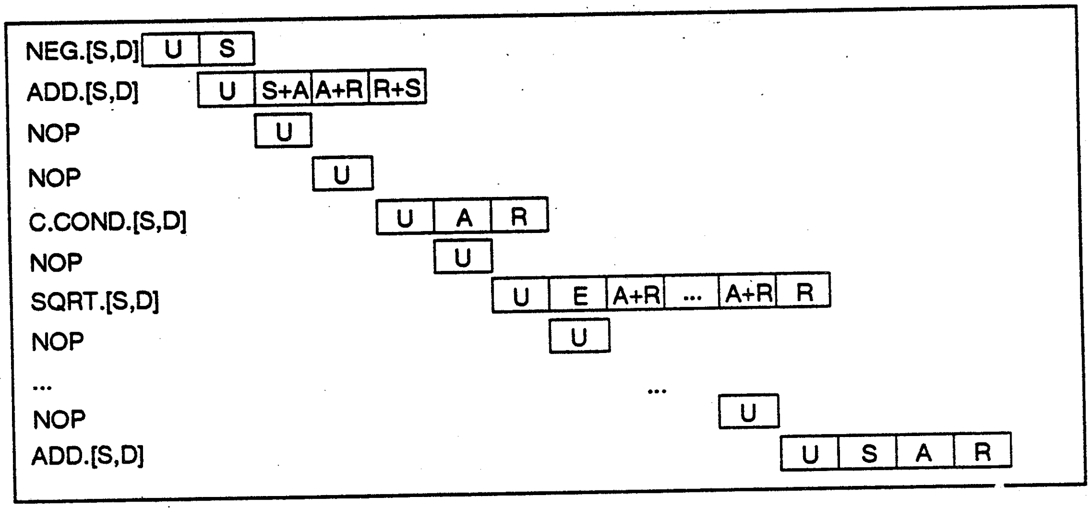


Figure 6-13 Instruction Cycle Overlap in R4000 FPU Adder

- The adder allows the cleanup stages (A, R) of a multiply instruction to be pipelined with the execution of ADD.[S,D], SUB.[S,D] or C.COND.[S,D], as long as no two instructions attempt to simultaneously use the same A and R pipe stages. For instance, Figure 6-14 shows a resource conflict between the mantissa add (A, stage 7) of instructions 1, 5, and 6. This figure also shows the resource conflict between result round (R) stage 8 of instructions 1, 5, and 6. The multiply cleanup cycles (A, R) can neither overlap nor pipeline with any other instruction currently in the adder's pipe. These constraints are shown in Figure 6-15, Figure 6-16, and Figure 6-17.

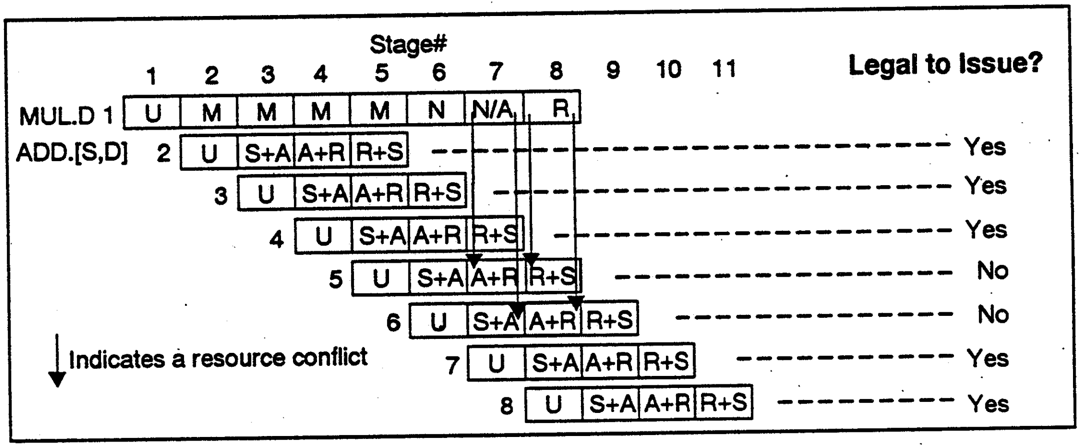


Figure 6-14 MUL.D and ADD.[S,D] Cycle Conflict in R4000 FPU Adder

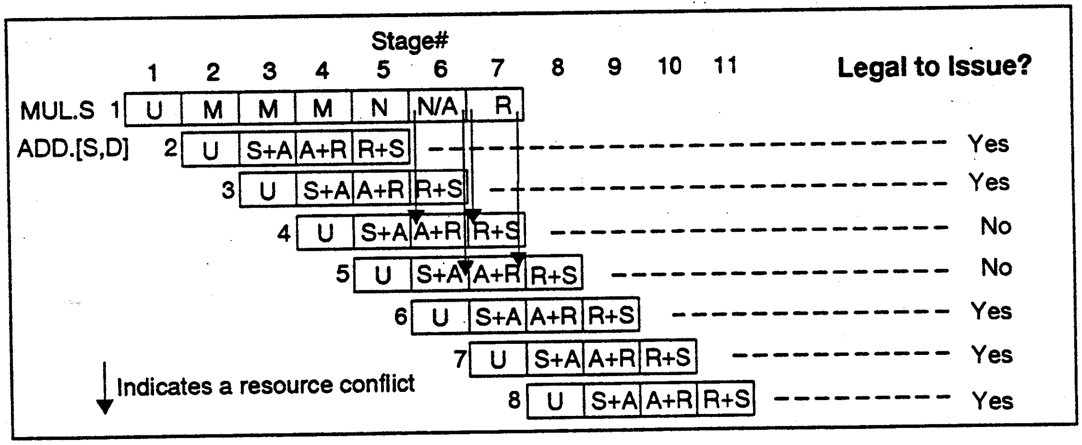


Figure 6-15 MUL.S and ADD.[S,D] Cycle Conflict in R4000 FPU Adder

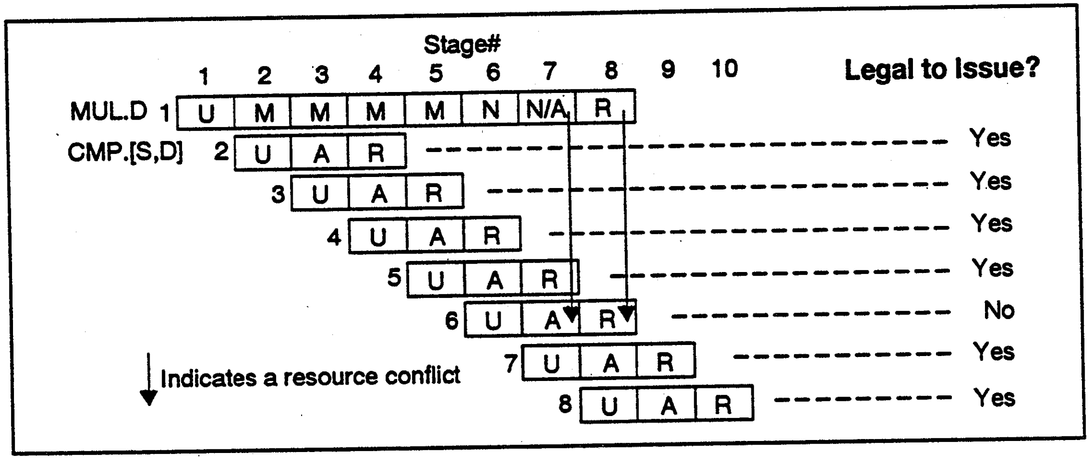


Figure 6-16 MUL.D and CMP.[S,D] Cleanup Cycle Conflict in R4000 FPU Adder

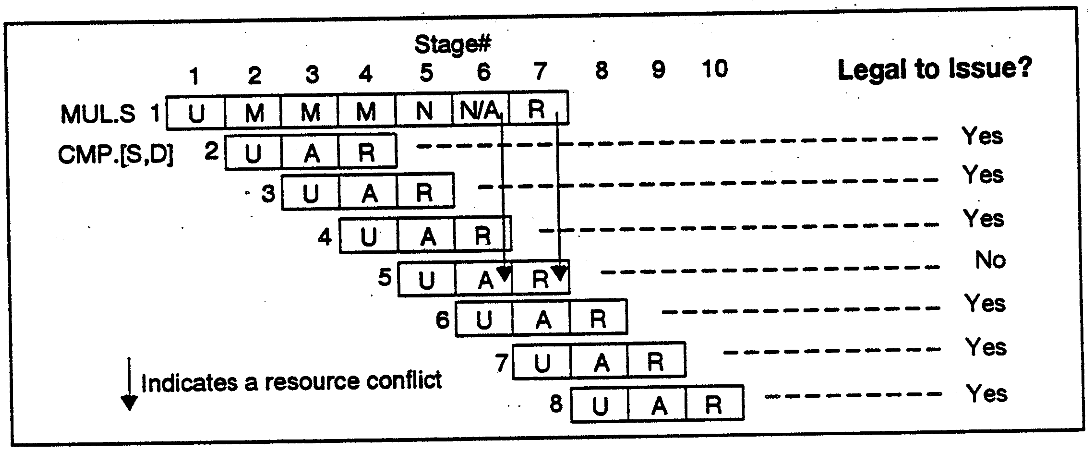


Figure 6-17 MUL.S and CMP.[S,D] Cleanup Cycle Conflict in R4000 FPU Adder

- The adder does not allow the preparation (U stage) and cleanup cycles (N, A, R) of a divide instruction to be pipelined with any other instruction; however, the adder does allow the last cycle of preparation or cleanup to be overlapped one clock by the following instructions's U stage (the CPU EX cycle) as shown in Figure 6-18.

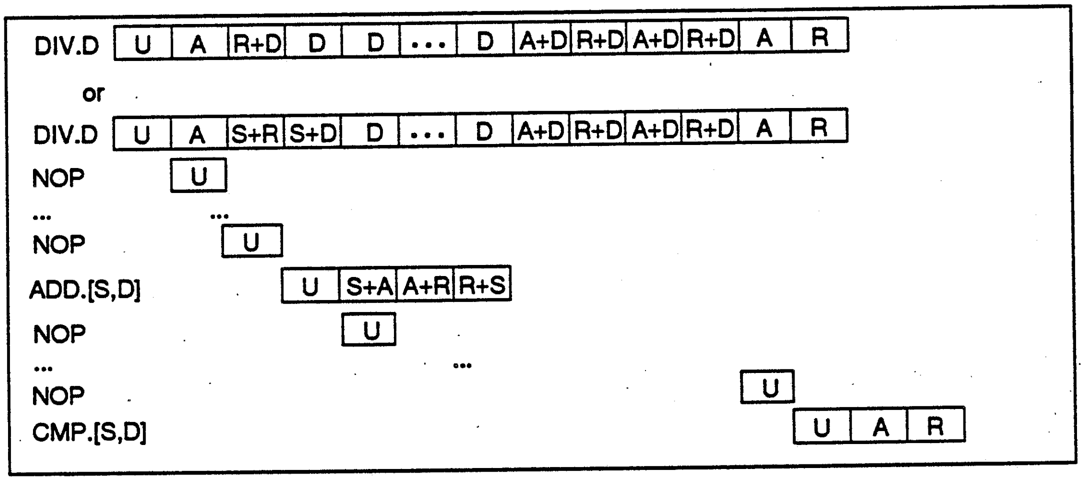


Figure 6-18 Adder Prep and Cleanup Cycle Overlap

Instruction Latency, Repeat Rate and Pipeline Stage Sequences

Table 6-15 shows the latency and repeat rate between instructions, together with the sequence of pipeline stages for each instruction. For instance, the latency of the ADD.[S,D] is 4, which means it takes four processor cycles to complete. The Repeat Rate column indicates how soon an instruction can be repeated; for instance, an ADD.[S,D] can be repeated after the conclusion of the third pipeline stage.

Table 6-15 Latency, Repeat Rate, and Pipe Stages of R4000 FPU Instructions

Instruction Type	Latency	Repeat Rate	Pipeline Stage Sequence
MOV.[S,D]	1	1	EX
ADD.[S,D]	4	3	U→ S+A→ A+R→ R+S
SUB.[S,D]	4	3	U→ S+A→ A+R→ R+S
C.COND.[S,D]	3	2	U→ A→ R
NEG.[S,D]	2	1	U→ S
ABS.[S,D]	2	1	U→ S
CVT.S.W	6	5	U→ A→ R→ S→ A→ R
CVT.D.W	5	4	U→ S→ A→ R→ S
CVT.S.L	7	6	U→ A→ R→ S→ S→ A→ R
CVT.D.L	4	3	U→ A→ R→ S
CVT.D.S	2	1	U→ S
CVT.S.D	4	3	U→ S→ A→ R
CVT.W.[S,D] *or* ROUND.W.[S,D] *or* TRUNC.W.[S,D] *or* CEIL.W.[S,D] *or* FLOOR.W.[S,D]	4	3	U→ S→ A→ R
MUL.S	7	3	U→ E/M→ M→ M→ N→ N/A→ R
MUL.D	8	4	U→ E/M→ M→ M→ M→ N→ N/A→ R
DIV.S	23	22	U→ S+A→ S+R→ S→ D...D→ D/A→ D/R→ D/A→ D/R→A→R
DIV.D	36	35	U→ A→ R→ D...D→ D/A→ D/R→ D/A →D/R→ A→ R
SQRT.S	2–54	2–53	U→ E→ A+R→......→ A+R→ A→ R
SQRT.D	2–112	2–111	U→ E→ A+R→......→ A+R→ A→ R

Resource Scheduling Rules

The FPU Resource Scheduler issues instructions while adhering to the rules described below. These scheduling rules optimize op unit executions; if the following rules are not followed the hardware interlocks to guarantee correct operation.

DIV.[S,D] can start only when all of the following conditions are met in the RF stage:

- The divider is idle.
- The adder is either idle, or in its second-to-last execution cycle.
- The multiplier is either idle, or in its first execution cycle.

Idle means an operation unit, adder, multiplier, or divider, is either not processing any instruction, or is currently at its last execution cycle completing an instruction.

MUL.[S,D] can start only when all of the following conditions are met in the RF stage:

- The multiplier is either idle, or it is:
  - within the third execution cycle (EX+2) if the most recent instruction in multiplier's pipe is MUL.S, or
  - within the fourth execution cycle (EX+3) if the most recent instruction in multiplier's pipe is MUL.D.
- The adder is either idle, or it must not be:
  - processing the first execution cycle (EX) of a conversion from long integer to short floating-point, CVT.S.L,
  - within the first three preparation cycles (EX..EX+2) of a DIV.S,
  - in the second preparation cycle (EX+1) of a DIV.D, or
  - processing a square root instruction.
- The divider is either idle, or it must not be:
  - executing within the last fifteen cycles of a DIV.[S,D],
  - in the second execution cycle (EX+1) of a DIV.D, or
  - in the first three execution cycles (EX..EX+2) of a DIV.S.

SQRT.[S,D] can start when both of the following conditions are met in the RF stage:

- The adder is either idle, or it is in its second-to-last execution cycle.
- The multiplier and divider must be idle.

CVT.fmt instructions can only start when all of the following conditions are met in the RF stage:

- The adder is either idle, or it is in its second-to-last execution cycle.
- The multiplier is either idle, or in one of the states described below:
  - If the instruction is an CVT.S.L, CVT.S.W or CVT.D.W, the multiplier must be idle.
  - If the instruction is an CVT.D.L, CVT.S.D, CVT.W.[S,D], CEIL.W.[S,D], FLOOR.W.[S,D], ROUND.W.[S,D], or TRUNC.W.[S,D], the multiplier must not be executing beyond the first cycle (EX) of a MUL.S or the second cycle (EX+1) of a MUL.D. If two multiply instructions have already been initiated in the multiplier, none of these convert instructions are allowed to start.
  - If the instruction is an CVT.D.S, the multiplier must not be executing the second-to-last execution cycle of either the first or second MUL.[S,D] in the multiplier pipe.
- The divider is idle, or not executing the first three (EX..EX+2) nor the last fifteen cycles of a DIV.[S,D].

ADD.[S,D] or **SUB.[S,D]** can start only when all of the following conditions are met in the RF stage:

- The adder is either idle, or it is in its second-to-last execution cycle.
- The multiplier is either idle, or, among two possible MUL.[S,D] instructions, it is not executing within either the fourth or fifth execution cycle from the last.
- The divider is either idle, or it is not executing within the first three (EX..EX+2) nor the last fifteen cycles of a DIV.[S,D].

NEG.[S,D] or ABS.[S,D] can start only when all of the following conditions are met in the RF stage:

- The adder is either idle, or it is in its second-to-last execution cycle.
- The multiplier is either idle, or it is not executing the second-to-last execution cycle.
- The divider is either idle, or it is not executing the first three (EX..EX+2) nor the last fifteen cycles of a DIV.[S,D].

C.COND.[S,D] can start only when all of the following conditions are met in the RF stage:

- The adder is either idle, or it is in its second-to-last execution cycle.
- The multiplier is either idle, or it is not executing the fourth cycle from the last.
- The divider is either idle, or it is not executing the first three (EX..EX+2) nor the last fifteen cycles of a DIV.[S,D].

Floating-Point Exceptions

7

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

The FP *Control/Status* register described in Chapter 6, *Floating-Point Unit*, contains an enable bit for each exception type; these 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 is executed. 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)
- Overflow (O)
- Underflow (U)
- Divide by Zero (Z)
- Invalid Operation (V)

with Cause bits, Enables, and Flag bits (status flags).

The FPU adds a sixth exception type, unimplemented operation (E), to be used when the FPU cannot implement the standard MIPS floating-point architecture, including cases where the FPU cannot determine the correct exception behavior. This exception indicates that a software implementation must be used. 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 7-1 illustrates the *Control/Status* register bits used to support exceptions.

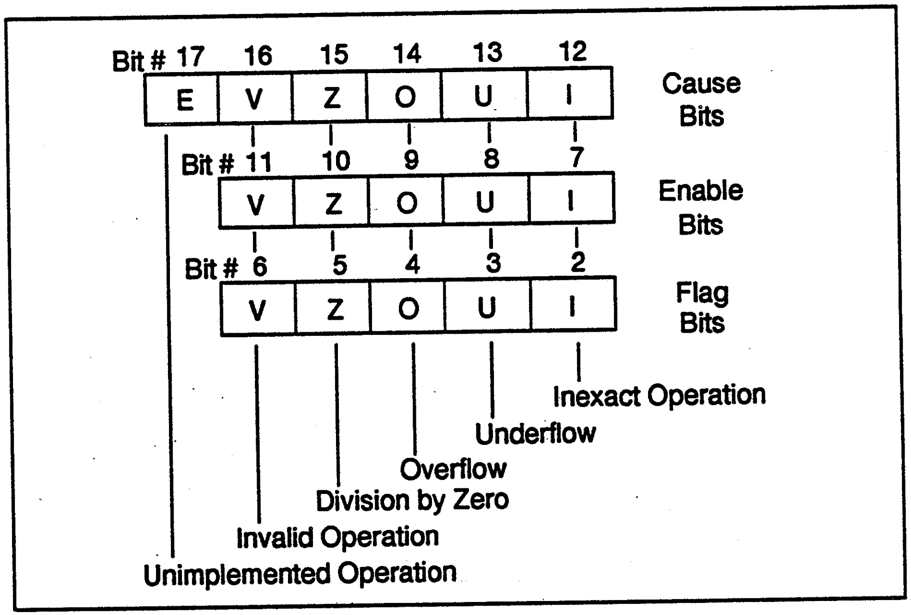


Figure 7-1 Control/Status Register Exception/Flag/Trap/Enable Bits

Each of the five IEEE standard exceptions (V, Z, O, U, I) is associated with a trap under user control, which is enabled by setting one of the five Enable bits. When an exception occurs, both the corresponding Cause and Flag bits are set. If the corresponding Enable bit is set, the FPU generates an interrupt to the CPU and the subsequent exception processing allows a trap to be taken.

Exception Trap Processing

When a floating-point exception trap is taken, the *Cause* register indicates that the floating-point coprocessor is the cause of the exception trap. The 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.

Flags

For each IEEE exception, a Flag bit is provided. This Flag bit is set on any occurrence of its corresponding exception condition, with no corresponding exception trap signaled. The Flag bit is reset by writing a new value into the *Status* register; flags can be saved and restored individually, or as a group, by software.

When no exception trap is signaled, a default action is taken by the floating-point coprocessor, which provides a substitute value for the exception-causing result of the floating-point operation. The particular default action taken depends upon the type of exception, and in the case of the Overflow exception, the current rounding mode. Table 7-1 lists the default action taken by the FPU for each of the IEEE exceptions.

Table 7-1 Default FPU Exception Actions

Field	Description	Rounding mode	Default action
V	Invalid operation	ANY	Supply a quiet Not a Number (NaN)
Z	Division by zero	ANY	Supply a properly signed ∞
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 – ∞
U	Underflow exception	ANY	Supply a rounded result
I	Inexact exception	ANY	Supply a rounded result

The FPU detects internally the eight conditions that can cause exceptions. When the FPU encounters one of these unusual situations, it causes either an IEEE exception or an Unimplemented Operation exception (E). Table 7-2 lists the exception-causing situations and contrasts the behavior of the FPU with the requirements of the IEEE standard.

Table 7-2 FPU Exception-Causing Conditions

FPA Internal result	IEEE Stndrd	Trap Enab.	Trap Disab.	Note
Inexact result	I	I	I	Loss of Accuracy
Exponent overflow	O, I*	O, I	O, I	Normalized exponent > Emax
Divide-by-zero	Z	Z	Z	Zero is (exponent = Emin-1, mantissa = 0)
Overflow on convert	V	V	E	Source out of integer range
Signaling NaN source	V	V	E	Quiet NaN source produces quiet NaN result
Invalid operation	V	V	E	0/0, etc.
Exponent underflow	U	E	E	Normalized exponent < Emin
Denormalized source	none	E	E	Exponent = E-1 and mantissa <> 0

* Standard specifies inexact exception on overflow only if overflow trap is disabled.

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

Inexact Exception (I)

The FPU generates the Inexact exception if the rounded result of an operation is not exact or if it overflows.

NOTE: 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, then the FPU uses a coprocessor stall mechanism to execute the instruction. It is impossible, however, for the FPU to predetermine if an instruction will produce an inexact result. Therefore, 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.

Trap Enabled Results: If Inexact exception traps are enabled, the result register is not modified and the source registers are preserved.

Trap Disabled Results: 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:
  $(+ \infty) + (- \infty)$ or $(- \infty) - (- \infty)$
- Multiplication: 0 times ∞, with any signs.
- Division: 0/0, or ∞/∞, with any signs.
- Conversion of a floating-point number to a fixed-point format when an overflow, or operand value of infinity or NaN, precludes a faithful representation in that format.
- Comparison of predicates involving < or > without ?, when the operands are unordered.
- 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 will cause this exception if one or both operands is a signaling NaN.
- Square root: $\sqrt{x}$, 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 754-specified functions implemented in software, such as Remainder: *x* REM *y*, where *y* is zero or *x* is infinite; conversion of a floating-point number to a decimal format whose value causes an overflow, or is infinity or NaN; and transcendental functions, such as ln (–5) or cos–1(3). Refer to **Appendix B** for examples or for routines to handle these cases.

Trap Enabled Results: The original operand values are undisturbed.

Trap Disabled Results: The FPU always signals an Unimplemented exception because it does not create the NaN that the IEEE standard specifies should be returned under these circumstances.

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 non-zero number. Software can simulate this exception for other operations that produce a signed infinity, such as ln(0), sec(π/2), csc(0), or 0^{-1}.

Trap Enabled Results: The result register is not modified, and the source registers are preserved.

Trap Disabled Results: The result, when no trap occurs, is a correctly signed infinity.

Overflow Exception (O)

The Overflow exception is signaled when the magnitude of the rounded floating-point result, if the exponent range were to be unbounded, is larger than the destination format's largest finite number. (This exception also sets the Inexact exception and Flag bits.)

Trap Enabled Results: The result register is not modified, and the source registers are preserved.

Trap Disabled Results: The result, when no trap occurs, is determined by the rounding mode and the sign of the intermediate result (as listed in Table 7-1).

Underflow Exception (U)

Two related events contribute to the Underflow exception:

- The creation of a tiny non-zero result between ±2Emin which can cause some later exception because it is so tiny.
- The extraordinary loss of accuracy during the approximation of such tiny numbers by denormalized numbers.

IEEE Standard 754 permits a choice in the manner in which these events are detected but requires they be detected the same way for all operations.

The IEEE standard specifies that *tininess* may be detected either:

- after rounding (when a nonzero result, computed as though the exponent range were unbounded, would lie strictly between ±2Emin), or
- before rounding (when a nonzero result, computed as though the exponent range and the precision were unbounded, would lie strictly between ±2Emin).

The MIPS architecture requires that tininess be detected after rounding.

Loss of accuracy can be detected as either:

- denormalization loss (when the delivered result differs from what would have been computed if the exponent range were unbounded), or
- 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 inexact result.

Trap Enabled Results: When an underflow trap is enabled, underflow is signaled when tininess is detected regardless of loss of accuracy. If underflow traps are enabled, the result register is not modified, and the source registers are preserved.

Trap Disabled Results: When an underflow trap is not enabled, underflow is signaled (using the underflow flag) only when both tininess and loss of accuracy have been detected. The delivered result might be zero, denormalized, or ±2Emin

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* cause bit and traps. The operand and destination registers remain undisturbed and the instruction is emulated in software. Any of the IEEE 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 properly handle. These include:

- Denormalized operand
- Not a Number operand
- Denormalized result
- Underflow
- Reserved opcodes
- Unimplemented formats
- Operations which are invalid for their format (for instance, CVT.S.S)

NOTE: 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.

Trap Enabled Results: The original operand values are undisturbed.

Trap Disabled Results: This trap cannot not be disabled.

Saving and Restoring State

Sixteen doubleword coprocessor load or store operations save or restore the coprocessor floating-point register state in memory. The remaining 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 coprocessor *Control/Status* register (*FCR31*) is read, and the coprocessor is executing one or more floating-point instruction, 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 one of the pending instructions cannot be completed, the instruction is placed in the *Exception* register, if present, and information indicating the type of exception is placed in the *Control/Status* register. State information in the status word indicates that exceptions are pending when state is restored.

Writing a zero value to the Cause field of *Control* register *31* clears all pending exceptions, permitting normal processing to be restarted 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 the 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.

Trap Handlers for IEEE Standard Exceptions

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

By retrieving an instruction using the processor *EPC* register, the trap handler determines:

- The 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 the 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 the Inexact exception and the Overflow or Underflow exception.

Signal Descriptions

8

This chapter describes the signals used by and in conjunction with the R4000. The signals discussed include:

- System Interface
- Clock/Control Interface
- Secondary Cache Interface
- Interrupt Interface
- Initialization Interface
- JTAG Interface

System Interface

These signals comprise the interface between the R4000 and other components in the system. Signals IvdAck* and IvdErr* are available only on the R4000SC and MC. All other signals are available on all three package configurations.

ExtRqst*: **External request** **Input**
An external agent asserts ExtRqst* to request use of the system interface. The R4000 grants the request by asserting Release*.

IvdAck*: **Invalidate acknowledge** **Input**
An external agent asserts IvdAck* to signal successful completion of a processor invalidate or update request (R4000MC and SC only).

IvdErr*: **Invalidate error** **Input**
An external agent asserts IvdErr* to signal unsuccessful completion of a processor invalidate or update request (R4000MC and SC only).

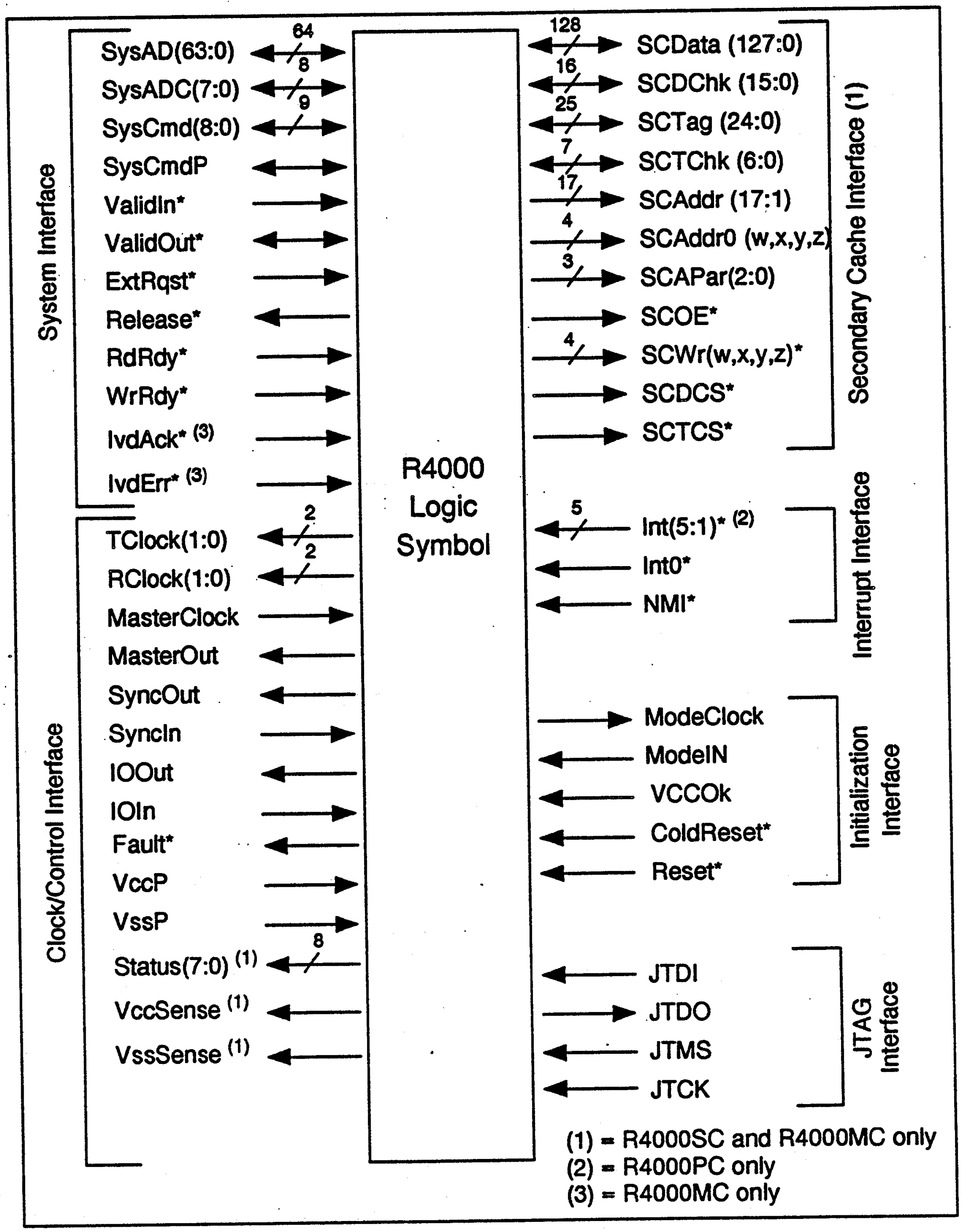


Figure 8-1 R4000 Logic Symbol Diagram

Release*: **Release interface** **Output**
In response to the assertion of ExtRqst*, the R4000 asserts Release* to signal 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, invalidate, or update requests in both secondary-cache and no-secondary-cache mode or can accept a read followed by write request, a read followed by a potential update request, or a read followed by a potential update followed by a write request in secondary cache mode.

SysAD(63:0): **System address/data bus** **Input/Output**
A 64-bit address and data bus for communication between the processor and an external agent.

SysADC(7:0): **System address/data check bus** **Input/Output**
An 8-bit bus containing check bits for the SysAD bus.

SysCmd(8:0): **System command/data identifier bus parity** **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**
A single, even-parity bit for the SysCmd bus.

ValidIn*: **Valid input** **Input**
An 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 R4000 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.

WrRdy*: **Write ready** **Input**
An external agent asserts WrRdy* when it can accept a processor write request.

Clock/Control Interface

These signals comprise the interface for clocking and maintenance functions.

IOOut: **I/O output** **Output**
Output slew rate control feedback loop output. Must be connected to IOIn through a delay loop that models the IO path from the R4000 to an external agent.

IOIn: **I/O input** **Input**
Output slew rate control feedback loop input (see IOOut).

MasterClock: **Master clock** **Input**
Master clock input establishes the processor operating frequency.

MasterOut: **Master clock out** **Output**
Master clock output aligned with MasterClock.

RClock(1:0): **Receive clocks** **Output**
Two identical receive clocks that establish the system interface frequency.

SyncOut: **Synchronization clock out** **Output**
Synchronization clock output. Must be connected to SyncIn through an interconnect that models the interconnect between MasterOut, TClock, RClock, and the external agent.

SyncIn: **Synchronization clock in** **Input**
Synchronization clock input.

TClock(1:0): **Transmit clocks** **Output**
Two identical transmit clocks that establish the system interface frequency.

Fault*: **Fault** **Output**
The R4000 asserts Fault* to indicate a mismatch output of boundary comparators.

Status(7:0): **Status** **Output**
An 8-bit bus that indicates the current operation status of the processor.

VccP: **Quiet VCC for PLL** **Input**
Quiet Vcc for the internal phase locked loop.

VccSense: **VCC sense** **Input/Output**

This is a special pin used only in component testing and characterization. It provides a separate, direct connection from the on-chip VCC node to a package pin without attaching to the in-package power planes. Test fixtures treat VccSense as an analog output pin: the voltage at this pin directly shows the behavior of the on-chip VCC. Thus, characterization engineers can easily observe the effects of di/dt noise, transmission line reflections, etc. VccSense should be connected to VCC in functional system designs.

VssP: **Quiet VSS for PLL** **Input**

Quiet Vss for the internal phase locked loop.

VssSense: **VSS sense** **Input/Output**

VssSense provides a separate, direct connection from the on-chip VSS node to a package pin without attaching to the in-package ground planes. VssSense should be connected to VSS in functional system designs.

Secondary Cache Interface

These signals comprise the interface between the R4000 and the secondary cache. These signals are available only on the R4000MC and SC.

SCAddr(17:1):	**Secondary cache address bus**	**Output**
SCAddr0W:	**Secondary cache address lsb**	**Output**
SCAddr0X:	**Secondary cache address lsb**	**Output**
SCAddr0Y:	**Secondary cache address lsb**	**Output**
SCAddr0Z:	**Secondary cache address lsb**	**Output**

The 18-bit address bus for the secondary cache. Bit 0 has four output lines to provide additional drive current.

SCAPar(2:0): **Secondary cache address parity busOutput**
A 3-bit bus that carries the parity of the SCAddr bus and the cache control lines SCWR*, SCDCS* and SCTCS*. The individual bit definitions are:

SCAPar2 - Even Parity for SCAddr(17:12) and SCWR*

SCAPar1 - Even Parity for SCAddr(11:6) and SCDCS*

SCAPar0 - Even Parity for SCAddr(5:0) and SCTCS*

SCData(127:0): **Secondary cache data bus Input/ Output**
A 128-bit bus used to read or write cache data from and to the secondary cache data RAM.

SCDChk(15:0): **Secondary cache data ECC bus Input/Output**
A 16-bit bus that carries two 8-bit ECC field covering the 128 bits of SCData from/to secondary cache. SCDChk(15:8) corresponds to SCData(127:64) and SCDChk(7:0) corresponds to SCData(63:0).

SCDCS*: **Secondary cache data chip select Output**
Chip select enable signal for the secondary cache data RAM.

SCOE*: **Secondary cache output enable Output**
Output enable for the secondary cache data and tag RAM.

SCTag(24:0): **Secondary cache tag bus Input/Output**
A 25-bit bus used to read or write cache tags from and to the secondary cache.

SCTChk(6:0): **Secondary cache tag ECC bus Input/Output**
A 7-bit bus that carries an Error Checking and Correcting (ECC) field covering the SCTag from and to the secondary cache.

SCTCS*: **Secondary cache tag chip select Output**
Chip select enable signal for the secondary cache tag RAM.

SCWrW*: **Secondary cache write enable Output**

SCWrX*: **Secondary cache write enable Output**

SCWrY*: **Secondary cache write enable Output**

SCWrZ*:	**Secondary cache write enable** Write enable for the secondary cache data and tag RAM.	**Output**

Interrupt Interface

These signals comprise the interface used by external agents to interrupt the R4000 processor. Int*(5:1) is available only on the R4000PC; Int*(0) and NMI* are available on all three configurations.

Int*(5:1): **Interrupt** **Input**
Five of six general processor interrupts, bit-wise ORed with bits 5:1 of the interrupt register.

Int*(0): **Interrupt** **Input**
One of six general processor interrupts, bit-wise ORed with bit 0 of the interrupt register.

NMI*: **Non-maskable interrupt** **Input**
Non-maskable interrupt, ORed with bit 6 of the interrupt register.

Initialization Interface

These signals comprise the interface by which an external agent initializes the R4000 operating parameters. All of these signals are available on all three processor configurations.

ColdReset*: **Cold reset** **Input**
This signal must be asserted for a power on reset or a cold reset. The clocks SClock, TClock, and RClock begin to cycle and are synchronized with the de-assertion edge of ColdReset*. ColdReset* must be de-asserted synchronously with MasterOut.

ModeClock: **Boot mode clock** **Output**
Serial boot-mode data clock output at the system clock frequency divided by two hundred and fifty six.

ModeIn: **Boot mode data in** **Input**
Serial boot-mode data input.

Reset*: **Reset** **Input**
This signal must be asserted for any reset sequence. It may be asserted synchronously or asynchronously for a cold reset, or synchronously to initiate a warm reset. Reset* must be de-asserted synchronously with MasterOut.

VCCOk: **VCC is OK** **Input**
When asserted, this signal indicates to the R4000 that the +5 volt power supply has been above 4.75 volts for more than 100 milliseconds and will remain stable. The assertion of VCCOk initiates the initialization sequence.

JTAG Interface

These signals comprise the interface by which the JTAG boundary scan mechanism is provided.

JTDI: **JTAG data in** **Input**
Data is serially scanned in through this pin.

JTCK: **JTAG clock input** **Input**
The R4000 outputs 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.

JTMS: **JTAG command** **Input**
JTAG command signal, signals that the incoming serial data is command data.

Signal Summary

Table 8-1: R4000 SC/MC Processor Signal Summary

Description	Name	I/O	Asserted State	3-State
Secondary cache data bus	SCData(127:0)	I/O	High	Yes
Secondary cache data ECC bus	SCDChk(15:0)	I/O	High	Yes
Secondary cache tag bus	SCTag(24:0)	I/O	High	Yes
Secondary cache tag ecc bus	SCTChk(6:0)	I/O	High	Yes
Secondary cache address bus	SCAddr(17:1)	O	High	No
Secondary cache address lsb	SCAddr0Z	O	High	No
Secondary cache address lsb	SCAddr0X	O	High	No
Secondary cache address lsb	SCAddr0X	O	High	No
Secondary cache address lsb	SCAddr0W	O	High	No
Secondary cache address parity bus	SCAPar(2:0)	O	High	No
Secondary cache output enable	SCOE*	O	Low	No
Secondary cache write enable	SCWrZ*	O	Low	No
Secondary cache write enable	SCWrY*	O	Low	No
Secondary cache write enable	SCWrX*	O	Low	No
Secondary cache write enable	SCWrW*	O	Low	No
Secondary cache data chip select	SCDCS*	O	Low	No
Secondary cache tag chip select	SCTCS*	O	Low	No
System address/data bus	SysAD(63:0)	I/O	High	Yes
System address/data check bus	SysADC(7:0)	I/O	High	Yes
System command/data identifier bus	SysCmd(8:0)	I/O	High	Yes
System command/data identifier bus parity	SysCmdP	I/O	High	Yes
Valid input	ValidIn*	I	Low	No
Valid output	ValidOut*	O	Low	Yes
External request	ExtRqst*	I	Low	No
Release interface	Release*	O	Low	No
Read ready	RdRdy*	I	Low	No
Write ready	WrRdy*	I	Low	No
Invalidate acknowledge	IvdAck*	I	Low	No
Invalidate error	IvdErr*	I	Low	No
Interrupt	Int*(0)	I	Low	No
Non-maskable interrupt	NMI*	I	Low	No
Boot mode data in	ModeIn	I	High	No
Boot mode clock	ModeClock	O	High	No
JTAG data in	JTDI	I	High	No
JTAG data out	JTDO	O	High	No
JTAG command	JTMS	I	High	No
JTAG clock input	JTCK	I	High	No

Description	Name	I/O	Asserted State	3-State
Transmit clocks	TClock(1:0)	O	High	No
Receive clocks	RClock(1:0)	O	High	No
Master clock	MasterClock	I	High	No
Master clock out	MasterOut	O	High	No
Synchronization clock out	SyncOut	O	High	No
Synchronization clock in	SyncIn	I	High	No
I/O output	IOOut	O	High	No
I/O input	IOIn	I	High	No
VCC is OK	VCCOk	I	High	No
Cold reset	ColdReset*	I	Low	No
Reset	Reset*	I	Low	No
Fault	Fault*	O	Low	No
Quiet VCC for PLL	VccP	I	High	No
Quiet VSS for PLL	VssP	I	High	No
Status	Status(7:0)	O	High	No
VCC sense	VccSense	I/O	N/A	No
VSS sense	VssSense	I/O	N/A	No

Table 8-2: R4000 PC Processor Signal Summary

Description	Name	I/O	Asserted State	3-State
System address/data bus	SysAD(63:0)	I/O	High	Yes
System address/data check bus	SysADC(7:0)	I/O	High	Yes
System command/data identifier bus	SysCmd(8:0)	I/O	High	Yes
System command/data identifier bus parity	SysCmdP	I/O	High	Yes
Valid input	ValidIn*	I	Low	No
Valid output	ValidOut*	O	Low	Yes
External request	ExtRqst*	O	Low	No
Release interface	Release*	O	Low	No
Read ready	RdRdy*	I	Low	No
Write ready	WrRdy*	I	Low	No
Interrupts	Int*(5:1)	I	Low	No
Interrupt	Int*(0)	I	Low	No
Non-maskable interrupt	NMI*	I	Low	No
Boot mode data in	ModeIn	I	High	No
Boot mode clock	ModeClock	O	High	No
JTAG data in	JTDI	I	High	No
JTAG data out	JTDO	O	High	No
JTAG command	JTMS	I	High	No
JTAG clock input	JTCK	I	High	No
Transmit clocks	TClock(1:0)	O	High	No
Receive clocks	RClock(1:0)	O	High	No
Master clock	MasterClock	I	High	No
Master clock out	MasterOut	O	High	No
Synchronization clock out	SyncOut	O	High	No
Synchronization clock in	SyncIn	I	High	No
I/O output	IOOut	O	High	No
I/O input	IOIn	I	High	No
VCC is OK	VCCOk	I	High	No
Cold reset	ColdReset*	I	Low	No
Reset	Reset*	I	Low	No
Fault	Fault*	O	Low	No
Quiet VCC for PLL	VccP	I	High	No
Quiet VSS for PLL	VssP	I	High	No

System Interface

9

The system interface allows the processor to access those external resources required to satisfy cache misses, while also permitting an external agent access to certain of the processor's internal resources.

In the R4000MC configuration, the system interface provides those processor mechanisms necessary to maintain the cache coherency of shared data, while also providing to an external agent the mechanisms with which to maintain system-wide multiprocessor cache coherency.

This section describes the system interface from the point of view of both the processor and the external agent.

System Events

First, a definition: A *system event* is an event that occurs within the processor and requires access to external system resources.

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

System events may be:

- A load that misses in both the primary and secondary caches.
- A store that misses in both the primary and secondary caches.
- A store that hits in either the primary or secondary data cache on a shared line, and an uncached load or store.

Note that a miss in both caches requires the write back to memory of the cache line being replaced, if the line is in a dirty cache state.

Under certain conditions, system events are also caused by cache operation instructions.

Two types of system events are described: processor requests and external requests.

Processor Requests

A processor request is a request or a series of requests, through the system interface, to access some external resource.

Processor requests include read, write, null write, invalidate and update.

- *Read* is a request for a block, double word, word, or partial word of data either from main memory or from another system resource.
- *Write* provides a block, double word, word, or partial word of data to be written either to main memory or to another system resource.
- *Null write* indicates that an expected write has been cancelled as a result of an external request.
- *Invalidate* is a request to invalidate a specified cache line in every other cache in the system.
- *Update* is a request to update every other cache in the system with the specified double word, word, or partial word of data.

External Requests

An external agent requesting access to processor caches or to a processor status register generates an *external request*. This access request passes through the system interface.

External requests include read, write, invalidate, update, snoop, intervention, and null requests. External invalidate, update, snoop and intervention requests, as a group, are referred to as *external coherence requests*.

- *Read* is a request for a word of data from a processor internal resource.
- *Write* provides a word of data to be written to a processor internal resource.

- *Invalidate* specifies a cache line, in the processor's primary and secondary caches, that must be marked invalid.
- *Update* provides a double word, word, or partial word of data to be written to the processor's primary and secondary caches.
- *Snoop* checks the processor secondary cache to see if a valid copy of a particular cache line exists; if the valid copy exists, it then checks to see what cache state the line is in. The processor returns the state of the cache line at the specified physical address in the secondary cache, and may modify the state of the cache line.
- *Intervention* requires the processor to return an indication of the state of the cache line at the specified physical address in the secondary cache. Under certain conditions related to the state of the cache line and the nature of the intervention request, the contents of the primary and secondary cache line may themselves be returned, or the state of the line may itself be modified.
- *Null* requests require no action by the processor. They simply provide a mechanism for an external agent to either return control of the secondary cache to the R4000, or to return the system interface to the master state without affecting the processor.

Read Requests

There are two types of *read requests*: processor and external. When a processor or an external agent receives a read request, it must access the specified resource and return the requested data.

- A processor read request may be split from the external agent's return of the requested data; the response (requested) data may be returned at any time after the read request, provided the system interface bus is not being used. An external agent may even initiate an unrelated external request before it returns the response data for a processor read. A processor read request is complete after the last word of response data has been received from the external agent.

- For external read requests, the data is returned directly in response to the read request. External read requests may not be split from the return of response data. An external read request is complete after the processor returns the requested word of data.

Pending Read Requests

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.

Read Responses

The return of data in response to a processor read request is accomplished through a *read response*. While a read response is technically an external request, read responses has a characteristic that makes it differ from all other external requests—system interface arbitration is not performed. For this reason, read responses are handled separately from all other external requests, and are simply called read responses.

Write Requests

A processor write request is complete after the last word of data has been transmitted. An external write request is complete after the word of data has been transmitted.

Update and Invalidate Requests

A processor update request requires a completion acknowledge by the invalidate acknowledge signal **IvdAck*** or the invalidate error signal **IvdErr*** —unless the update is canceled by the external agent.

Update cancellation is signaled to the processor during external invalidate, update, snoop, and intervention requests; **IvdErr*** is used to signal that a processor update request has failed. When the processor update request fails, the issuing processor takes a bus error on the store instruction that generated the failed request.

Since the completion acknowledge for processor invalidate and update requests is signaled through the system interface on dedicated pins, the completion acknowledge may occur in parallel with processor and external requests. A processor update request that has been submitted, but for which the processor has not yet received an acknowledge or a cancellation, is said to be *unacknowledged*.

An external update request is complete after the request has been transmitted.

Snoop Requests

An external snoop request is complete after the processor returns the state of the specified cache line.

Intervention Requests

An external intervention request is complete after the processor returns the state of the specified cache line, if the processor does not return the contents of the cache line, or after the processor returns the last word of data for the specified cache line.

Note that the data identifier associated with the response data may signal that the returned data is erroneous, causing the processor to take a bus error.

Flow Control for Requests

The processor must manage the flow of processor requests and external requests. The processor controls the flow of external requests by the external request arbitration signals **ExtRqst***, and **Release***. An 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 waiting for the processor to assert **Release*** for one cycle. Mastership of the system interface is always returned to the processor after an external has been issued. The processor will not accept a subsequent external request until it has completed the current one.

Processor requests are managed by the processor in two distinct modes: *secondary-cache mode* and *no-secondary-cache mode*. These modes are programmable through the boot-time mode control interface described in Chapter 12. The allowed modes of operation are dependent on the package configuration for the processor. A processor in the small configuration package must be programmed to run in no-secondary-cache mode. A processor in the large configuration package may be programmed to run in secondary-cache or no-secondary-cache mode. If not programmed appropriately, the behavior of the processor is undefined.

In no-secondary-cache mode, the processor will issue requests in a strict sequential fashion; that is, the processor is only allowed to have one request pending at any time. The processor will issue a read request and wait for a read response before issuing any subsequent requests. The processor will submit a write request only if there are no reads pending.

The processor provides 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, invalidate, and update requests while **WrRdy*** controls the flow of processor write requests. Processor null write requests must always be accepted and cannot be delayed by either **RdRdy*** or **WrRdy***. The processor samples the signal **RdRdy*** to determine the *issue* cycle for a processor read, invalidate, or update request and the processor samples the signal **WrRdy*** to determine the *issue* cycle of a processor write request. The issue cycle for a processor read, invalidate, or update request is defined to be the first address cycle for the request for which the signal **RdRdy*** was asserted two cycles previously. The issue cycle for a processor write request is defined to be the first address cycle for the write request for which the signal **WrRdy*** was asserted two cycles previously. If the processor wishes to issue a request but is unable to because one of the signals **RdRdy*** or **WrRdy*** is deasserted, the processor will repeat the address cycle for the request until the issue cycle is accomplished. Once the issue cycle is accomplished, data transmission will begin for a request that includes data. There will always be one and only one issue cycle for any processor request.

The processor will accept external requests while attempting to issue a processor request by releasing the system interface to slave state in response to an assertion of **ExtRqst***. Note that the rules governing the issue cycle of a processor request are strictly applied to determine the action the processor is taking. The processor will either accomplish the issue of the processor request, in which case the processor request will be completed in its entirety before an external request will be accepted, or the processor will release the system interface to slave state without accomplishing the issue of the processor request. In the latter case, the processor will attempt to issue the processor request again after the external request is completed, and the rules governing issue cycle will again apply.

In no-secondary-cache mode an external agent must be capable of accepting a processor read request at any time there are no processor read requests pending and the signal **RdRdy*** has been asserted for two or more cycles. An external agent must be capable of accepting a processor write request at any time there are no processor read requests pending and the signal **WrRdy*** has been asserted for two or more cycles.

In secondary-cache mode, the processor issues requests both individually as in no-secondary-cache mode and in groups that begin with a processor read request called *clusters*. A cluster consists of a processor read request followed by one or two additional processor requests issued while the read request is pending. All of the requests that are part of a cluster must be accepted before the response to the read request that begins a cluster may be returned to the processor. cluster can include:

- a processor read request, followed by a write request
- a processor read request, followed by potential update
- a processor read request, followed by a potential update request, followed by a write request.

The issue of potential update requests within a cluster can be disabled via the boot-time mode control interface. A processor potential update request is defined as any update request that is issued while a processor read request is pending. In addition, a bit in the command for processor updates identifies potential updates. Potential updates are issued in conjunction with a processor read request. That is, once the processor accomplishes the issue of a read request, a potential update request follows if one is required regardless of the state of **RdRdy***. Potential update requests do not obey the **RdRdy*** flow control rules for issue, but rather issue with a single address cycle regardless of the state of **RdRdy***.

A write request that is part of a cluster does obey the **WrRdy*** rules for issue. The processor accepts external requests between the issue of a processor read request, or a processor read request followed by a potential update request and the issue of a processor write request within a cluster. The processor signals that it is issuing a cluster that contains a processor write request by issuing a *read-with-write-forthcoming* request instead of an ordinary read request to start the cluster. The read-with-write-forthcoming request is identified by a bit in the command for processor read requests. The external agent must accept all of the requests that form a cluster before it may return a response to the read request that began the cluster. The behavior of the processor is undefined if the external agent returns a response to a processor read request that begins a cluster before accepting all of the requests that form the cluster.

Since the processor does accept external requests between the issue of a read-with-write-forthcoming request that begins a cluster and the issue of the write request that completes a cluster, it is possible for an external request to obviate the need for the write request within the cluster. For instance, if the external agent issued an external invalidate request that targeted the cache line the processor was attempting to write back, the state of the cache line would be changed to invalid, and the write back for the cache line would no longer be needed. In this event, the processor issues a processor null write request after completing the external request to complete the cluster. Processor null write requests do not obey the **WrRdy*** flow control rules for issue, but rather issue with a single address cycle regardless of the state of **WrRdy***. Any external request that changes the state of a cache line from dirty exclusive or dirty shared to clean exclusive, shared, or invalid obviates the need for a write back of that cache line.

A processor potential update request remains potential until the response to the pending processor read request that began the cluster is received. If the read response data is returned in one of the shared states, shared or dirty shared, the potential update is no longer potential and must receive an acknowledge via either the signal **IvdAck*** or **IvdErr***. If the read response data is returned in one of the exclusive states, clean exclusive or dirty exclusive, the potential update is nullified and the processor neither expects nor requires an acknowledge.

In secondary-cache mode, an external agent must be capable of accepting a processor read request followed by a potential update request any time there are no processor read requests pending, no unacknowledged processor update requests, and the signal **RdRdy*** has been asserted for two or more cycles. An external agent must be capable of accepting a processor write request at any time there are no processor read requests pending, or there is a processor read-with-write-forthcoming request pending with no unacknowledged processor update requests that are compulsory, and the signal **WrRdy*** has been asserted for two or more cycles.

After issuing a processor read request, the processor does not issue a subsequent read request until it has received a read response for the read request, whether the read request began a cluster or not. After issuing a processor update request, or after a potential update request is no longer potential, the processor does not issue a subsequent request until it has received an acknowledge for the update request. After the processor has issued a write request, the processor does not issue a subsequent request until at least four cycles after the issue cycle of the write request.

The following sections detail the sequence, protocol, and syntax of processor and external requests. 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 processor's system interface pins to realize a processor or external request. Syntax refers to the precise definition of bit patterns on encoded buses such as the command bus.

Processor Request Sequencing

The processor generates a request or a series of requests through the system interface to satisfy system events. Processor requests are managed in two distinct modes, secondary-cache mode and no-secondary-cache mode. The following sections detail the sequence of requests generated by the processor for each system event in secondary-cache and no-secondary-cache mode.

Primary and Secondary Cache Miss on a Load

When the processor misses in both the primary and secondary caches on a load, it must obtain the cache line that contains the data element to be loaded from an external agent before it can proceed. If the new cache line will replace a current cache line that is in the state dirty exclusive or dirty shared, the current cache line must be written back before the new line can be loaded in the primary and secondary caches.

The processor examines the coherency attribute in the TLB entry for the page that contains the requested cache line, and, if the coherency attribute is exclusive, it issues a coherent read request that also requests exclusivity. If the coherency attribute is sharable or update, the processor issues a coherent read request, and, if the coherency attribute is noncoherent, the processor issues a noncoherent read request.

In no-secondary-cache mode, the processor issues a read request for the cache line that contains the data element to be loaded. The processor then waits for an external agent to provide the read data in response to the read request. Then, if the current cache line must be written back, the processor issues a write request for the current cache line.

In secondary-cache mode, if the current cache line does not need to be written back and the coherency attribute for the page that contains the requested cache line is anything other than exclusive, the processor issues a read request for the cache line that contains the data element to be loaded. If the current cache line needs to be written back and the coherency attribute for the requested cache line is not exclusive, the processor will issue a cluster consisting of a read-with-write-forthcoming request for the cache line that contains the data element to be loaded followed by a write request for the current cache line. If the current cache needs to be written back, and the coherency attribute for the page containing the requested cache line is exclusive, the processor issues a cluster consisting of an exclusive read-with-write-forthcoming request, followed by a write request for the current cache line.

Primary and Secondary Cache Miss on a Store

When the processor misses in both the primary and secondary caches on a store, it must obtain the cache line that contains the target location of the store from an external agent before it can proceed. In secondary cache mode, if the new cache line replaces a current cache line that is in the state dirty exclusive or dirty shared, the current cache line must be written back before the new line can be loaded in the primary and secondary caches.

The processor examines the coherency attribute in the TLB entry for the page that contains the requested cache line to see if this cache line is being maintained with a write invalidate or a write update cache coherency protocol. If the coherency attribute is sharable or exclusive, a write invalidate protocol is in effect, and a coherent read that also requests exclusivity is issued. If the coherency attribute is update, a write update protocol is in effect and a coherent read request is issued. If the coherency attribute is noncoherent, a noncoherent read request is issued.

In no-secondary-cache mode, the processor issues a read request for the cache line that contains the data element to be loaded. The processor then waits for an external agent to provide the read data in response to the read request. Then, if the current cache line must be written back, the processor issues a write request for the current cache line.

In secondary-cache mode, if the current cache line does not need to be written back and the coherency attribute for the page that contains the requested cache line is noncoherent, the processor issues a read request for the cache line that contains the target location of the store. If the current cache line does not need to be written back and the coherency attribute for the page that contains the requested cache line is sharable or exclusive, the processor issues a read request. If the current cache line does not need to be written back andthe coherency attribute for the page that contains the requested cache line is update, and potential updates are enabled, the processor issues a cluster consisting of a read request followed by a potential update request. If the current cache line needs to be written back, and the coherency attribute for the requested cache line is noncoherent, the processor issues a cluster consisting of a read-with-write-forthcoming request for the cache line that contains the target location of the store followed by a write request for the current cache line. If the current cache line needs to be written back and the coherency attribute for the page that contains the requested cache line is sharable or exclusive, the processor issues a cluster consisting of a read-with-write-forthcoming request followed by a write request for the current cache line. If the current cache line needs to be written back and the coherency attribute for the page that contains the requested cache line is update, and potential updates are enabled, the processor issues a cluster consisting of a read-with-write-forthcoming request followed by a potential update request followed by a write request for the current cache line.

If the processor issues a cluster that contains a potential update, and the response data for the read request is returned with an indication that it must be placed in the cache in a shared state, either shared or dirty shared, the potential update becomes compulsory. Once a potential update becomes compulsory, the external agent must forward the update to the system, and signal an acknowledge to the processor when the update is complete. In this case the processor will not complete the store until the update has been acknowledged.

If the processor issues a cluster that contains a potential update, and the response data for the read request is returned in an exclusive state, clean exclusive or dirty exclusive, the potential update is nullified. Once a potential update has been nullified, the external agent must simply discard the update. The processor will not wait for or expect an acknowledge to a potential update that has been nullified.

If the processor issues a read request, or a cluster that does not contain a potential update, and the response data for the read request is returned with an indication that it must be placed in the cache in a shared state, either shared or dirty shared, the processor will then issue an invalidate request or an update request depending on the coherency attribute for the page that contains the target location of the store instruction. If the coherency attribute is update, an update request is issued, otherwise an invalidate request is issued. The external agent must forward the update to the system and signal an acknowledge to the processor for the update request. The processor will not complete the store until it has received an acknowledge for the update request.

The concept of potential updates is introduced to provide the external agent a chance to use the system bus more efficiently. In an update protocol, it is quite likely that a cache line requested by a processor coherent read request will be returned in a shared state, and that the processor will then have to issue an update request before it can complete a store instruction. The potential update issued with the read request in a cluster allows the external agent to anticipate the read response on the system bus, and, if it arrives with an indication that it is shared, to quickly gain control of the system bus and transmit the required update to the rest of the system. This provides the processor with the acknowledge as quickly as possible and also allows the processor to complete the store instruction as quickly as possible. Without the potential update request, the response data must be returned to the processor. The processor then issues an update request which must then be forwarded to the system bus before an acknowledge can be returned to the processor.

Note that potential updates behave in all cases as if they have not yet been issued by the processor. Potential updates are not subject to cancellation, and do not expect or require an acknowledge. When a potential update is nullified, the processor behaves as if no update request was ever issued. When a potential update is no longer potential, the processor behaves as if it had issued an update request at that instant. Once a potential update is no longer potential it is subject to cancellation, and the processor requires an acknowledge for the update request.

Secondary Cache Hit on a Store to a Shared Line

When the processor hits in the secondary on a cache line that is marked shared or dirty shared, the processor must issue an update request and wait to receive an acknowledge before the store can be completed. The processor checks the coherency attribute in the TLB for the page that contains the cache line that is the target of the store to determine if the cache line is being managed using a write invalidate or write update cache coherency protocol. If the coherency attribute is sharable or exclusive, a write invalidate protocol is in effect and the processor issues an invalidate request. If the coherency attribute is update, a write update protocol is in effect and the processor issues an update request. The processor will not complete the store until an external agent signals an acknowledge for the update request.

Uncached Load or Store

When the processor performs an uncached load, it issues a noncoherent read request. When the processor performs an uncached store, it issues a write request.

Cache Operations

The processor provides a variety of cache operations for use in maintaining the state and contents of the primary and secondary caches. During the execution of the cache operation instructions, the processor may issue write requests or invalidate requests.

External Request Handling

An external agent must arbitrate with the processor for access to the system interface before it can issue an external request. The external agent signals that it wishes to begin an external request and waits for the processor to signal that it is ready to accept the request before issuing any new external requests. Based on its internal state and the current state of the system interface, the processor decides when to accept a new external request. The processor signals that it is ready to accept an external request based on the following criteria:

1. 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 may issue a new processor request while the external agent is requesting access to the system interface to issue an external request.
2. The processor will accept an external request after completing a processor request or a processor request cluster that is in progress.
3. While waiting for the assertion of **RdRdy*** to issue a processor read request, the processor will accept an external request provided that the request is delivered to the processor one or more cycles before **RdRdy*** is asserted.
4. While waiting for the assertion of **WrRdy*** to issue a processor write request, the processor will accept an external request provided that the request is delivered to the processor one or more cycles before **WrRdy*** is asserted.
5. While waiting for the response to a read request after the processor has made an uncompelled change to a slave state, an

external agent may issue an external request before providing the read response data.

Invalidate and Update Cancellation

An external agent may discover that a processor request for an update cannot be completed based on state changes in the external system that have not yet been reflected into the processor's caches. An example of this in a bus-based system is when a processor issues an invalidate, but, before the external bus interface can transmit the invalidate, an invalidate is received from another processor that targets the same cache line. In this case, the processor's cache does not reflect the current state of the system, and the unacknowledged invalidate cannot be transmitted. When this occurs, the external agent must cancel the update. The processor, upon receiving a cancellation, will process any external requests that the external agent wishes to issue and then re-examine the state of the cache to determine what action to take.

In the previous example, this would cause the processor to process an external request to invalidate the cache line that was the target of the store. The processor would then re-examine the state of the cache and discover that the cache line that was the target of the store is now invalid. Finally, the processor processes the store as a store miss and issues a read request instead of an invalidate request.

Potential updates may not be canceled until they become compulsory. Potential updates are issued within a cluster under pending reads and become compulsory after the read request is satisfied. In more general terms, an external request that indicates processor update cancellation may not be issued when a processor read is pending and may not be issued unless a compulsory update is unacknowledged. The behavior of the processor is undefined if the cancellation indication is signaled on an external coherence request to the processor while a processor read is pending or when there is no compulsory update unacknowledged.

Load Linked Store Conditional Considerations

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 for which the execution of a Load Linked Store Conditional instruction sequence is visible as a change in the nature of a processor read request. Specifically, if the data location targeted by a Load Linked Store Conditional instruction sequence maps to the same cache line that the instruction area containing the Load Linked Store Conditional code sequence is mapped to, then immediately after executing the Load Linked instruction the cache line that contains the link location will be 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 remains set. The link bit is set by the Load Linked instruction, and is cleared by any change of cache state for the cache line containing the link address, or a return from exception.

In order for the Load Linked Store Conditional instruction sequence to work correctly, all coherency traffic targeting the link address must be visible to the processor, and the cache line containing the link location must remain in a shared state in every cache in the system. This guarantees that a Store Conditional executed by some other processor is visible to the processor as a coherence request which changes the state of the cache line that contains the link location. To accomplish this, a read request issued by the processor which causes the replacement of a cache line that contains the link location while the link bit is set will indicate that the link address is being retained. The link address retained bit in the command for the read request will be asserted to provide this indication. This informs the external agent that even though the processor has replaced this cache line and no longer has it present in its cache, it still must see any coherence traffic that targets this cache line.

In addition, any snoop or intervention request that targets a cache line which is not present in the cache, but for which the snoop or intervention address matches the current link address while the link bit is set, will return an indication that the cache line is present in the cache in a shared state. A shared indication is returned even though the processor does not actually have the data content of the cache line. This is consistent since the processor never returns data in response to an intervention request for a cache line that is in the shared state. The shared response guarantees that the cache line that contains the link location will remain in a shared state in all other processor's caches, and therefore that any other processor attempting a store conditional to this link location must issue a coherence request in order to complete the store conditional.

System Interface Protocol

The system interface protocol describes the cycle-by-cycle signal transitions that occur on the pins of the system interface to realize requests between the processor and an external agent.

Introduction

The system interface is register to register. That is, processor outputs come directly from output registers and begin to change with the rising edge of SClock (SClock is an internal clock used by the processor to sample data at the system interface and to clock data into the processor's system interface output registers; see Chapter 10, *"Clock/Control Interface"* for more details), and processor inputs are fed directly to input registers that latch the inputs with the rising edge of SClock. Therefore, if an input to the processor is changed during a particular cycle in such a way that the new value is sampled at the end of the cycle, the earliest the processor can change one of its outputs in response to the input change is two cycles later. This methodology was chosen to allow the system interface to run at the highest possible clock frequency.

The primary communication paths for the system interface are a sixty-four bit address and data bus, **SysAD(63:0)** and a nine bit command bus, **SysCmd(8:0)**. The **SysAD** bus and the **SysCmd** bus are bi-directional; that is, they are driven by the processor to issue a processor request, and by an external agent to issue an external request. When the processor is driving the **SysAD** bus and the **SysCmd** bus, the system interface is in *master state*. When an external agent is driving the **SysAD** bus and the **SysCmd** bus, the system interface is in *slave state*.

A request through the system interface consists of an address, a system interface command that specifies the precise nature of the request, and a series of data elements if the request is for a write, read response, or update. Addresses and data elements are transmitted on the **SysAD** bus. System interface commands are transmitted on the **SysCmd** bus.

Cycles in which the **SysAD** bus contains a valid address are called *address cycles*. Cycles in which the **SysAD** bus contains a valid data element are called *data cycles*. In master state the processor will assert the signal **ValidOut*** whenever the **SysAD** bus and the **SysCmd** bus are valid. In slave state an external agent will assert the signal **ValidIn*** whenever the **SysAD** bus and the **SysCmd** bus are valid.

The **SysCmd** bus is used to identify the contents of the **SysAD** bus during any cycle in which it is valid. 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, the remainder of the **SysCmd** bus, **SysCmd(7:0)**, contains a sustem interface command. The encoding of system interface commands is detailed in the section on system interface syntax. During data cycles, the remainder of the **SysCmd** bus, **SysCmd(7:0)**, contains an indication of whether this is the last data element to be transmitted and other information about the data element. The content of the **SysCmd** bus during data cycles is called a *data identifier*. The encoding of data identifiers is detailed in the section on system interface syntax.

A request through the system interface consists of one or more identical address cycles, followed by a series of data cycles for requests that include data. The most efficient request through the system interface consists of a single address cycle followed by a single data cycle or a number of data cycles sufficient to transmit a block of data.

System Interface Arbitration

When an external agent needs to issue an external request through the system interface, it must first get the system interface into slave state. The transition from master state to slave state is arbitrated by the processor using the system interface handshake signals **ExtRqst*** and **Release***. An external agent signals that it wishes to issue an external request by asserting **ExtRqst***. Then, when the processor is ready to accept an external request, it releases the system interface from master state to slave state by asserting **Release*** for one cycle. The system interface returns to master state as soon as the issue of the external request is completed. Having asserted **ExtRqst***, an external agent may not de-assert **ExtRqst*** until the processor asserts **Release***. After the processor asserts **Release*** for one cycle, the external agent must deassert **ExtRqst*** two cycles after the assertion of **Release***. An external agent may continue to assert **ExtRqst*** if another external request follows the current request. After the first external request completes, the processor must assert **Release*** again before the second external request is issued to the processor.

The system interface will remain in master state until the external agent requests and is granted the system interface or until the processor issues a read request, or completes the issue of a cluster. Whenever a processor read request is pending, after the issue of a read request or after the issue of all of the requests in a cluster, the processor will switch the system interface to slave state even though the external agent is not arbitrating to issue an external request. This transition to slave state is specifically to allow the external agent to return read response data. The external agent must not assert the signal **ExtRqst*** for the purposes of returning read response data. **ExtRqst*** should only be asserted when the external agent needs to get the system interface into slave state to issue an external request.

The signal **ExtRqst*** is ***only*** used to arbitrate for the system interface, that is to request the transition of the system interface from master state to slave state. **ExtRqst*** must always de-assert two cycles after a cycle in which **Release*** is asserted unless the external agent wishes to perform a subsequent external request. **ExtRqst*** must not be asserted while the system interface is in slave state unless the external agent wishes to perform a subsequent external request.

The transition of the system interface from master state to slave state initiated by the processor when a processor read request is pending will be referred to as an *uncompelled* change to slave state. An uncompelled change to slave state will occur during or some number of cycles after the issue cycle of a read request or the last cycle of the last request in a cluster. The number of cycles depends on the state of the cache, the presence of a secondary cache and the secondary cache parameters. After an uncompelled change to slave state, the system interface remains in slave state until the external agent issues an external request. After the external request, the system interface will return to master state. An external agent must note that the processor has performed an uncompelled change to slave state and begin driving the address and data bus and the command 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***.

System Interface Request Descriptions

The following sections illustrate, through the use of text and detailed timing diagrams, the protocol of each processor and external request. The timing diagrams use abbreviations to show the contents of encoded busses during cycles in which they are defined. Following is a list of abbreviations and definitions used for each bus.

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 read request command.
RwWF – A read-with-write-forthcoming request command.
Write – A write request command.
Null – A null request command.
SINull – A system interface release null request command.
SCNull – A secondary cache release null request command.
Ivd – An invalidate request command.
Upd – An update request command.
Ivtn – An intervention request command.
Snoop – A snoop 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.
CData – A coherent data identifier for a data element other than the last data element.
CEOD – A coherent data identifier for the last data element.

Two closely spaced wavy vertical lines in a timing diagram indicate a repetition of the current cycle. That is, the cycle broken by the wavy lines may represent one or more identical cycles. This is referred to as a break in the timing diagram and is used to keep the timing diagrams concise and readable.

Invalidate and Update Acknowledge Protocol

Processor invalidate and update requests are acknowledged using the signals **IvdAck*** and **IvdErr***. An external agent drives either **IvdAck*** or **IvdErr*** for one cycle to signal the completion status of the current processor update request. update request acknowledge occurs in parallel with requests on the SysAD and SysCmd buses. **IvdAck*** or **IvdErr*** may be driven at any time after a processor update request is issued provided that the update request is compulsory.

Arbitration Protocol

System interface arbitration is implemented using the signals **ExtRqst*** and **Release***. When an external agent wishes to submit an external request, it asserts **ExtRqst***. The processor waits until it is ready to handle an external request and then assert **Release*** for one cycle before it tri-states the SysAD bus and SysCmd bus. The external agent begins driving the SysAD bus and the SysCmd bus two cycles after a cycle in which **Release*** is asserted. The external agent always deasserts **ExtRqst*** two cycles after a cycle in which **Release*** is asserted unless the external agent wishes to perform a subsequent external request. The external agent always releases the **SysAD** bus and the **SysCmd** bus at the completion of an external request.

The processor will assert **Release*** for one cycle as a processor read request is issued or sometime after a processor read request is issued to perform an uncompelled change to slave state. An external agent must begin driving the **SysAD** bus and the **SysCmd** bus two cycles after the cycle in which **Release*** is asserted. After an uncompelled change to slave state, the processor will return to master state at the end of the next external request, which may be the read response, or may be some other external request.

The processor to system handshake for external requests is illustrated in Figure 9-1.

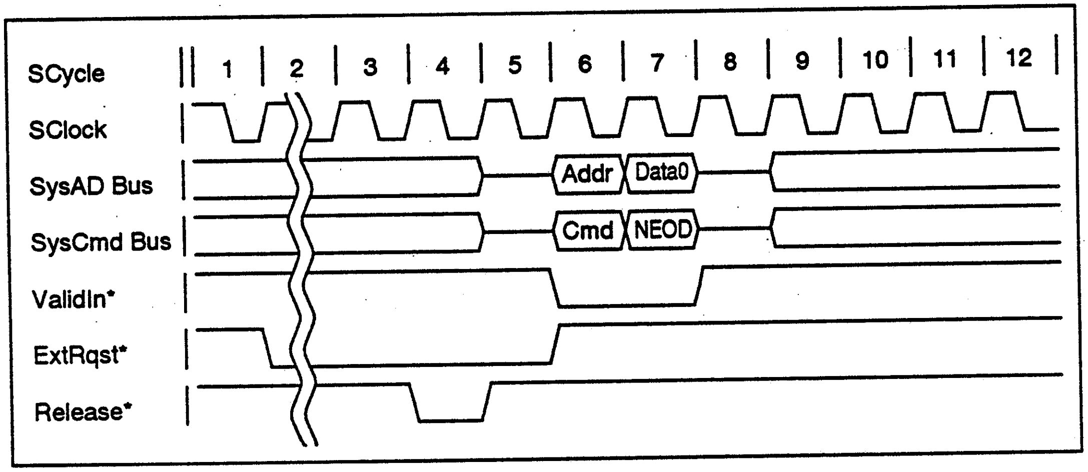


Figure 9-1 Arbitration Protocol for External Requests

Processor Read Request Protocol

A processor read request is issued, with the system interface in master state, by driving a read command on the **SysCmd** bus and a read address on the **SysAD** bus and asserting **ValidOut*** for one cycle. Only one processor read request may be pending at a time. The processor must wait for and retire an external read response before initiating a subsequent read.

The processor makes an uncompelled change to slave state either at the issue cycle of the read request or sometime after the issue cycle of the read request by asserting the **Release*** signal for one cycle. Once in slave state, an external agent may return the requested data via a read response. An 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*** may be asserted before or during a read response for the purposes of performing an external request other than a read response.

When a read is pending, **ExtRqst*** is asserted, and **Release*** is asserted for one cycle, it may be unclear if this assertion of **Release*** is in response to **ExtRqst***, or represents an uncompelled change to slave state. The only situation in which this assertion of **Release*** may not be considered an uncompelled change to slave state is if the system interface is operating in secondary-cache mode, the read request was a read-with-write-forthcoming request, and the expected write request has not yet been issued by the processor. In this case, the write request must be accepted by the external agent before the read response can be issued. In all other cases, the assertion of **Release*** may be considered to be an uncompelled change to slave state or to be in response to the assertion of **ExtRqst***. In this situation, the processor will accept either a read response, or any other external request. If an external request other than a response requestis issued, the processor will perform another uncompelled change to slave state after processing of the external request is complete.

The response request may either 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 was erroneous, it will cause the processor to take a bus error.

A processor read request and an uncompelled change to slave state occurring as the read request is issued is illustrated in Figure 9-2. A processor read request and the subsequent uncompelled change to slave state occurring sometime after the read request is issued is illustrated in Figure 9-3.

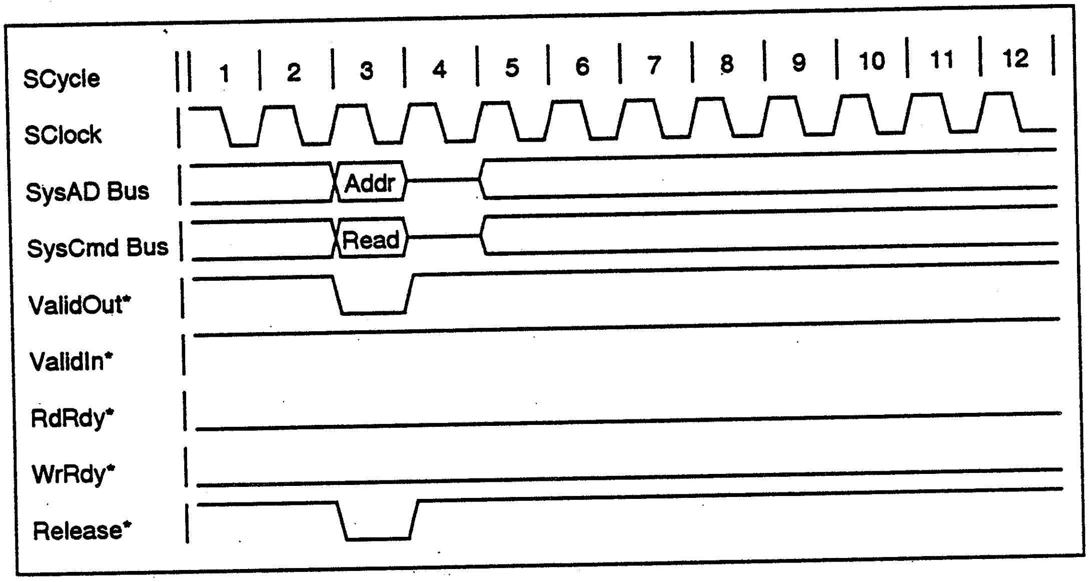


Figure 9-2 Processor Read Request Protocol

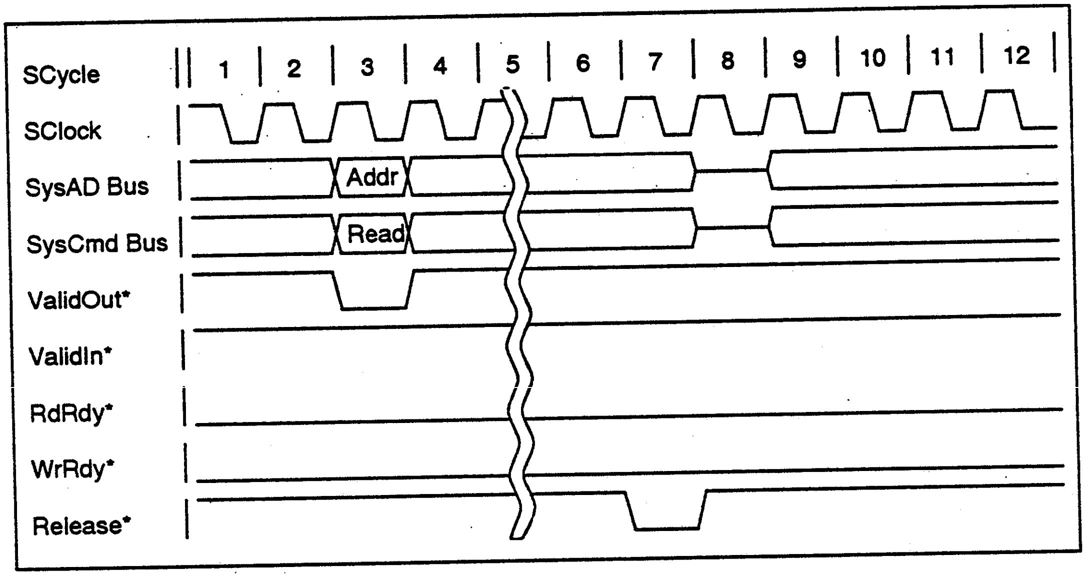


Figure 9-3 Processor Read Request Protocol, Change to Slave State Delayed

Processor Write Request Protocol

Processor write requests are issued with one of two protocols. Double word, word, and partial word writes use a single word write request protocol. Write requests for a block of data use a block write request protocol. Processor write requests are issued with the system interface in master state.

A processor single word write request is issued by driving a write command on the SysCmd bus and a write address on the SysAD bus and asserting **ValidOut*** for one cycle. This is followed by driving a data identifier on the SysCmd bus and data on the SysAD bus and asserting **ValidOut*** for one cycle. The data identifier associated with the data cycle must contain a last data cycle indication.

A processor coherent or noncoherent block write request is issued by driving a write command on the SysCmd bus and a write address on the SysAD bus and asserting **ValidOut*** for one cycle. This is followed by driving a data identifier on the SysCmd bus and data on the SysAD bus and asserting **ValidOut*** for a number of cycles sufficient to transmit the block of data. The data identifier associated with the last data cycle must contain a coherent or noncoherent last data cycle indication. The first data cycle may not immediately follow the address cycle. A processor noncoherent single word write request is illustrated in Figure 9-4. A processor coherent block request for eight words of data is illustrated in Figure 9-5 and Figure 9-6.

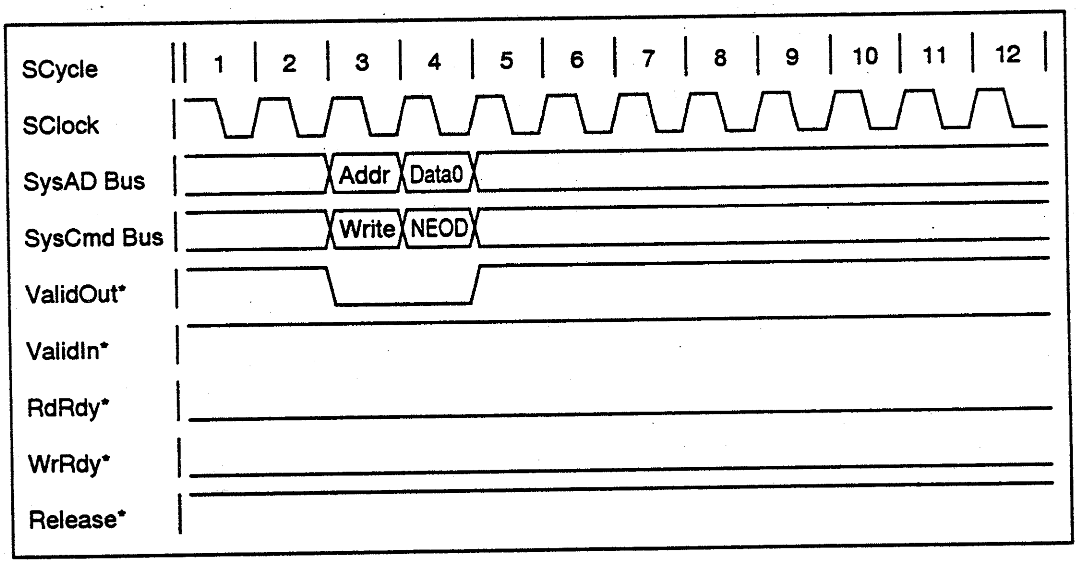


Figure 9-4 Processor Noncoherent Single Word Write Request Protocol

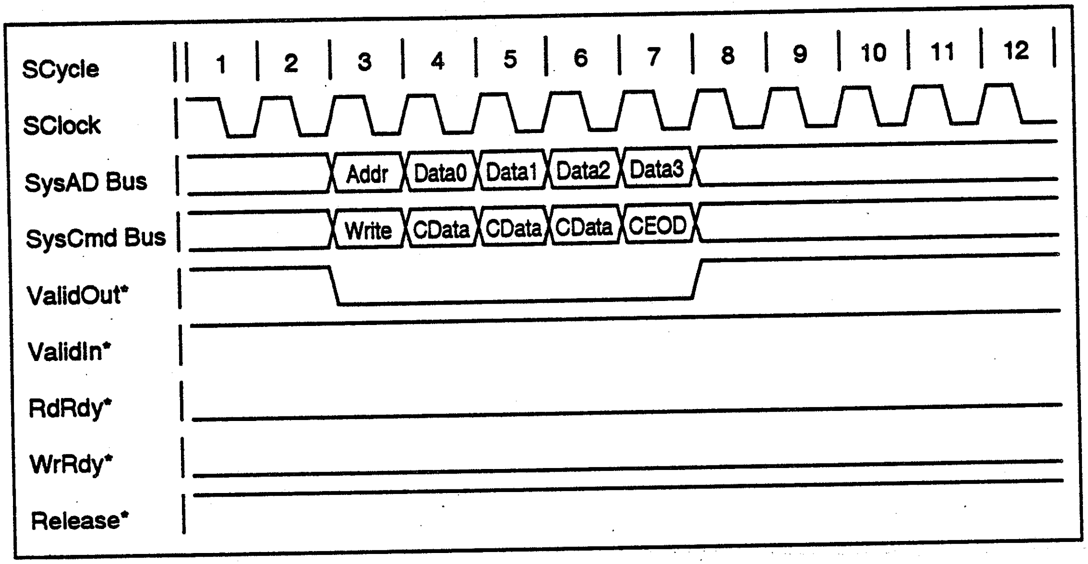


Figure 9-5 Processor Coherent Block Write Request Protocol

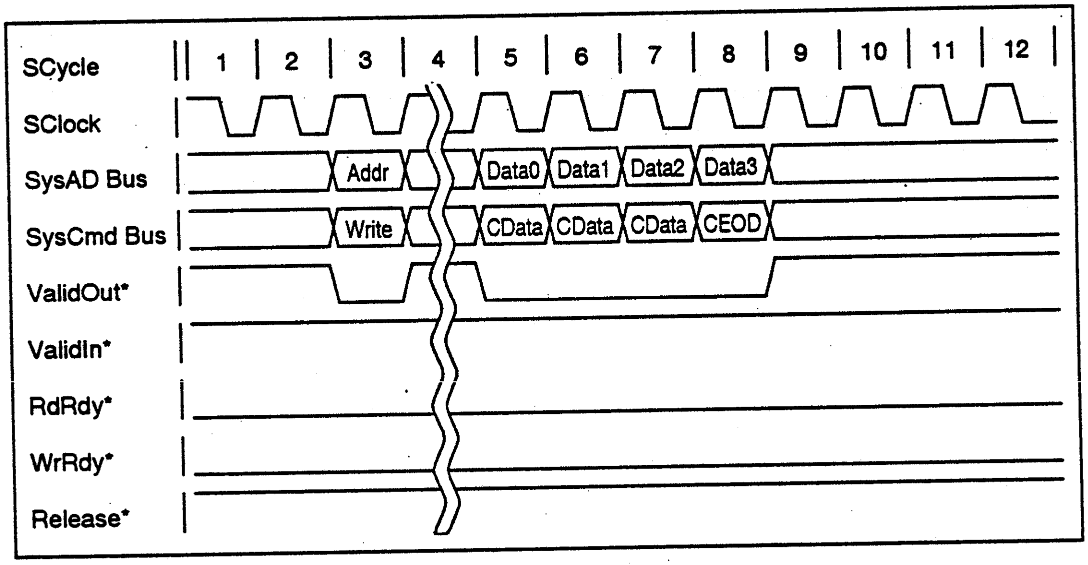


Figure ?-6 Processor Coherent Block Write Request Protocol

Processor Invalidate and Update Request Protocol

A processor invalidate request or update request will use the same protocol as a coherent word write request except that the command associated with the address cycle will indicate that this is an update request. The single data cycle will be unused for an invalidate.

Processor Null Write Request Protocol

A processor null write request is issued with the system interface in master state by driving a null command on the SysCmd bus and asserting **ValidOut*** for one cycle. The SysAD bus is unused during the address cycle associated with a null write request. Processor null write requests cannot be flow controlled with either **RdRdy*** or **WrRdy***, but rather always issue with a single address cycle. A processor null write request is illustrated in Figure 9-7.

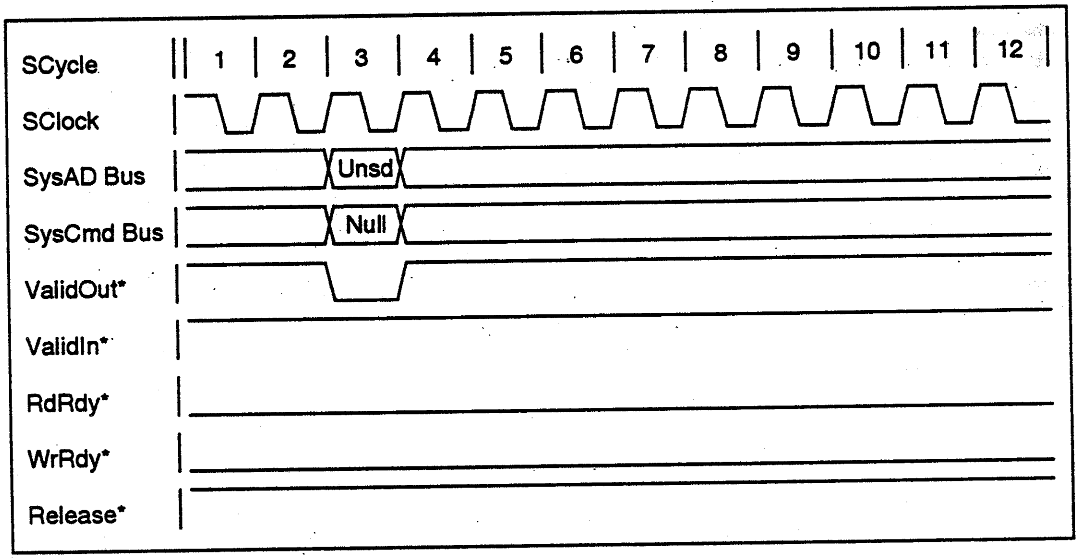


Figure 9-7 Processor Null Write Request Protocol

Processor Cluster Protocol

In secondary-cache mode, the processor issues requests both individually, as in no-secondary-cache mode, and in groups that begin with a processor read request called *clusters*. A cluster consists of a processor read request followed by one or two additional processor requests issued while the read request is pending. All of the requests that are part of a cluster must be accepted before the response to the read request that begins the cluster may be returned to the processor. A cluster includes a processor read request followed by a write request, or a processor read request followed by a potential update request, or a processor read request followed by a potential update request, followed by a write request.

The protocol of each of the requests that form a cluster is as described above. The number of unused cycles between the requests that form a cluster may be zero or greater. The processor makes an uncompelled change to slave state either during or following the last cycle of the last request in the cluster. A cluster consisting of a read request, followed by an update request, followed by a coherent block write request for eight words of data with minimum spacing between the requests that form the cluster, and an uncompelled change to slave state at the earliest opportunity is illustrated in Figure 9-8.

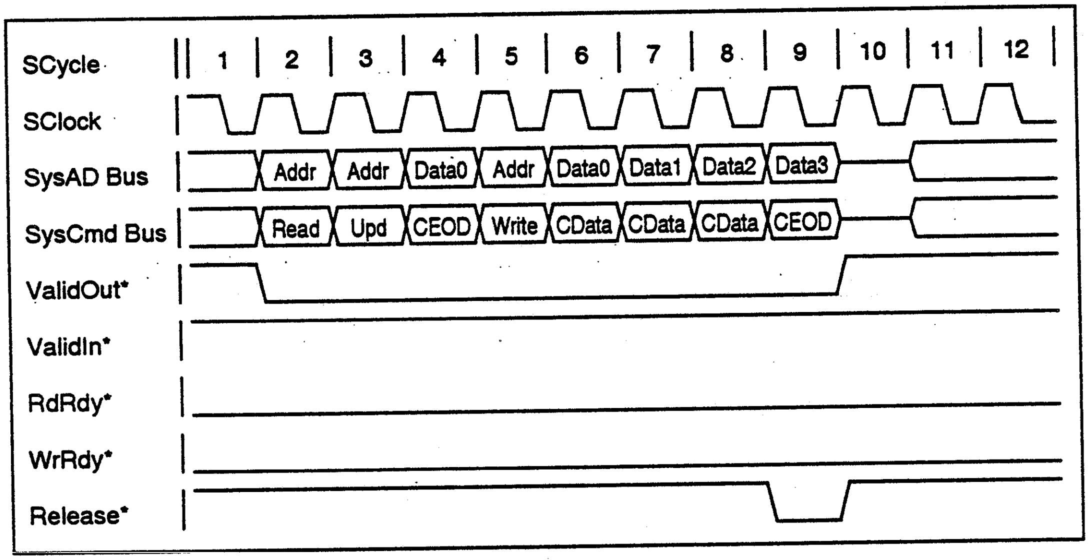


Figure 9-8 Processor Cluster Protocol

External Request Protocol

External requests may only be issued with the system interface in slave state. An external agent must assert **ExtRqst*** to arbitrate for the system interface, and wait for the processor to release the system interface to slave state before issuing an external request. If the system interface is already in slave state; i.e., the processor has previously performed an uncompelled change to slave state, an external agent may begin an external request immediately.

After issuing an external request, an 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*** is asserted. An external agent may hold **ExtRqst*** asserted if it needs to issue a string of external requests, but it must wait for the processor to assert **Release*** and return the system interface to slave state before it proceeds with the next external request. For a string of external requests, the external agent must de-assert **ExtRqst*** two cycles after the cycle in which **Release*** is asserted for the last external request in the string. The processor will continue to handle external requests as long as **ExtRqst*** is held asserted; however, the processor will not release the system interface to slave state for a subsequent external request until it has completed the current request. A string of external requests will not be interrupted by a processor request as long as **ExtRqst*** is held asserted throughout the issue of the string of external requests.

External Read Request Protocol

External reads are requests for a word of data from some processor internal resource. External read requests use a non-split protocol that does not allow any other request to occur at the system interface between the external read request and the read response. The protocol of an external read request encompasses the request from an external agent and the response from the processor.

An external read request consists of driving a read request command on the **SysCmd** bus and a read request address on the **SysAD** bus and asserting **ValidIn*** for one cycle. After the address and command are sent, the external agent releases the **SysCmd** and **SysAD** buses and allows the processor to begin driving them. The processor accesses the data that is the target of the read and returns the data to the external agent. The processor accomplishes this by driving a data identifier on the **SysCmd** bus, the response data on the **SysAD** bus, and asserting **ValidOut*** for one cycle. The data identifier indicates that this is response data and that it contains a last data cycle indication. To transition the system interface back to master state, the processor continues driving the **SysCmd** and **SysAD** buses after the read response is returned.

External read requests are only allowed to read a word of data from the processor. The processor response to external read requests for any data element other than a word is undefined.

An external read request with the system interface initially in master state is illustrated in Figure 9-9.

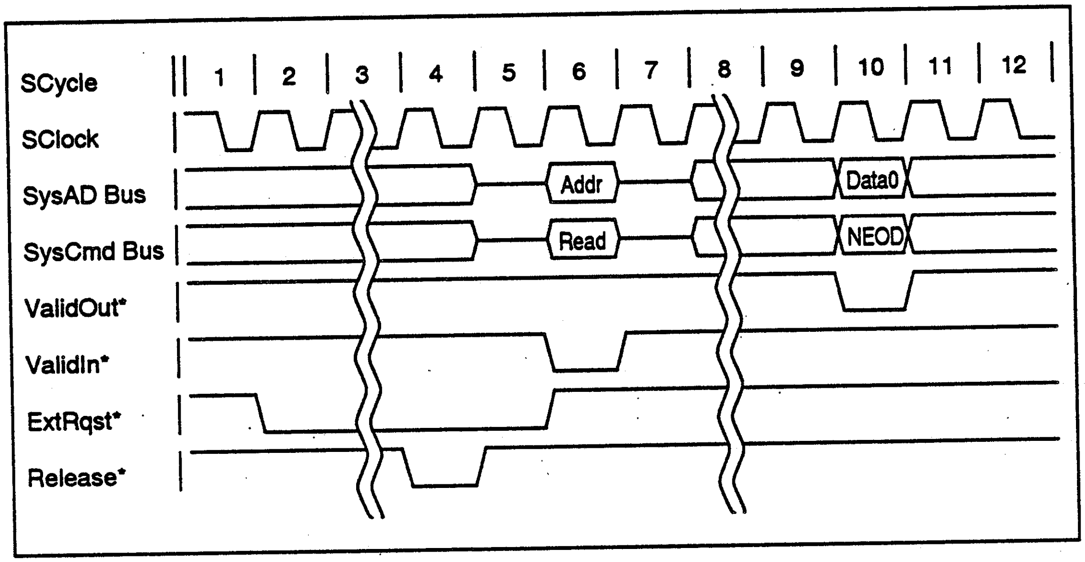


Figure 9-9 External Read Request, System Interface in Master State.

NOTE: The R4000 does not contain any resources that are readable with an external read request. The R4000 returns a bus error response to any external read request.

External Null Request Protocol

The processor supports two kinds of external null requests. A *system interface release external null request* is used to return the system interface to master state after it has been released to slave state without affecting the processor. An *scache release external null request* is used to return ownership of the secondary cache to the processor while the system interface remains in slave state for some period of time. This is important since any time the processor releases the system interface to slave state to accept an external request, it also acquires ownership of the secondary cache for use by the external request in anticipation of handling a coherence request. When an external agent requests ownership of the system interface for the purposes of using the SysAD bus for a transfer unrelated to the processor (such as DMA), this ownership of the secondary cache prevents the processor from satisfying subsequent primary cache misses. The scache release external request can be issued by the external agent to return ownership of the secondary cache to the processor. External null requests require no action from the processor other than to return the system interface to master state or to regain ownership of the secondary cache.

An external null request consists of driving a null request command on the **SysCmd** bus and asserting **ValidIn*** for one cycle. The **SysAD** bus is unused (does not contain valid data) during the address cycle associated with an external null request. After the address cycle is issued the null request is complete. For a system interface release external null request, the external agent releases the **SysCmd** and **SysAD** buses and allows the system interface to return to master state. For an scache release external null request, the system interface remains in slave state. An scache release external null request with the system interface initially in master state is illustrated in Figure 9-10. A system interface release external null request with the system interface initially in slave state is illustrated in Figure 9-11.

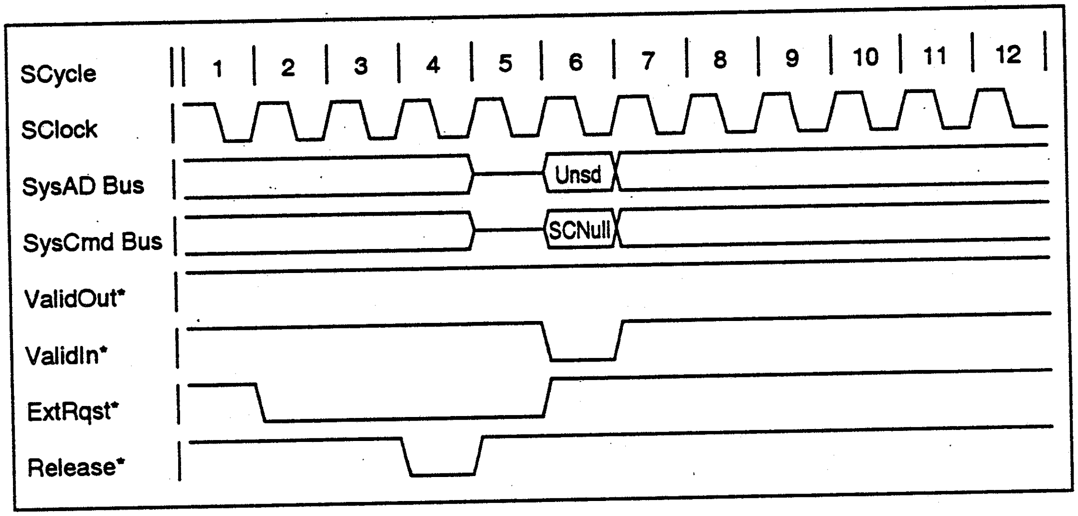


Figure 9-10 Secondary Cache Release External Null Request

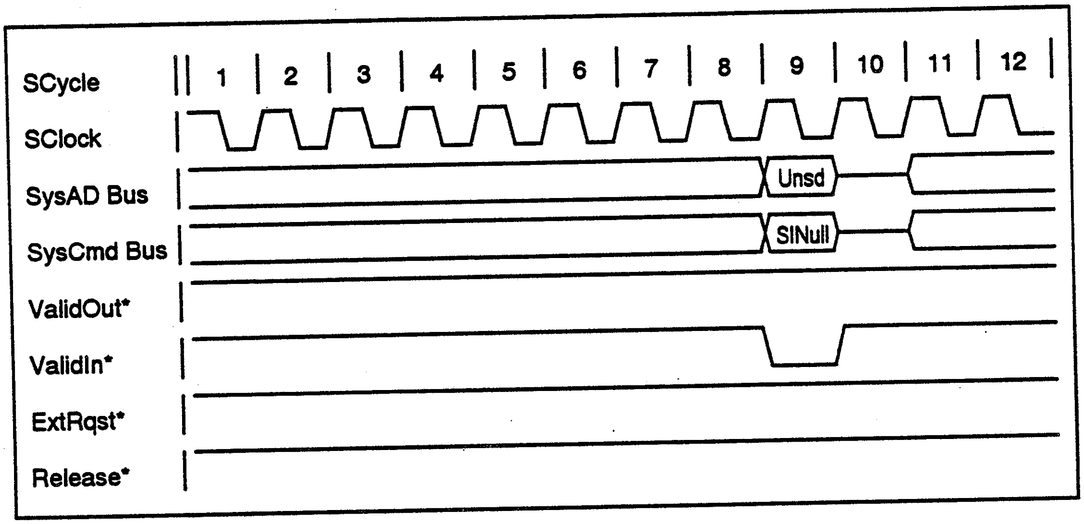


Figure 9-11 System Interface Release External Null Request

External Write Request Protocol

External write requests use a protocol identical to the processor single word write protocol except that the signal **ValidIn*** is asserted instead of the signal **ValidOut***. An external write request consists of driving a write command on the **SysCmd** bus and a write address on the **SysAD** bus and asserting **ValidIn*** for one cycle. This is followed by driving a data identifier on the **SysCmd** bus and data on the **SysAD** bus and asserting **ValidIn*** for one cycle. The data identifier associated with the data cycle must contain a coherent or noncoherent last data cycle indication. After the data cycle is issued, the write request is complete and the external agent releases the **SysCmd** and **SysAD** buses and allows the system interface to return to master state.

External write requests are only allowed to write a word of data to the processor. The behavior of the processor in response to an external write request for any data element other than a word is undefined.

An external write request with the system interface initially in master state is illustrated in Figure 9-12.

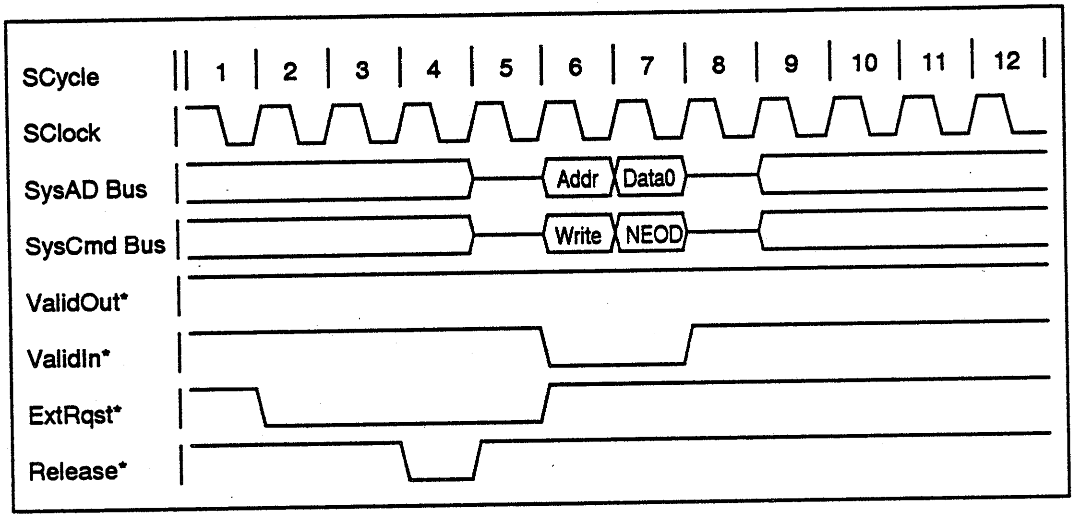


Figure 9-12 External Write Request

NOTE: The only writable resources in the R4000 are the processor interrupts.

External Invalidate and Update Request Protocol

External invalidate and update requests use a protocol identical to that for external write requests. The data element provided with an update request may be a double word, word, or partial word. The single data cycle will be unused (does not contain valid data) for an invalidate request. An external invalidate request following an uncompelled change to slave state is illustrated in Figure 9-13.

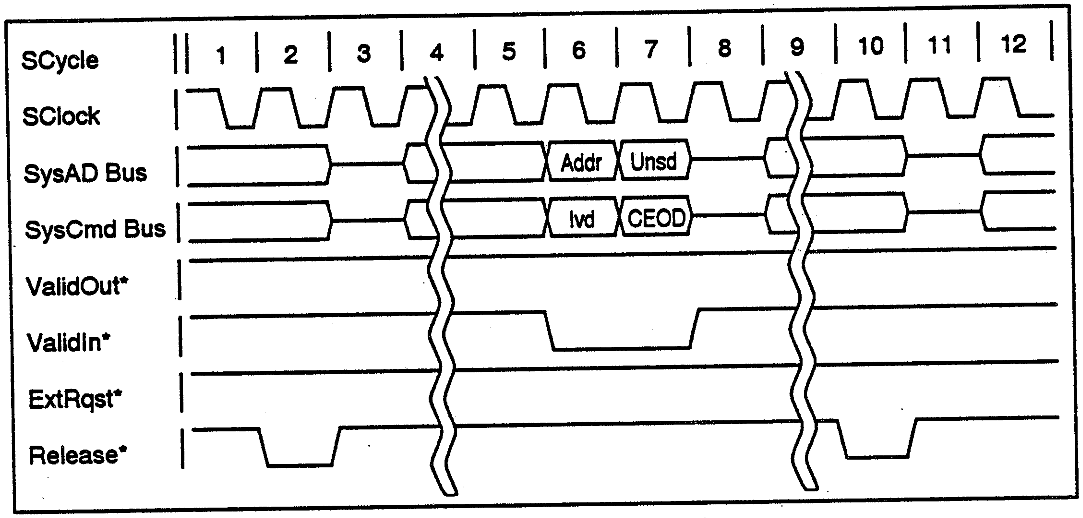


Figure 9-13 External Invalidate Request following an Uncompelled Change to Slave State

Read Response Protocol

An external agent must return data to the processor in response to a processor read request. It does this by first waiting for the processor to perform an uncompelled change to slave state and then returning the data via a single data cycle or a series of data cycles sufficient to transmit the requested data. After the last data cycle is issued, the read response is complete and the external agent will release the SysCmd and SysAD busses and allow the system interface to return to master state. Note that the processor will always perform an uncompelled change to slave state at some time after issuing a read request.

The data identifier for the data cycles must indicate that this is response data, and the data identifier associated with the last data cycle must contain a last data cycle indication. For read responses to coherent block read requests, each data identifier must include an indication of the cache state in which to load the response data. The cache state provided with each data identifier must be the same and must be either clean exclusive, dirty exclusive, shared, or dirty shared. The behavior of the processor if the cache state provided with the data identifiers is changed during the transfer of the block of data, or if the cache state provided is invalid is undefined.

The data identifier associated with a data cycle may indicate that the data transmitted during that cycle is erroneous; however, an external agent must return a block of data of the correct size regardless of erroneous data cycles. If a read response includes one or more erroneous data cycles, the processor takes a bus error.

Read response data must only be delivered to the processor when a processor read request is pending; that is, in response to a processor read request. The behavior of the processor is undefined when a read response is presented to it and there is no processor read pending. Further, if the processor issues a read-with-write-forthcoming request, a processor write request or a processor null write request must be accepted before the read response may be returned. The behavior of the processor is undefined if the read response is returned before a processor write request is accepted.

A processor word read request followed by a word read response is illustrated in Figure 9-14. A read response for a processor block read with the system interface already in slave state is illustrated in Figure 9-15.

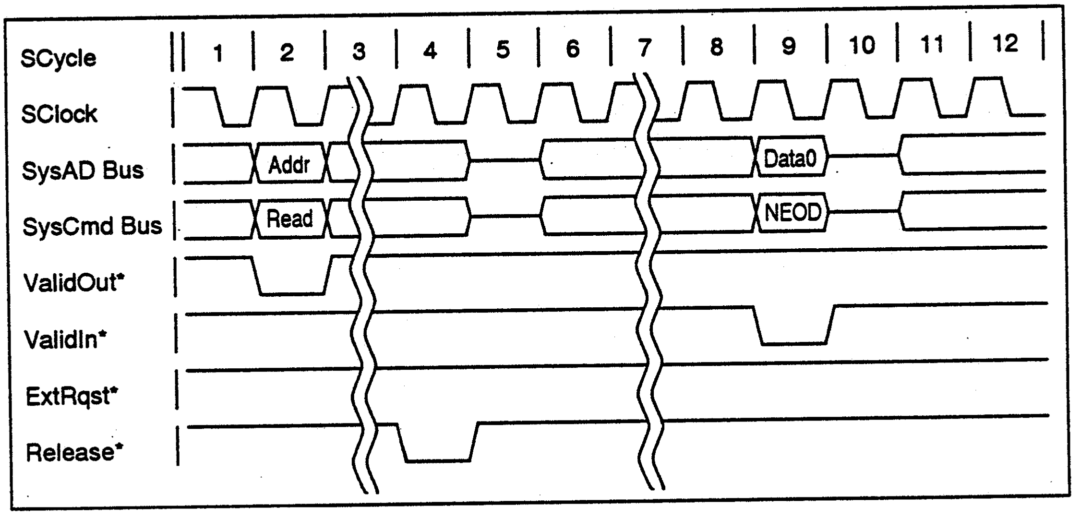


Figure 9-14 Processor Word Read Request followed by a Word Read Response

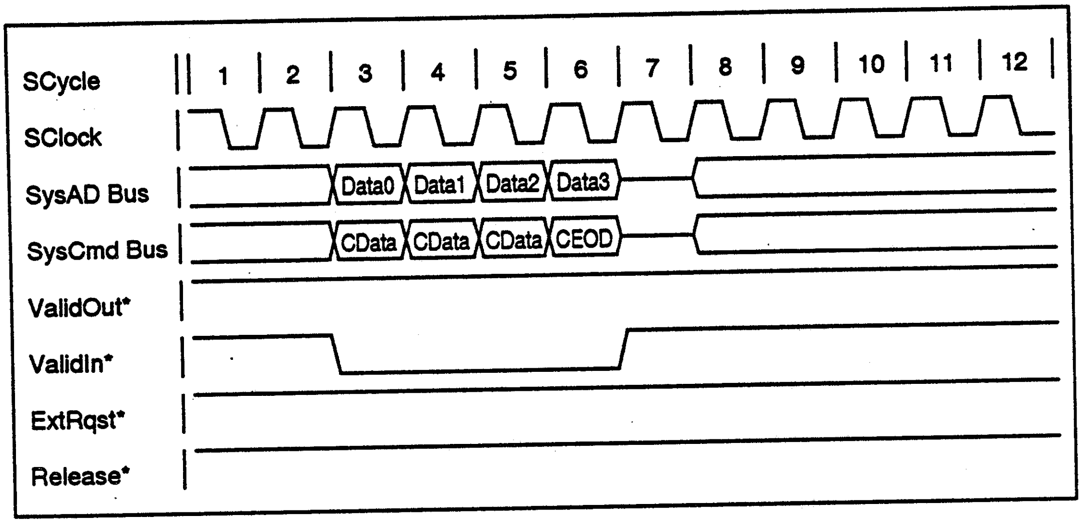


Figure 9-15 Block Read Response, System Interface already in Slave State

External Intervention Request Protocol

External intervention requests use a protocol similar to that for external read requests except that a cache line size block of data may be returned along with an indication of the cache state for the cache line. The cache state indication depends upon the state of the cache line and the value of the *data return* bit in the intervention request command.

The data return bit in the intervention request command may indicate return on dirty or return on exclusive. If the data return bit indicates return on dirty, and the cache line that is the target of the intervention request is in the state dirty exclusive or dirty shared, the contents of the cache line will be returned in response to the intervention request. If the data return bit indicates return on exclusive, and the cache line that is the target of the intervention request is in the state clean exclusive or dirty exclusive, the contents of the cache line will be returned in response to the intervention request. Otherwise, the response to the intervention request will not include the contents of the cache line, but will simply indicate the state of the cache line that is the target of the intervention request. Note that if the cache line that is the target of the intervention request is not present in the cache at all; that is, a tag comparison for the cache line at the target cache address fails, the cache line that is the target of the intervention request will be considered to be in the invalid state.

The processor will return an indication of the cache state in which a cache line was found but not its contents by driving a coherent data identifier that indicates the state of the cache line on the **SysCmd** bus and asserting **ValidOut*** for one cycle. The **SysAD** bus is unused during this data cycle. The data identifier will indicate that this is a response data cycle and will contain a last data cycle indication.

The processor will return the contents of a cache line along with an indication of the cache state in which it was found by issuing a sequence of data cycles sufficient to transmit the contents of the cache line. The data identifier transmitted with each data cycle indicates the cache state in which the cache line was found and that this is response data. The data identifier associated with the last data cycle will contain a last data cycle indication.

If the contents of a cache line are returned in response to an intervention request, it will be returned in sub-block order starting with the double word at the address supplied with the intervention request. For further details on sub-block ordering see Appendix D. Note, however, that if the intervention address targets the double word at the beginning of the block, sub-block ordering is equivalent to sequential ordering.

An external intervention request to a cache line found in the shared state with the system interface initially in master state is illustrated in Figure 9-16. An external intervention request to a cache line found in the dirty exclusive state with the system interface initially in slave state is illustrated in Figure 9-17.

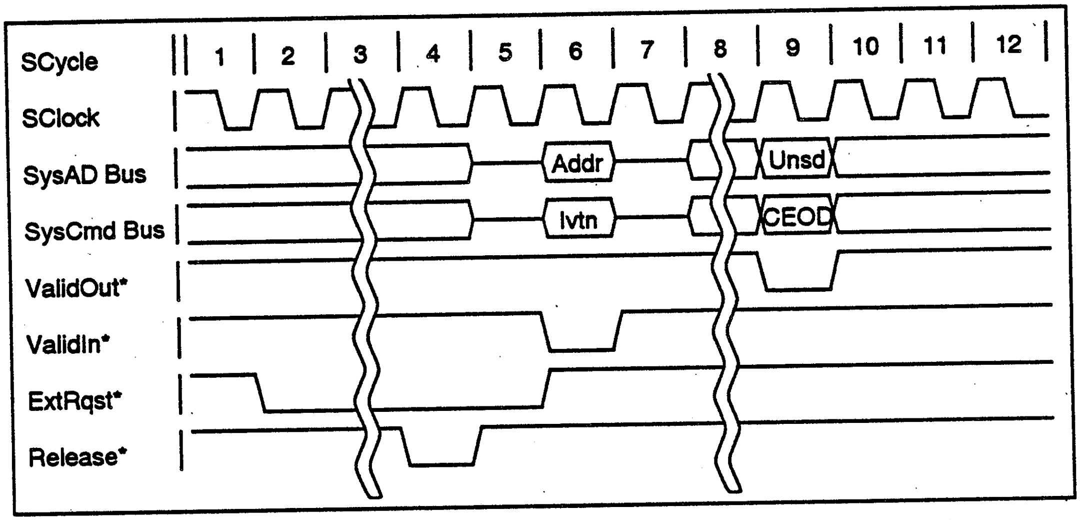


Figure 9-16 External intervention Request, Shared Line, System Interface in Master State

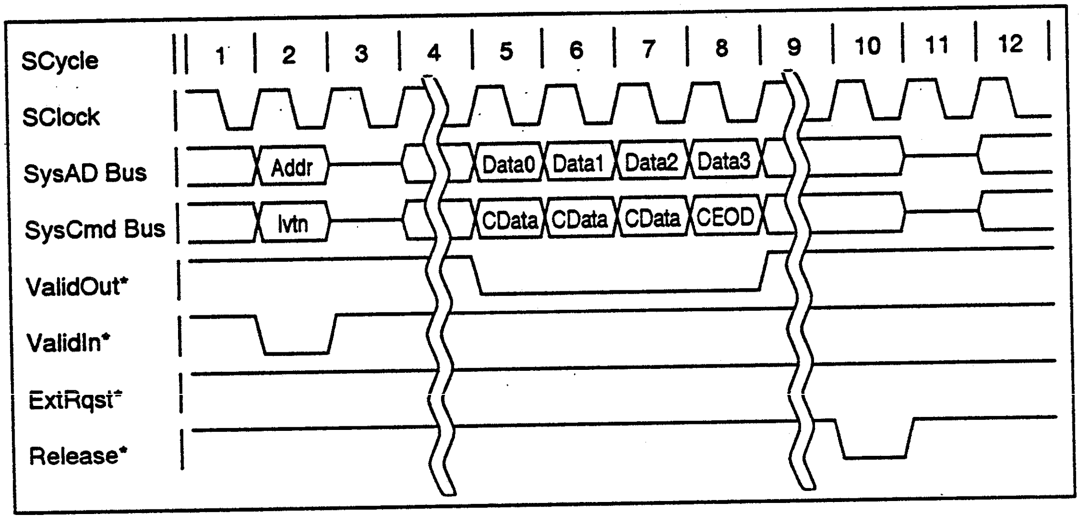


Figure 9-17 External Intervention Request, Dirty Exclusive Line, System Interface in Slave State

External Snoop Request Protocol

External snoop requests use a protocol identical to that for external read requests, except that, instead of returning data, the processor will respond to a snoop request with an indication of the current cache state for the cache line that is the target of the snoop request. The processor accomplishes this by driving a coherent data identifier on the **SysCmd** bus, and asserting **ValidOut*** for one cycle. The **SysAD** bus is unused during the snoop response. The processor will continue driving the **SysCmd** and **SysAD** busses after the snoop response is returned to transition the system interface back to master state.

Note that if the cache line that is the target of the snoop request is not present in the cache at all; that is, a tag comparison for the cache line at the target cache address fails, the cache line that is the target of the snoop request will be considered to be in the invalid state.

An external snoop request submitted with the system interface in master state is illustrated in Figure 9-18. An external snoop request submitted with the system interface in slave state is illustrated in Figure 9-19.

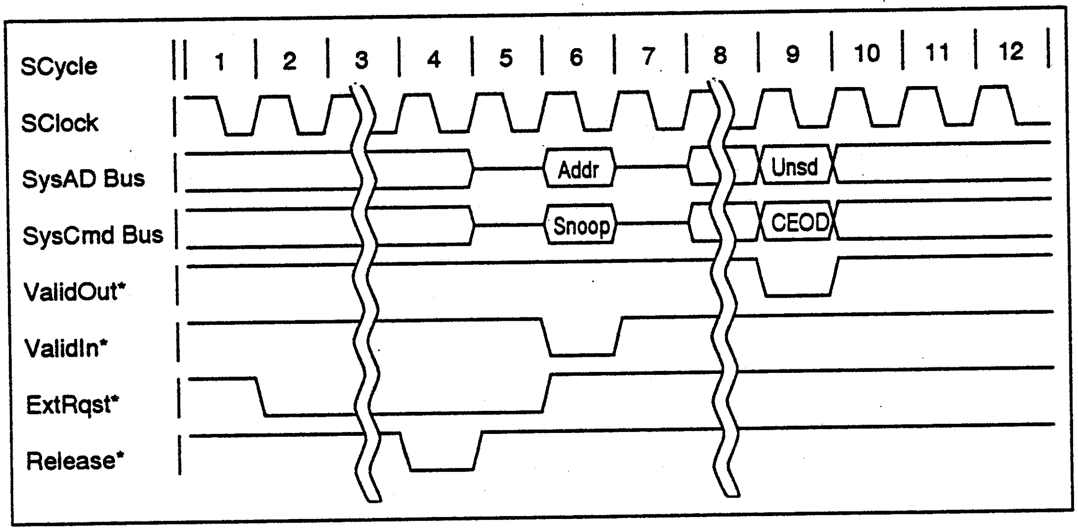


Figure 9-18 External Snoop Request, System Interface in Master State

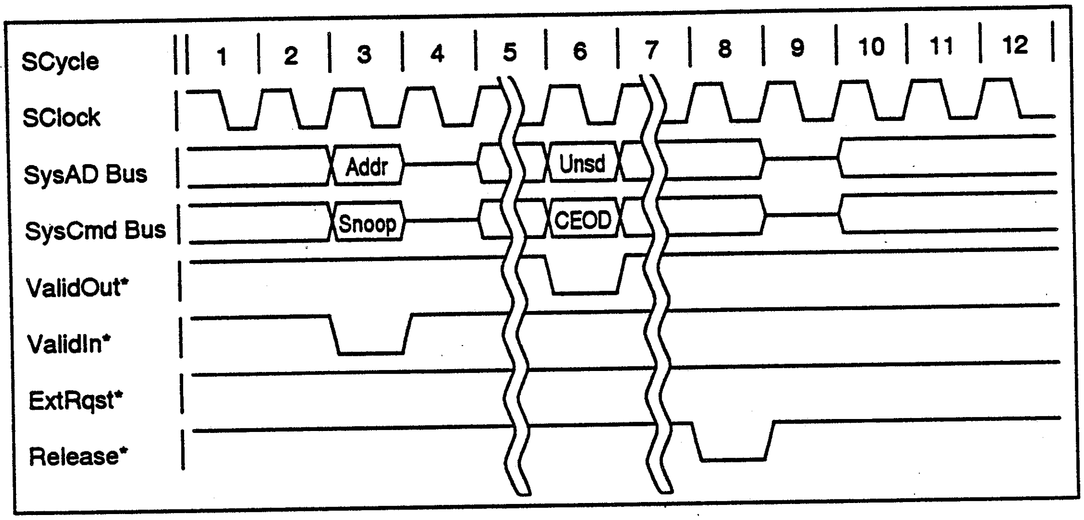


Figure 9-19 External Snoop Request, System Interface in Slave State

Processor Request and Cluster Flow Control

The signal **RdRdy*** may be used by an external agent to control the flow of a processor read, invalidate, or update request or a processor read request followed by a potential update request within a cluster. The processor samples the signal **RdRdy*** to determine if the external agent is currently capable of accepting a read, invalidate, or update request, or a read request followed by a potential update request. The signal **WrRdy*** controls the flow of a processor write request. The processor will not complete the issue of a read, invalidate, or update request or a read request followed by a potential update request until it issues an address cycle for the request for which the signal **RdRdy*** was asserted two cycles previously. The processor will not complete the issue of a write request until it issues an address cycle for the write request for which the signal **WrRdy*** was asserted two cycles previously.

Two processor write requests in which the issue of the second is delayed for the assertion of **WrRdy*** are illustrated in Figure 9-20. A processor cluster in which the issue of the read and a potential update request are delayed for the assertion of **RdRdy*** is illustrated in Figure 9-21. A processor cluster in which the issue of the write request is delayed for the assertion of **WrRdy*** is illustrated in Figure 9-22. The issue of a processor write request delayed for the assertion of **WrRdy*** and the completion of an external invalidate request is illustrated in Figure 9-23.

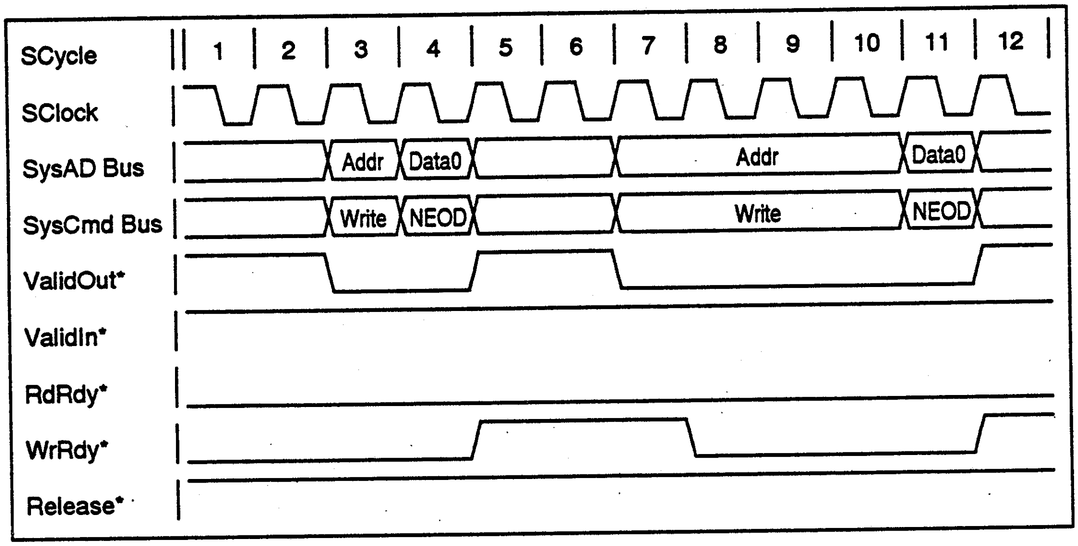


Figure 9-20 Two Processor Write Requests, Second Write Delayed for the Assertion of **WrRdy***

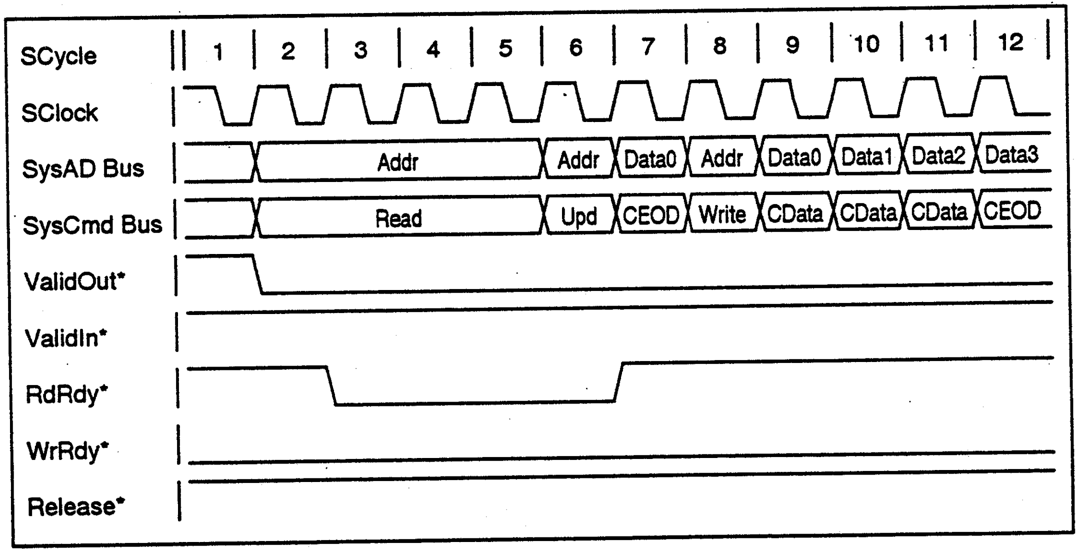


Figure 9-21 Processor Read Request Within a Cluster Delayed for the Assertion of **RdRdy***

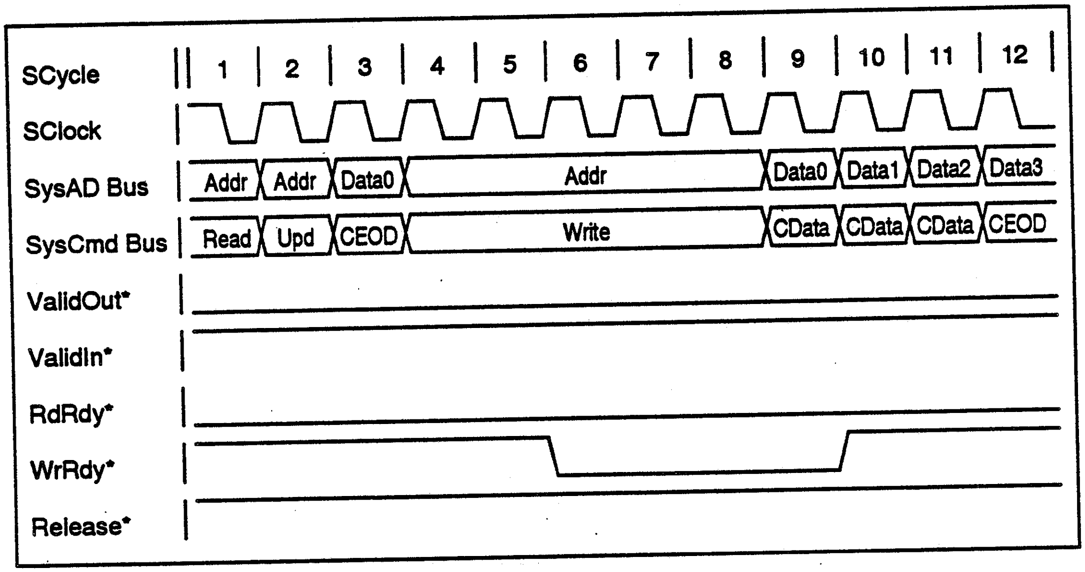


Figure 9-22 Processor Write Request Within a Cluster Delayed for the Assertion of **WrRdy***

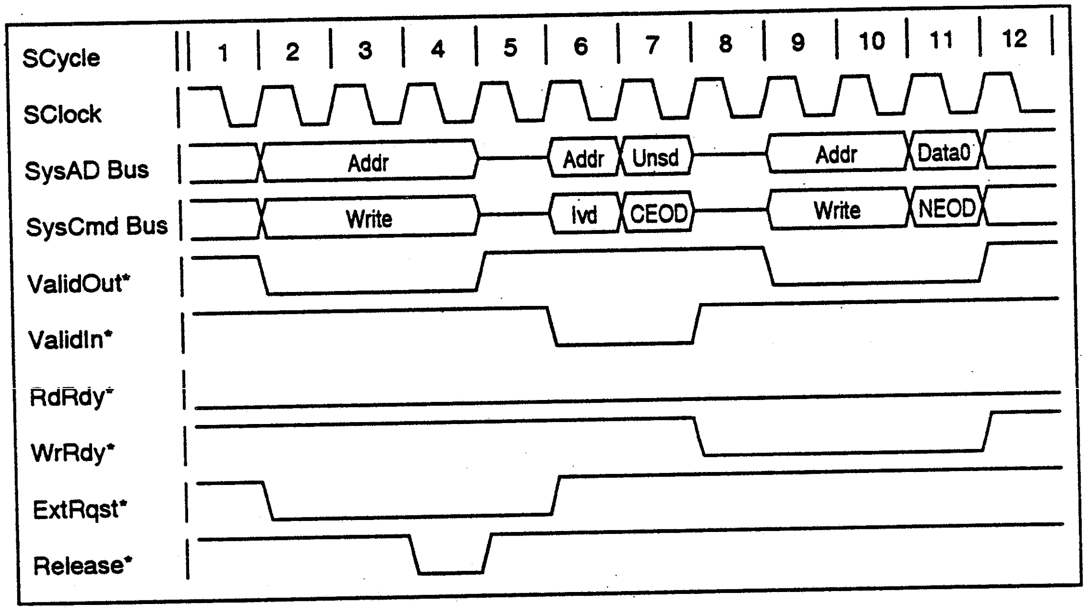


Figure 9-23 Processor Write Request Delayed for the Assertion of **WrRdy*** *and the Completion of an External Invalidate Request*

Data Rate Control

The system interface supports a maximum data rate of one double word per cycle. The maximum data rate the processor can support is directly related to the secondary cache access time, if the access time is too long, the processor will not be able to transmit and accept data at the maximum rate.

The rate at which data is delivered to the processor may be chosen by an external agent by driving data and asserting ValidIn* every n cycles instead of every cycle. The processor will only interpret cycles as valid data cycles during which **ValidIn*** is asserted and the SysCmd bus contains a data identifier. The processor will continue to accept data until the data word tagged as the last data word is received. An external agent may deliver data at any rate it chooses but must not deliver data to the processor faster than it is capable of accepting it.

Because the secondary cache is organized as a 128-bit RAM array, the processor will operate most efficiently if data is delivered to it in pairs of double words. It is most efficient to reduce the data rate by delivering a pair of double words to the processor, followed by some number of unused cycles, followed by another pair of double words. The pattern should be chosen to repeat at a rate determined by the secondary cache write cycle time. However, the processor will accept data in any pattern as long as the time between the transfer of any pair of odd-numbered double words is greater than or equal to the write cycle time of the secondary cache. Double words in the transfer pattern are numbered beginning at zero such that the odd numbered words are the second, fourth, sixth, and so on words transferred.

The maximum processor data rate for each of the possible secondary cache write cycle times and the most efficient data pattern for each data rate is illustrated in Table 9-1. In this and subsequent tables data patterns are specified using the letters "D" and "x", "D" indicates a data cycle and "x" indicates an unused cycle. A data pattern is specified as a sequence of letters, indicating a sequence of data and unused cycles that will be repeated to provide the appropriate data rate. For example, a data pattern specified by the sequence of letters "DDxx", to achieve a data rate of two words every four cycles, is a data pattern in which two data cycles are followed by two unused cycles followed by two data cycles and two unused cycles, and so on. A read response in which data is provided to the processor at a rate of two words every three cycles using the data pattern "DDx" is shown in Figure 9-24.

If data is delivered to the processor at a rate that exceeds the maximum the processor can support, based on the secondary cache write cycle time, the behavior of the processor is undefined. The secondary cache write cycle time is the sum of the parameters T_{Wr1Dly}, T_{WrSUp}, and T_{WrRc} described in the section on secondary cache write cycles. The rate at which the processor transmits data is programmable at boot time via the boot-time mode control interface. The transmit data rate may be programmed to any of the data rates and data patterns listed in Table 9-2, as long as the programmed data rate does not exceed the maximum the processor can support, based on the secondary cache access time. If a transmit data rate is programmed that exceeds the maximum the processor can support, the behavior of the processor is undefined. A processor write request for which the processor transmit data rate has been programmed to one double word every two cycles using the data pattern "DDxx" is shown in Figure 9-25.

Table 9-1 and Table 9-2 show the maximum processor and transmit data rates that can be achieved for a given set of SCache parameters, based on a PClock-to-SClock divisor of 2. To find the maximum allowable SCache write cycle time and SCache access time, multiply the maximum SCache numbers for each pattern by:

(PClock to SClock Divisor)/2

The minimum number for these parameters will always be the minimum access time supported by R4000.

Table 9-1 Maximum Processor Data Rates

SCache Write Cycle Time	Max Data Rate	Best Data Pattern
1-4 PCycles	1 Double/1 SClock Cycle	D
5-6 PCycles	2 Doubles/3 SClock Cycles	DDx
7-8 PCycles	1 Double/2 SClock Cycles	DDxx
9-10 PCycles	2 Doubles/5 SClock Cycles	DDxxx
11-12 PCycles	1 Double/3 SClock Cycles	DDxxxx

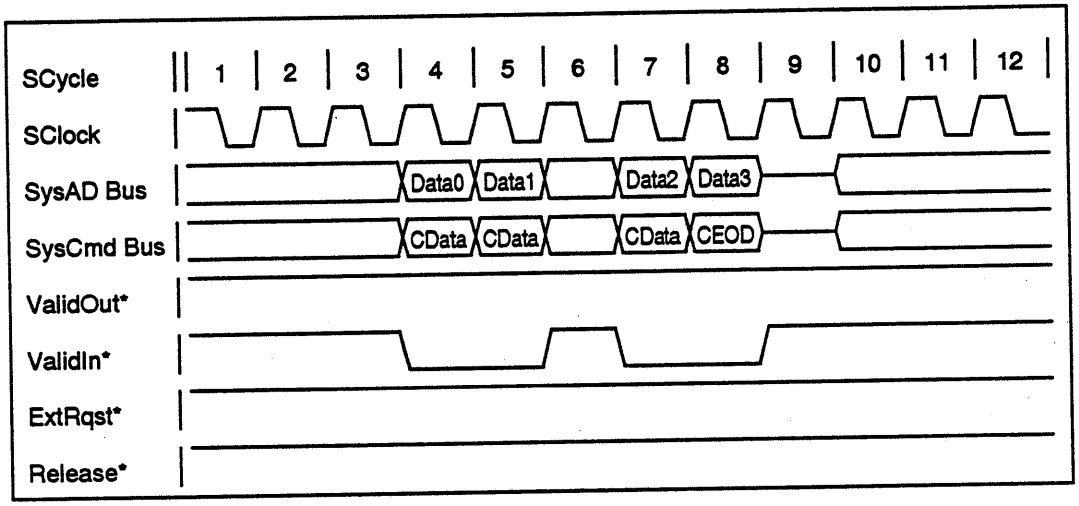


Figure 9-24 Read Response, Reduced Data Rate, System Interface in Slave State

Table 9-2 Transmit Data Rates

Data Rate	Data Pattern	Max SCache Access
1 Double/1 SClock Cycle	D	4 PCycles
2 Doubles/3 SClock Cycles	DDx	6 PCycles
1 Double/2 SClock Cycles	DDxx	8 PCycles
1 Double/2 SClock Cycles	DxDx	8 PCycles
2 Doubles/5 SClock Cycles	DDxxx	10 PCycles
1 Double/3 SClock Cycles	DDxxxx	12 PCycles
1 Double/3 SClock Cycles	DxxDxx	12 PCycles
1 Double/4 SClock Cycles	DDxxxxxx	16 PCycles
1 Double/4 SClock Cycles	DxxxDxxx	16 PCycles

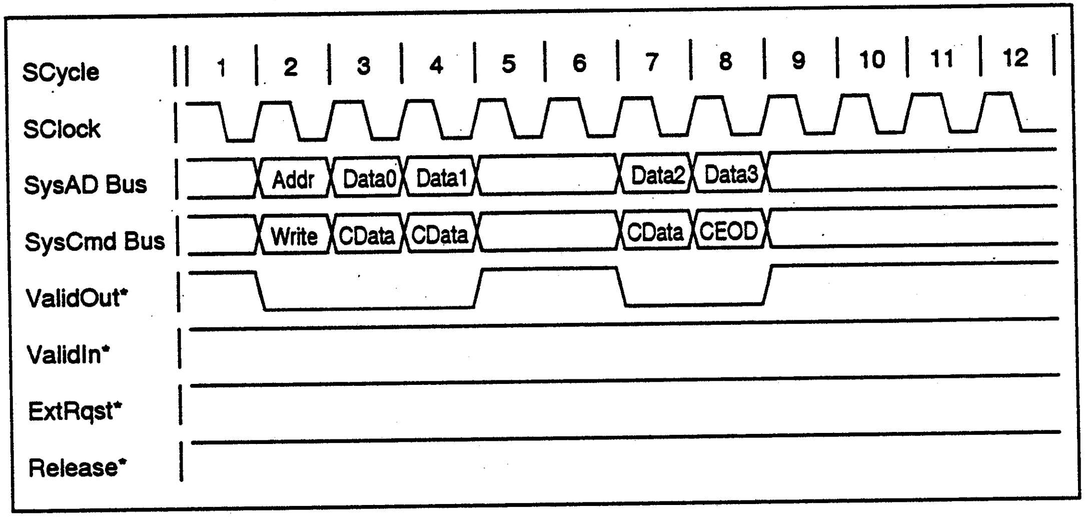


Figure 9-25 Processor Write Request, Transmit Data Rate Reduced

Multiple Drivers on the SysAD Bus

In most applications the SysAD bus will be a point to point connection from the processor to a bidirectional, registered, transceiver in an external agent. For those applications, the SysAD bus has only two possible drivers, the processor and the external agent. However, certain applications may wish to add additional drivers and receivers to the SysAD bus, and allow transmissions to take place over the SysAD bus that the processor is not involved in. To accomplish this, the external agent must coordinate the usage of the SysAD bus using the arbitration handshake signals and the external null requests.

To implement an independent transmission on the SysAD bus that does not involve the processor, the external agent must request the SysAD bus to issue an external request. If the processor is being used with a secondary cache, and after the processor releases the system interface to slave state, the external agent should issue a scache release external null request to return ownership of the secondary cache to the processor. The external agent can then allow the independent transmission to take place on the SysAD bus making sure that **ValidIn*** is not asserted while the transmission is occurring. When the transmission is complete, the external agent can issue a system interface release external null request to return the system interface to master state.

System Interface Endianness

The endianness of the system interface is programmed at boot time through the boot time mode control interface, and remains fixed until the next time the processor mode bits are read. The endianness of the system interface and the external system cannot be changed by software; the reverse endian bit can be set by software to reverse the interpretation of endianess inside the processor, but the endianess of the system interface remains unchanged.

Cycle Counts for System Interface Interactions

To facilitate system design, the processor specifies minimum and maximum cycle counts for various processor transactions and for the processor's response time to external requests. Processor requests themselves are constrained by the system interface request protocol, and the cycle counts for such requests can be determined by examining the protocol. The spacing between requests within a cluster, the waiting period for the processor to release the system interface to slave state in response to an external request, and the response time for an external request that requires a response is variable and subject to minimum and maximum cycle counts. The remainder of this section will describe and tabulate the minimum and maximum cycle counts for these system interface interactions.

The minimum and maximum number of unused cycles between the requests within a cluster is a function of processor internal activity. The minimum number of unused cycles separating requests within a cluster is zero: the requests may be adjacent. The maximum number of unused cycles separating requests within a cluster varies depending on the requests that form the cluster. The minimum and maximum number of unused cycles separating requests within a cluster is summarized in Table 9-3.

Table 9-3 Unused Cycles Separating Requests Within a Cluster

From Processor Request	To Processor Request	Minimum Unused SClock Cycles	Maximum Unused SClock Cycles
Read	Invalidate or Update	0	2
Read	Write	0	2
Invalidate or Update	Write	0	2

The number of cycles the processor may wait to release the system interface to slave state for an external request is referred to as the *release latency*. The release latency is a function of processor internal activity and processor request activity. The processor will release the system interface to accept an external request under the conditions described above. When no processor requests are in progress, internal activity, such as refilling the primary cache from the secondary cache, may cause the processor to wait some number of cycles before releasing the system interface. Release latency is defined as the number of cycles **ExtRqst*** is asserted for before the signal **Release*** is asserted. Release latency is considered in three categories:

1. Release latency when the external request signal is asserted during the cycle two cycles before the last cycle of a processor request or two cycles before the last cycle of the last request in a cluster.

2. Release latency when the external request signal is not asserted during a processor request or cluster, or asserts during the last cycle of a processor request or cluster.

3. Release latency when the processor does an uncompelled change to slave state.

The minimum and maximum release latencies for requests that fall into categories (1), (2), and (3) above are summarized in Table 9-4.

Table 9-4 Release Latency (in Pcycles) for Category (1), (2), and (3) External Requests

Category	Minimum *	Maximum *
(1)	4	6
(2)	4	24
(3)	0	TBD

*These cycle counts are preliminary and are subject to change.

The number of cycles the processor may take to respond to an external request that requires a response, that is, an external intervention request, read request, or snoop request, will be referred to as the *intervention response latency*, *external read response latency*, or *snoop response latency* respectively. The number of cycles of latency is the number of unused cycles between the address cycle of the request and the first data cycle of the response. Intervention response latency and snoop response latency are a function of processor internal activity and secondary cache access time. The minimum and maximum intervention response latency and snoop response latency, as a function of secondary cache access time, is summarized in Table 9-5. External read response latency is purely a function of processor internal activity. The minimum and maximum external read response latency is summarized in Table 9-6.

Table 9-5 Intervention Response Latency and Snoop Response Latency (in Pcycles)

Max SCache Access	Intervention response latency*		Snoop response latency*	
	Min	Max	Min	Max
1-4 PCycles	6	26	6	26
5-6 PCycles	8	28	8	28
7-8 PCycles	10	30	10	30
9-10 PCycles	12	32	12	32
11-12 PCycles	14	34	14	34

*These cycle counts are preliminary, and are subject to change.

Table 9-6 External Read Response Latency (in Pcycles)

	Min*	Max*
External Read Response Latency	4	4

*These cycle counts are preliminary, and are subject to change.

System Interface Syntax

System interface commands specify the precise nature and attributes of any system interface request during the address cycle for the request. System interface data identifiers specify the attributes of a data element transmitted during a system interface data cycle. The followings sections describe the syntax, that is, the bitwise encoding, of system interface commands and data identifiers.

For system interface commands and data identifiers associated with external requests, reserved bits and reserved fields in the command or data identifier should be deasserted—that is set to one (1) or all ones respectively. For system interface commands and data identifiers associated with processor requests, reserved bits and reserved fields in the command and data identifier are undefined.

System Interface Command and Data Identifier Syntax

System interface commands and data identifiers are encoded in nine bits, and are transmitted from the processor to an external agent or from an external agent to the processor on the SysCmd bus during address and data cycles. Bit eight (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 defines the encoding of the **SysCmd** bus for system interface commands. A common encoding is used for all system interface commands. **SysCmd(8)** must be set to 0 for all system interface commands.

For all system interface commands **SysCmd(7:5)** specify the system interface request type which may be read, write, null, invalidate, update, intervention, or snoop. The encoding of **SysCmd(7:5)** for system interface commands is illustrated in Table 9-7.

Table 9-7 Encoding of SysCmd(7:5) for System Interface Commands

SysCmd(7:5)	Command
0	Read Request
1	Read-With-Write-Forthcoming Request
2	Write Request
3	Null Request
4	Invalidate Request
5	Update Request
6	Intervention Request
7	Snoop Request

For read requests, the remainder of the **SysCmd** bus specifies the attributes of the read. **SysCmd(4:3)** encode block, coherency, and exclusivity attributes for the read. A read request with a write request forthcoming cannot be a double word, word, or partial word read. For coherent and noncoherent block reads **SysCmd(2)** specifies whether the address of the cache line being replaced by this read request is being retained in the link address register and **SysCmd(1:0)** encodes the block size for the read. For double word, word, or partial word reads, **SysCmd(2:0)** encodes the size of the read data in bytes. The encodings of **SysCmd(4:3)** for read commands are shown below in Table 9-8. The encodings of **SysCmd(2:0)** for coherent and noncoherent block reads and double word, word, or partial word reads is shown in Table 9-9 and Table 9-10, respectively.

Table 9-8 Encoding of SysCmd(4:3) for Read Requests

SysCmd(4:3)	Read attributes
0	Coherent block read
1	Coherent block read, exclusivity requested
2	Noncoherent block read
3	Double word, single word, or partial word read

Table 9-9 Encoding of SysCmd(2:0) for Coherent and Noncoherent Block Read Requests

SysCmd(2)	Link address retained indication
0	Link address not retained
1	Link address retained
SysCmd(1:0)	Read block size
0	Four words
1	Eight words
2	Sixteen words
3	Thirty-two words

Table 9-10 Encoding of SysCmd(2:0) for Double Word, Word, or Partial Word Read Requests

SysCmd(2:0)	Read data size
0	One byte valid (Byte)
1	Two bytes valid (Halfword)
2	Three bytes valid (Tribyte)
3	Four bytes valid (Word)
4	Five bytes valid (Quintibyte)
5	Six bytes valid (Sextibyte)
6	Seven bytes valid (Septibyte)
7	Eight bytes valid (Double Word)

For write requests, the remainder of the SysCmd bus specifies the attributes of the write. SysCmd(4:3) encode block attributes for the write. For block writes SysCmd(2) specifies whether the cache line associated with the write request will be replaced or retained after the write is completed and SysCmd(1:0) encode the block size for the write. For double word, word, or partial word writes SysCmd(2:0) encode the size of the write data in bytes. The encodings of SysCmd(4:3) for write commands are shown below in Table 9-11. The encodings of SysCmd(2:0) for block writes or double word, word, or partial word writes are shown in Table 9-12 and Table 9-13, respectively.

Table 9-11 Encoding of SysCmd(4:3) for Write Requests

SysCmd(4:3)	Write attributes
0	Reserved
1	Reserved
2	Block write
3	Doubleword, word, or partial word write.

Table 9-12 Encoding of SysCmd(2:0) for Block Write Requests

SysCmd(2)	Cache line replacement attributes
0	Cache line replaced
1	Cache line retained
SysCmd(1:0)	**Write block size**
0	Four words
1	Eight words
2	Sixteen words
3	Thirty-two words

Table 9-13 Encoding of SysCmd(2:0) for Doubleword, Word, or Partial Word Write Requests

SysCmd(2:0)	Write data size
0	One byte valid (Byte)
1	Two bytes valid (Halfword)
2	Three bytes valid (Tribyte)
3	Four bytes valid (Word)
4	Five bytes valid (Quintibyte)
5	Six bytes valid (Sextibyte)
6	Seven bytes valid (Septibyte)
7	Eight bytes valid (Doubleword)

Processor null write requests, system interface release external null requests, and scache release external null requests all use the null request command. For processor null requests, SysCmd(4:3) specifies that this is a null write request. For external null requests, SysCmd(4:3) specifies whether this is a system interface release null request or a scache release null request.The encodings of SysCmd(4:3) for processor null requests are shown in Table 9-14. The encodings of SysCmd(4:3) for external null requests are shown in Table 9-15.

Table 9-14 Encoding of SysCmd(4:3) for Processor Null Requests

SysCmd(4:3)	Null attributes
0	Null write
1	Reserved
2	Reserved
3	Reserved

Table 9-15 Encoding of SysCmd(4:3) for External Null Requests

SysCmd(4:3)	Null attributes
0	System interface release
1	Scache release
2	Reserved
3	Reserved

For invalidate and update requests SysCmd(4) is used by external requests to indicate that the external request is in conflict with an unacknowledged processor update request, canceling the update. SysCmd(4) is reserved for processor update requests. SysCmd(3) is used by processor requests to specify whether the update is potential or compulsory. SysCmd(3) is reserved for processor invalidate requests. SysCmd(3) is used by external update requests to indicate whether the update request will change the state of the updated cache line to shared, or leave the state of the updated cache line unchanged. SysCmd(2:0) specifies the size of the data element in bytes for update requests. The encodings of SysCmd(4:0) for processor invalidate and update requests is shown in Table 9-16. The encodings of SysCmd(4:0) for external invalidate and update requests are shown in Table 9-17. SysCmd(4:0) is reserved for processor invalidate requests.

Table 9-16 Encoding of SysCmd(4:0) for Processor Update Requests

SysCmd(4)	**Reserved**
SysCmd(3)	**Update type**
0	Compulsory
1	Potential
SysCmd(2:0)	**Update data size**
0	One byte valid (Byte)
1	Two bytes valid (Halfword)
2	Three bytes valid (Tribyte)
3	Four bytes valid (Word)
4	Five bytes valid (Quintibyte)
5	Six bytes valid (Sextibyte)
6	Seven bytes valid (Septibyte)
7	Eight bytes valid (Doubleword)

Table 9-17 Encoding of SysCmd(4:0) for External Update Requests

SysCmd(4)	**Processor unacknowledged update cancellation**
0	Update cancelled
1	No cancellation
SysCmd(3)	**Update cache state change attributes**
0	Cache state changed to Shared
1	No change to cache state
SysCmd(2:0)	**Update data size**
0	One byte valid. (Byte)
1	Two bytes valid (Halfword)
2	Three bytes valid (Tribyte)
3	Four bytes valid (Word)
4	Five bytes valid (Quintibyte)
5	Six bytes valid (Sextibyte)
6	Seven bytes valid (Septibyte)
7	Eight bytes valid (Doubleword)

NOTE: SysCmd(3:0) is reserved for External Invalidate Requests

For intervention and snoop requests SysCmd(4) is used to indicate that this external request is in conflict with an unacknowledged processor update request, canceling the update. The processor never issues an intervention or snoop request. SysCmd(3) is the data response on dirty bit for intervention requests and is reserved for snoop requests. If the data response on dirty bit is asserted, the processor returns the contents of the cache line in response to an intervention request if the line is found in state dirty exclusive or dirty shared. If the data response on dirty bit is deasserted, the processor returns the contents of the cache line in response to an intervention request if the line is found in state clean exclusive or dirty exclusive. For both snoop and intervention requests, SysCmd(2:0) specify a cache state change function that is applied to the cache line atomically with respect to the intervention or snoop response.

The encodings ofSysCmd(4:0) for intervention requests are shown in Table 9-19; the encodings SysCmd(4:0) for snoop requests are shown in Table 9-19.

Table 9-18 Encodings of SysCmd(4:0) for Intervention Requests

SysCmd(4)	Processor unacknowledged update cancellation
0	Update cancelled
1	No cancellation
SysCmd(3)	Data response on dirty bit
0	Return cache line data if in state dirty exclusive or dirty shared
1	Return cache line data if in state clean exclusive or dirty exclusive
SysCmd(2:0)	Cache state change function
0	No change to cache state
1	If cache state is clean exclusive, change to shared, otherwise no change to cache state
2	If cache state is clean exclusive or shared, change to invalid, otherwise no change to cache state
3	If cache state is clean exclusive, change to shared or if cache state is dirty exclusive, change to dirty shared, otherwise no change to cache state
4	If cache state is clean exclusive, dirty exclusive, or dirty shared, change to shared, otherwise no change to cache state
5	Change to invalid regardless of current cache state
6	Reserved
7	Reserved

Table 9-19 Encodings of SysCmd(4:0) for Snoop Requests

SysCmd(4)	Processor unacknowledged update cancellation
0	Update cancelled
1	No cancellation
SysCmd(3)	Reserved
SysCmd(2:0)	Cache state change function
0	No change to cache state
1	If cache state is clean exclusive, change to shared, otherwise no change to cache state
2	If cache state is clean exclusive or shared, change to invalid, otherwise no change to cache state
3	If cache state is clean exclusive, change to shared or if cache state is dirty exclusive, change to dirty shared, otherwise no change to cache state
4	If cache state is clean exclusive, dirty exclusive, or dirty shared, change to shared, otherwise no change to cache state
5	Change to invalid regardless of current cache state
6	Reserved
7	Reserved

System Interface Data Identifier Syntax

This section defines the encoding of the SysCmd bus for system interface data identifiers. A common encoding is used for all system interface data identifiers. **SysCmd(8)** must be set to 1 for all system interface data identifiers. System interface data identifiers have two formats, one for coherent data and another for noncoherent data:

- Data associated with processor block write requests and processor double word, word, or partial word write requests is noncoherent.
- Data associated with processor update requests is noncoherent.
- Data returned in response to a processor coherent block read request is coherent.

- Data returned in response to a processor noncoherent block read request or a processor double word, word, or partial word read request is noncoherent.
- Data associated with external update requests is noncoherent.
- Data associated with external write requests is noncoherent.
- Data returned in response to an external read request is noncoherent.
- Data returned in response to an external intervention request is coherent.

For coherent and noncoherent data identifiers, both processor and external, SysCmd(7) marks the last data element and SysCmd(6) indicates whether or not the data is response data. Response data is data returned in response to a read request or an intervention request. SysCmd(5) is the good data bit and indicates whether or not the data element is error free. Erroneous data contains an uncorrectable error. Erroneous data returned to the processor will cause a processor bus error. The processor will deliver data with the good data bit deasserted when a primary parity error is detected for a transmitted data item. A secondary cache data ECC error can be detected by comparing the values transmitted on the SysAD and SysADC. For external data identifiers, both coherent and noncoherent, SysCmd(4) indicates to the processor whether to check the data and check bits for this data element, and SysCmd(3) is reserved. For processor data identifiers, both coherent and noncoherent, SysCmd(4:3) are reserved.

For coherent data identifiers SysCmd(2:0) indicate a cache state for the data. This indication provides the cache state with which to load the cache line for responses to processor coherent read requests. It also indicates the cache state in which the line was found for data associated with the response to an external intervention request or for the data cycle issued in response to an external snoop request. For noncoherent data identifiers SysCmd(2:0) is reserved.

The encodings of SysCmd(7:3) for processor data identifiers are illustrated in Table 9-20. The encodings of SysCmd(7:3) for external data identifiers are illustrated in Table 9-21. The encodings of SysCmd(2:0) for coherent data identifiers are illustrated in Table 9-22.

Table 9-20 Encoding of SysCmd(7:3) for Processor 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 9-21 Encoding 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)	Data checking enable
0	Check the data and check bits
1	Don't check the data and check bits
SysCmd(3)	Reserved

Table 9-22 Encoding of SysCmd(2:0) for Coherent Data Identifiers

SysCmd(2:0)	Cache state
0	Invalid
1	Reserved
2	Reserved
3	Reserved
4	Clean Exclusive
5	Dirty Exclusive
6	Shared
7	Dirty Shared

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. Addresses associated with double word, word, or partial word transactions, i.e. double word, word, or partial word read and write requests and update requests, are aligned for the size of the data element. Specifically; for double word requests, the low-order three bits of the address are zero; for word requests, the low-order two bits of the address are zero; and for half-word requests, the low-order bit of the address is zero. For byte, tri-byte, quinti-byte, sexti-byte and septi-byte requests the address provided is a byte address.

Addresses associated with block requests are aligned to double-word boundaries; that is, the low-order three bits of the address are zero. The order in which data is returned in response to a processor block read request can be programmed via the boot-time mode control interface to sequential ordering or sub-block ordering. If sequential ordering is enabled, the processor always delivers the address of the double word at the beginning of the block on a block read request. An external agent must return the block of data sequentially starting at the beginning of the block. If sub-block ordering is enabled, the processor delivers the address of the double word within the block that it wants returned first. An external agent must return the block of data using sub-block ordering starting with the addressed double word. For further details on sub-block ordering see Appendix D. Only an R4000 in the R4000SC and R4000MC configuration with a secondary cache may be programmed to use sequential ordering.

For block write requests, the processor always delivers the double word address of the double word at the beginning of the block, and delivers data beginning with the double word at the beginning of the block and progressing sequentially through the double words that form the block.

During data cycles, the driven byte lines depend upon the position of the data with respect to the aligned double word containing the data (this may be a byte, halfword, tri-byte, word, quinti-byte, sexti-byte, septi-byte, or a double word.). For example, on a byte request whose address modulo 8 is 0, SysAD 7...0 get driven during the data cycles. Please refer to Figure 2.2.

Processor Internal Address Map

External reads and writes to the processor are provided to access processor internal resources that may be of interest to an external agent. However, the R4000 does not contain any resources that are readable with an external read request. The R4000 will return a bus error response to any external read request. The only writable resource in this version of the R4000 are the processor interrupts.

The processor decodes bits 6:4 of the address associated with an external read or write request to determine which processor internal resource is the target of the request. The only processor internal resource available for access by an external request is the interrupt resource, and it is only accessible via an external write request. The interrupt resource is accessed via an external write request with an address of 000 on bits 6:4 of the SysAD bus. See the section on interrupts for further details on external writes to the interrupt resource.

Clock/Control Interface

10

This chapter describes the clocks used in the R4000 and the processor status reporting mechanism. The topics covered here include:

- Basic System Clocks
- Interfacing to a Phase-Locked System
- Interfacing to a System without Phase Locking
- Processor Status Outputs

Basic System Clocks

Each clock in the R4000 is explained below.

MasterClock

The processor bases all internal and external clocking on the single clock input MasterClock. The processor generates the clock output MasterOut at the same frequency as MasterClock and aligns MasterOut with MasterClock, if SyncIn is shorted to SyncOut. MasterOut is provided for use in clocking external logic that must cycle at MasterClock frequency, such as the reset logic, and the processor aligns MasterOut with SyncOut.

SyncIn/SyncOut

The processor generates the clock output SyncOut at the same frequency as MasterClock and aligns SyncIn with MasterClock. SyncOut must be connected to the clock input SyncIn so that the processor can compensate for output driver delays and input buffer delays in aligning SyncIn with MasterClock.

PClock

The processor generates the internal clock PClock at twice the frequency of MasterClock and precisely aligns every other rising edge of PClock with the rising edge of MasterClock. PClock is used by all internal registers and latches.

SClock

The processor divides PClock by 2, 3, or 4 (programmed via the initialization control interface) to generate the internal clock SClock. SClock is used by the processor to sample data at the system interface and to clock data into the processor's system interface output registers. The rising and falling edges of SClock are aligned with the rising edges of PClock.

TClock

The processor generates TClock at the same frequency as SClock. TClock is a transmit clock that can be used by an external agent to clock its output registers and as the global system clock for the logic that makes up the external agent. TClock is identical to SClock, and the edges of TClock are precisely aligned with the edges of SClock. TClock is used by external agent circuitry. TClock is aligned with MasterClock if SynIn is shorted to SyncOut.

Figure 10-1 shows the clocks for a PClock-to-SClock division of two. Figure 10-2 shows the clocks for a PClock-to-SClock division of four.

RClock

The processor generates RClock at the same frequency as SClock. RClock is a receive clock that can be used by an external agent to clock its input registers. RClock is skewed with respect to TClock and SClock so that it leads TClock by 25% of the SClock cycle time. RClock is used by external agent circuitry.

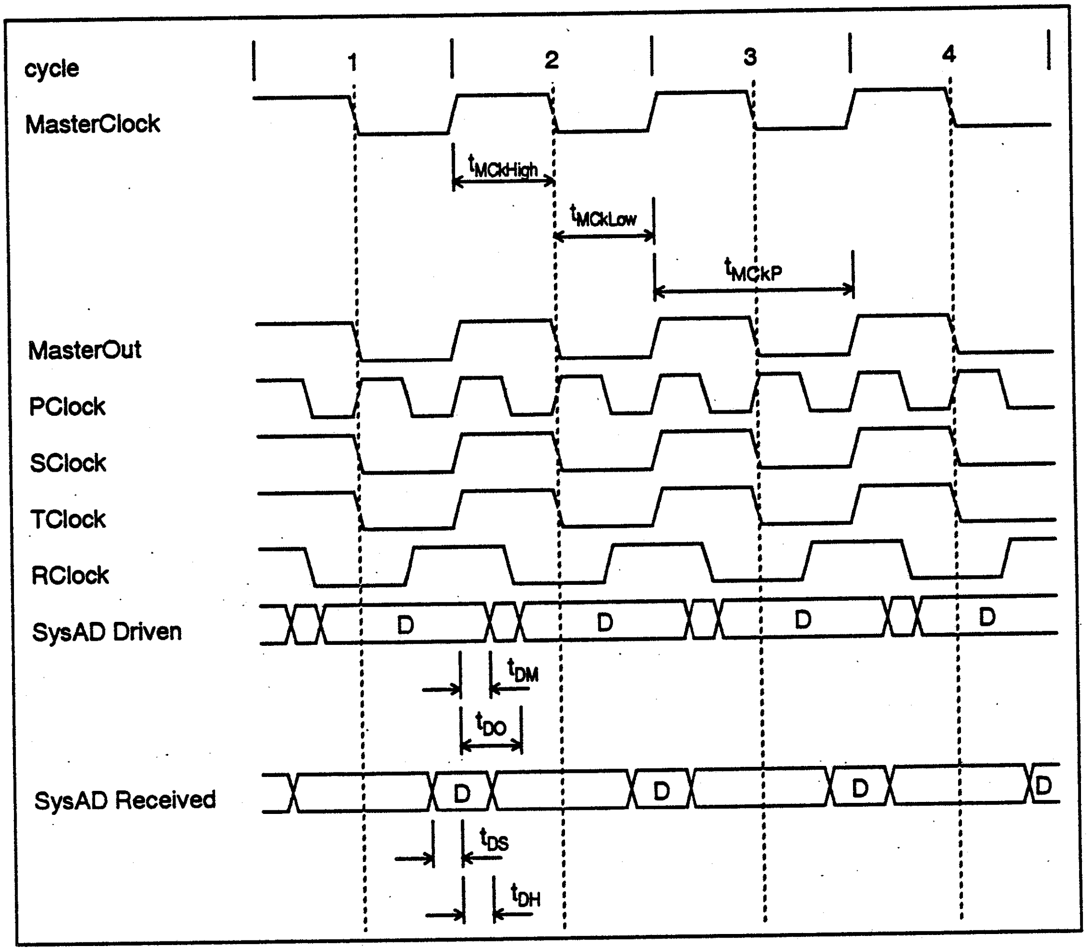


Figure 10-1 Processor Clocks, PClock to SClock Divisor of 2

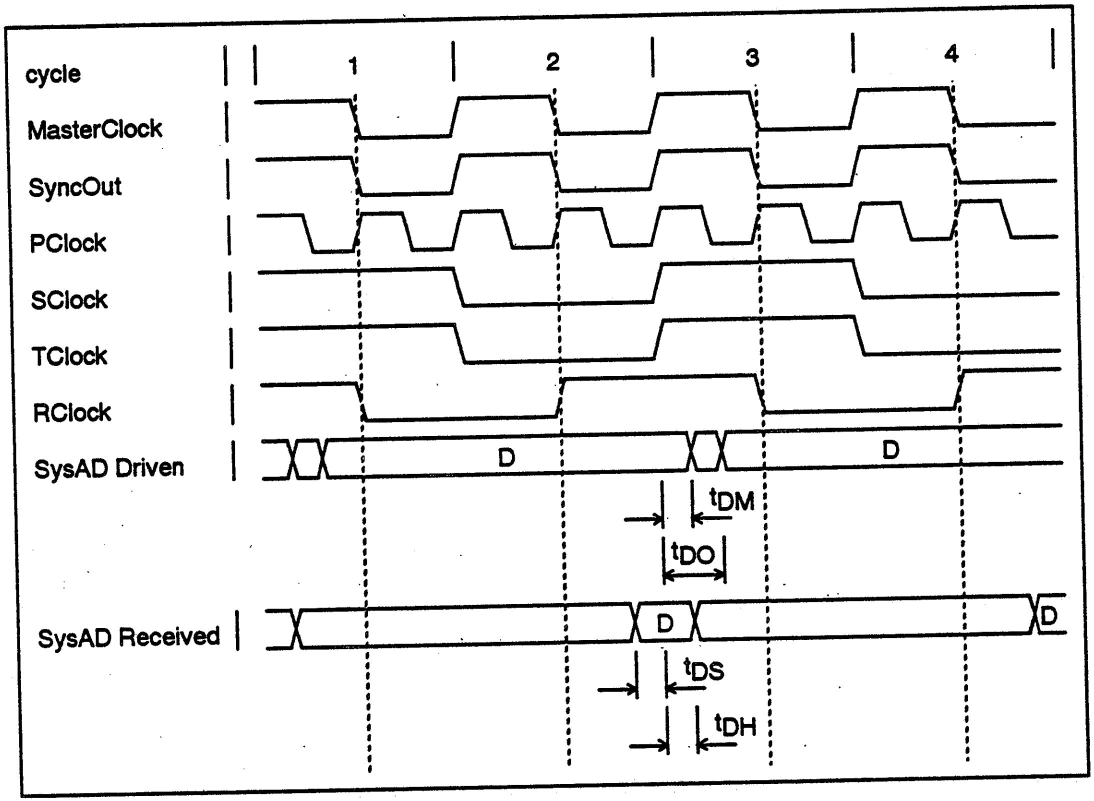


Figure 10-2 Processor Clocks, PClock to SClock Divisor of 4

System Timing Parameters

Data provided to the processor must be stable a minimum of t_{DS} nanoseconds (ns) before the rising edge of SClock and held valid for a minimum of t_{DH} ns after the rising edge of SClock. This setup and hold time is required for data to propagate through the processor's input buffers and meet the setup and hold time requirements of the processor's input latches.

Data provided by the processor becomes stable a minimum of t_{DM} ns after the rising edge of SClock and a maximum of t_{DO} ns after the rising edge of SClock. This drive-off time is the sum of the maximum delay through the processor's output drivers and the maximum clock to Q delay of the processor's output registers.

Certain processor inputs (specifically VCCOk, ColdReset*, and Reset*) are sampled based on MasterClock, while certain processor outputs (specifically Status(7:0)) are driven out based on MasterClock.

The same setup, hold, and drive-off parameters, t_{DS}, t_{DH}, t_{DM}, and t_{DO}, apply to these inputs and outputs, but they are with respect to MasterClock instead of SClock.

The values of t_{DS}, t_{DH}, t_{DM}, and t_{DO} for the R4000 processor are tabulated in *AC Characteristics.*

The alignment of SyncOut, PClock, SClock, TClock, and RClock is accomplished by the processor with internal Phase Locked Loop (PLL) circuits that generate aligned clocks based on SyncOut/SyncIn. PLL circuits by their nature are only capable of generating aligned clocks for MasterClock frequencies within a limited range. Minimum and maximum frequencies for MasterClock for various speed ratings of the R4000 processor are tabulated in *AC Characteristics.*

Clocks generated using PLL circuits contain some inherent inaccuracy, or *jitter*, in their alignment with respect to the MasterClock. That is, a clock aligned with MasterClock by the processor's PLL circuits may lead or trail MasterClock by an amount as large as the related maximum jitter. Maximum jitter for the clocks generated by various speed ratings of the R4000 processor is tabulated in *AC Characteristics.*

Clock Interfacing to a Phase-Locked System

When the processor is used in a phase-locked system, the components of the external agent must phase lock their operation to a common MasterClock. In such a system, the delivery of data and the sampling of data has common characteristics for all components, even if the components have different delay values. The *transmission time* (the amount of time a signal has to propagate along the trace from one component to another) between any two components A and B of a phase locked system can be calculated from the following equation:

$$\begin{aligned}\text{Transmission Time} = \quad & (\text{SClock period}) - (t_{DO} \text{ for A}) - (t_{DS} \text{ for B}) \\ & - (\text{Clock Jitter for A Max}) - (\text{Clock Jitter for B Max})\end{aligned}$$

A block-level diagram of a phase-locked system employing the R4000 processor is shown in Figure 10-3.

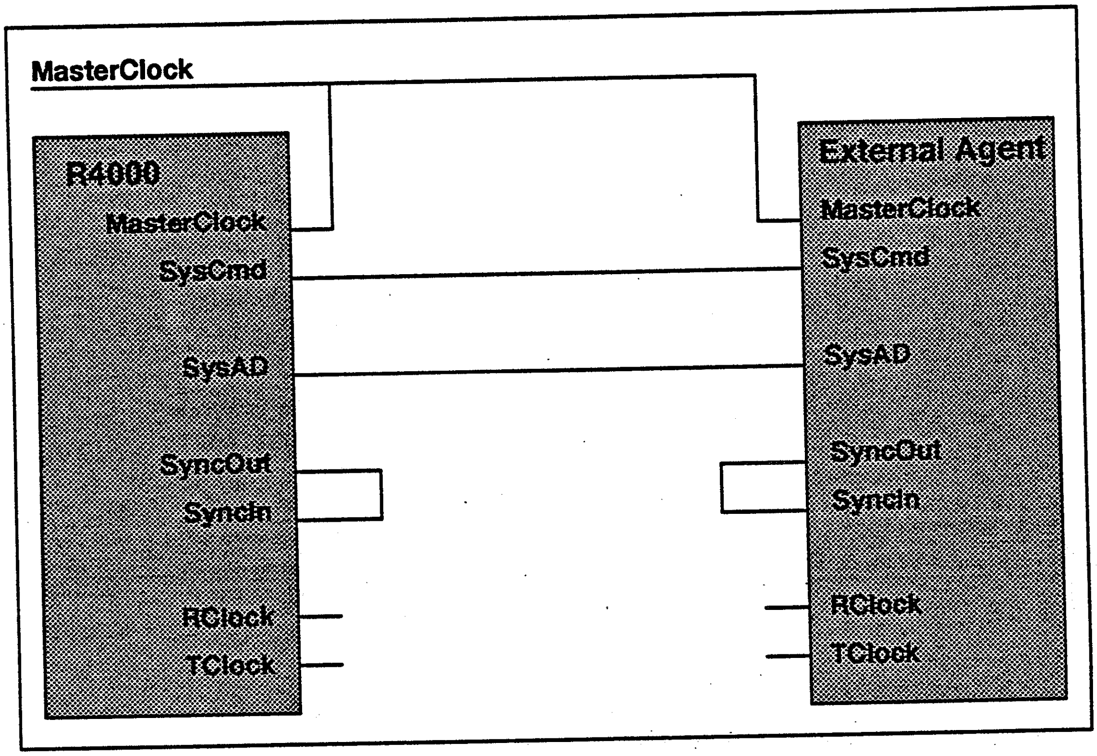


Figure 10-3 Phase-Locked System Employing the R4000 Processor

Clock Interfacing to a System Without Phase-Lock

When the processor is used in a system in which the external agent cannot phase lock to a common MasterClock, the output clocks RClock and TClock may be used to clock the remainder of the system. Two clocking methodologies are shown below: one for interfacing to gate-array devices, and one for interfacing to discrete CMOS logic devices.

Interface to Gate-Array System

When interfacing to a gate-array system, both RClock and TClock are used for clocking within the gate-arrays. The gate array buffers RClock internally and uses the buffered version to clock registers that sample processor outputs. These sample registers should be immediately followed by staging registers clocked by an internally buffered version of TClock. The buffered version of TClock should be used as the global system clock for the logic inside the gate array and as the clock for all registers that drive processor inputs.

The use of staging registers places a constraint on the sum of the clock-to-Q delay of the sample registers and the setup time of the synchronizing registers inside the gate arrays:

Clock-QDelay +
Setup of Synch Register 0.25 (RClock period)
- (Maximum Clock Jitter for RClock)
- (Maximum Delay Mismatch for Internal Clock Buffers
on RClock and TClock)

The transmission time for a signal from the processor to an external agent composed of gate arrays in a system without phase lock can be calculated from the following equation:

Transmission Time = (75% of TClock period) - (t_{DO} for R4000)
+ (Minimum External Clock Buffer Delay)
- (External Sample Register Setup Time)
- (Maximum Clock Jitter for R4000 Internal Clocks)
- (Maximum Clock Jitter for RClock)

The transmission time for a signal from an external agent composed of gate arrays to the processor in a system without phase lock can be calculated from the following equation:

Transmission Time = (TClock period) - (t_{DS} for R4000)
- (Maximum External Clock Buffer Delay)
- (Maximum External Output Register Clock to Q Delay)
- (Maximum Clock Jitter for TClock)
- (Maximum Clock Jitter for R4000 Internal Clocks)

A block-level diagram of a system without phase-lock, employing the R4000 processor and an external agent implemented as a gate-array, is shown in Figure 10-4.

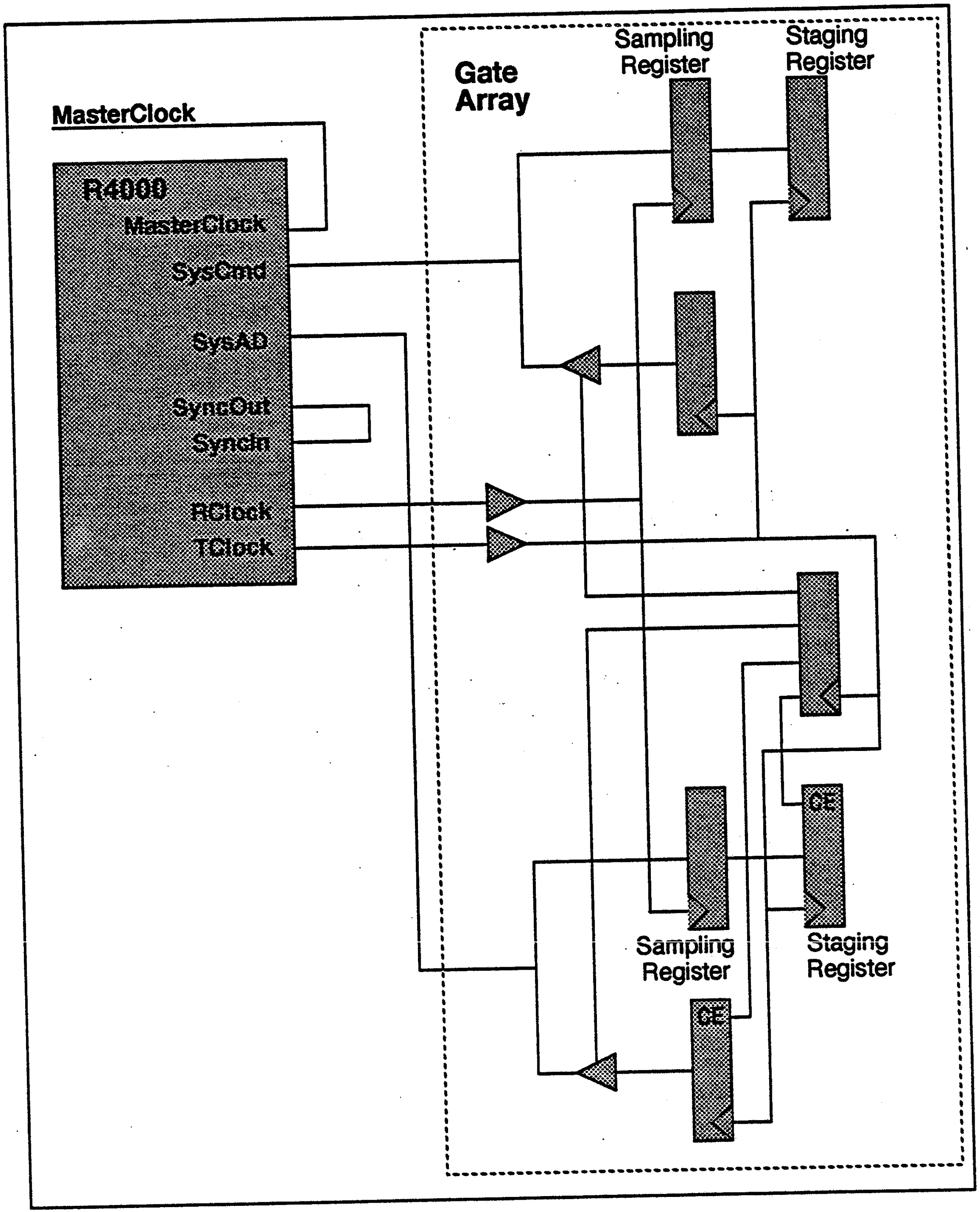


Figure 10-4 System Without Phase Lock Employing the R4000 Processor

Interface to CMOS Logic System

When interfacing to CMOS logic system, matched delay clock buffers are used to allow the processor to generate aligned clocks for the external logic. One of the matched delay clock buffers is inserted in the processor's SyncOut SyncIn clock alignment path, skewing SyncOut, MasterOut, RClock, and TClock to lead MasterClock by the delay of the matched delay clock buffer while leaving PClock aligned with MasterClock. The remaining matched delay clock buffers can be used to generate a buffered version of TClock aligned with MasterClock. The alignment error of the buffered version of TClock is the sum of the maximum delay mismatch of the matched delay clock buffers and the maximum clock jitter of TClock. The buffered version of TClock is used to clock registers that sample processor outputs, as the global system clock for the discrete logic that forms the external agent, and to clock registers that drive processor inputs.

The transmission time for a signal from the processor to an external agent composed of discrete CMOS logic devices can be calculated from the following equation:

Transmission Time = (TClock period) - (t_{DO} for R4000)
- (External Sample Register Setup Time)
- (Maximum External Clock Buffer Delay Mismatch)
- (Maximum Clock Jitter for R4000 Internal Clocks)
- (Maximum Clock Jitter for TClock)

The transmission time for a signal from an external agent composed of discrete CMOS logic devices can be calculated from the following equation:

Transmission Time = (TClock period) - (t_{DS} for R4000)
- (Maximum External Output Register Clock to Q Delay)
- (Maximum External Clock Buffer Delay Mismatch)
- (Maximum Clock Jitter for R4000 Internal Clocks)
- (Maximum Clock Jitter for TClock)

Note that, using this clocking methodology, the hold time of data driven from the processor to an external sampling register is a critical parameter. In order to guarantee hold time, the minimum output delay of the processor, t_{DM}, must be greater than the sum of the minimum hold time for the external sampling register, the maximum clock jitter for R4000 internal clocks, the maximum clock jitter for TClock, and the maximum delay mismatch of the external clock buffers.

A block-level diagram of a system without phase lock employing the R4000 processor and an external agent composed of both a gate array and discrete CMOS logic devices is shown in Figure 10-5.

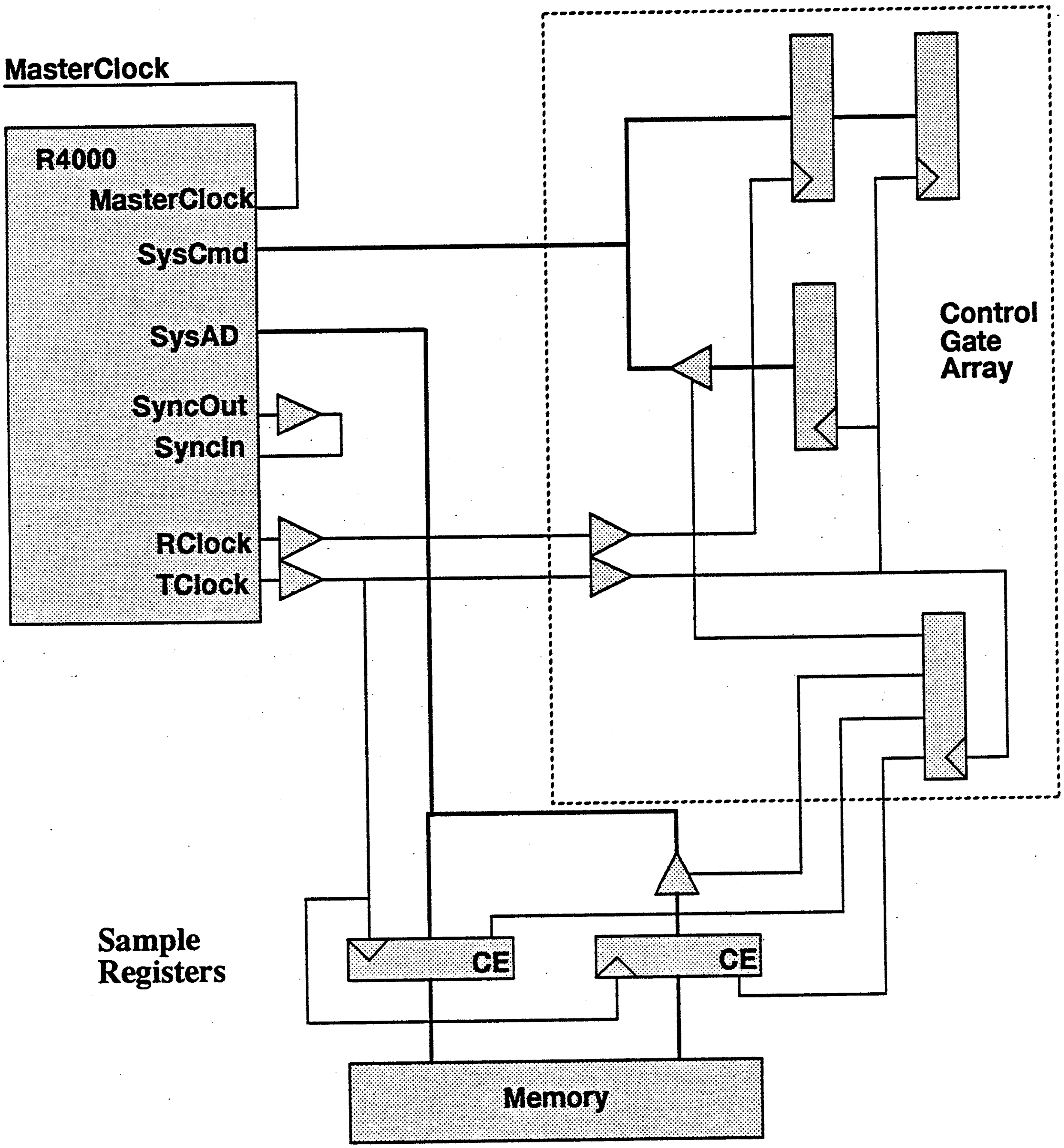


Figure 10-5 System Without Phase Lock Employing the R4000 Processor

Processor Status Outputs

The R4000 processor provides eight status outputs, Status(7:0), that change with each rising edge of MasterClock to indicate the processor's internal state during each of the two most recent PCycles. Status(7:0) is treated as two fields: Status(3:0) indicates the processor's internal state during the most recent PCycle, and Status(7:4) indicates the processor's internal state during the PCycle preceding the most recent PCycle. The encoding of processor internal state for Status(7:4) or Status(3:0) is shown in Table 10-1. The four-bit decode describes the instruction occupying the WB stage during a given PCycle.

Table 10-1 Encoding of Processor Internal State for Status(7:4) or Status(3:0).

Status(7:4) or Status(3:0)	Processor internal state	
0	Run cycle:	Other integer instruction
1	Run cycle:	Integer Load
2	Run cycle:	Integer Untaken Branch
3	Run cycle:	Integer Taken Branch
4	Run cycle:	Integer Store
5	Reserved	
6	Reserved	
7	Run cycle:	Killed by integer slip
8	Stall cycle:	Other stall type
9	Stall cycle:	Primary Instruction Cache
a	Stall cycle:	Primary Data Cache
b	Stall cycle:	Secondary Cache
c	Run cycle:	Floating-Point
d	Run cycle:	Killed by branch
e	Run cycle:	Killed by exception
f	Run cycle:	Killed by floating point slip

Cache Organization, Operation, and Coherency

11

This chapter contains a description of the cache memory hierarchy, the operation of the primary and secondary caches and the R4000's interface to the secondary cache, and cache-coherent operation in a multiprocessor system.

Cache Organization

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

Primary Caches

The R4000 maintains the folllowing four primary cache states:

- *Invalid*
- *Shared*
- *Clean Exclusive*
- *Dirty Exclusive*

The cache state of a line in the processor's primary cache indicates the *validity, shared, dirty,* and *ownership* attributes of the cache line.

- A cache line that does not contain valid information must be marked *invalid.*
- A cache line in any state other than invalid contains valid information.
- A cache line that is present in more than one cache in the system is said to be *shared* and must be in one of the shared states.

- A cache line that is present in exactly one cache in the system is said to be *exclusive* and may be in one of the exclusive states.
- A cache line that contains data that is consistent with memory is said to be *clean* and may be in one of the clean states.
- A cache line that contains data that is not consistent with memory is said to be *dirty* and must be in one of the dirty or shared states.

A cache line can have only one *owner* at a time. The owner of a cache line is responsible for providing the current contents in the cache line on any read request. A cache line is owned by the processor if the state of the secondary cache line is dirty exclusive, dirty shared, or if the state of the primary cache line is dirty exclusive when no secondary cache is present. Clean cache lines are always owned by memory.

In addition, if the owner of a cache line is a processor, that processor is responsible for writing the cache line back to memory when it is replaced in the course of either satisfying a cache miss or during the execution of a Writeback or Writeback Invalidate cache instruction.

Primary Instruction Cache

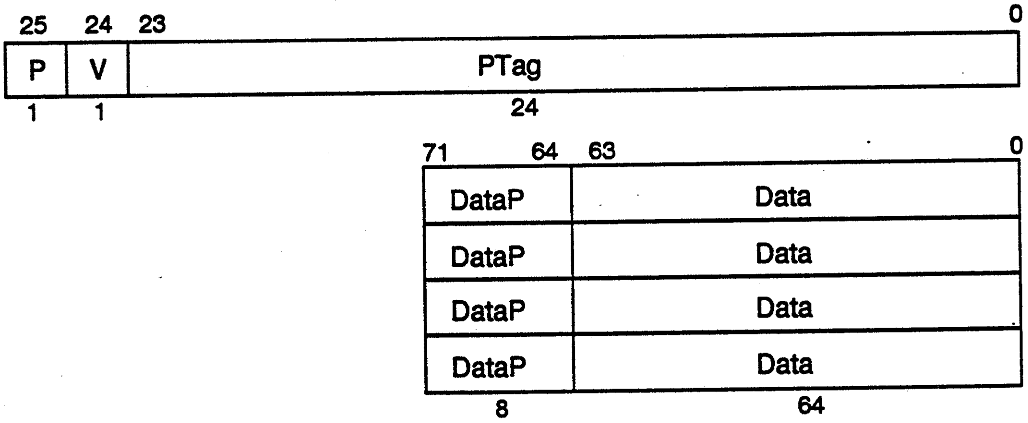
The R4000 primary instruction cache is:

- Direct-mapped.
- Indexed with a virtual address.
- Checked with a physical tag.
- Organized with either a 4-word (16-byte) or 8-word (32-byte) cache line.

The primary instruction cache states are determined by the following cache line attribute:

Invalid	The cache line does not contain valid information.

Each line of instruction cache data has an associated 26-bit tag. The tag contains a 24-bit physical address, a single valid bit, and a parity bit. Byte parity is used on the instruction data. The format of an 8-word (32 byte) primary instruction cache line is shown in Figure 11-1.



PTag physical tag (bits 35..12 of the physical address)
V valid bit
Data cache data
P even parity for the *PTag* and *V* fields
DataP even parity-1 parity bit per byte of data

Figure 11-1 Format of R4000 8-Word Primary Instruction Cache Line

The 4-word primary instruction cache line is accessed using 2 PCLK cycles; the 8-word primary instruction cache line is accessed using 4 PCLK cycles.

Primary Data Cache

The R4000 primary data cache is:

- Write-back.
- Direct-mapped.
- Indexed with a virtual address.
- Checked with a physical tag.
- Organized with either a 4-word (16-byte) or 8-word (32-byte) cache line.

The primary cache states indicate the following cache line attributes:

Invalid
The cache line does not contain valid information.

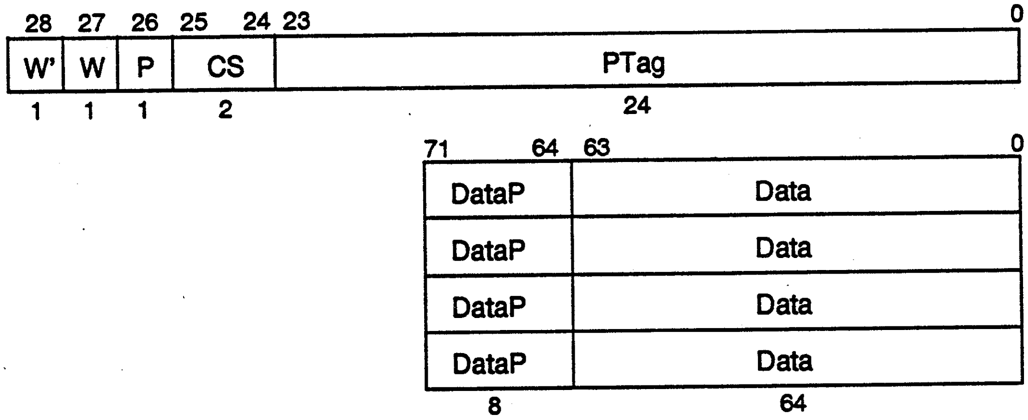
Shared
The cache line contains valid information and may be present in another cache. The cache line may or may not be consistent with memory, and may or may not be owned.

Clean Exclusive
The cache line contains valid information and is not present in any other cache. The cache line is consistent with memory and is not owned.

Dirty Exclusive
The cache line contains valid information and is not present in any other cache. The cache line is inconsistent with memory and is owned by a processor.

Each line of primary cache data has an associated 29-bit tag. The tag contains a 24-bit physical address, 2-bit cache line state, and a write-back bit. The write-back bit has its own parity bit, and the tag and cache line state share a parity bit.

Figure 11-2 shows the format of a 8-word (32 byte) primary data cache line.



W' even parity for the write-back bit

W write-back bit (set if data is modified and different from secondary cache and memory)

P even parity for the PTag and CS fields

CS primary cache state

- *0* Invalid—all R4000 configurations
- *1* Shared (either Clean or Dirty)—R4000MC configurations only
- *2* Clean Exclusive—R4000SC and MC configurations
- *3* Dirty Exclusive—all R4000 configurations

PTag physical tag (bits 35..12 of the physical address)

DataP even parity for the data

Data cache data

Figure 11-2 Format of R4000 8-Word Primary Data Cache Line

In all R4000 processors, the *W* (write-back) bit, not the cache state, indicates when the primary cache contains modified data that must be written back to memory or the secondary cache.

In the R4000PC, the states *Invalid* and *Dirty Exclusive* are used to describe the cache line. In the R4000SC, the states *Invalid, Clean Exclusive,* and *Dirty Exclusive* are used to describe the cache line. In the R4000MC, all four states are used to describe the cache line and to control whether load and store operations need to access the secondary cache for coherency purposes. The effects of load and store operations for the four primary cache states are described in Table 11-2.

Table 11-1 R4000MC Data Cache Coherency States

Primary Cache States	Secondary Cache States	Action on Load	Action on Store
Invalid	All	Miss	Miss
Shared	Shared	None	Read secondary tag. If the coherency algorithm is Update on Write then send update and set the secondary cache state to Dirty Shared If the coherency algorithm is Invalidate on Write, then send invalidate and set the primary and secondary cache states to Dirty Exclusive.
	Dirty Shared	None	
	Dirty Exclusive	None	If Dirty Exclusive, set the primary cache state to Dirty Exclusive.
Clean Exclusive	Clean Exclusive	None	Set the primary and secondary cache states to Dirty Exclusive.
	Dirty Exclusive	None	Set the primary data cache state to Dirty Exclusive.
Dirty Exclusive	Dirty Exclusive	None	None

When the primary cache is filled from the secondary cache, the secondary cache state is mapped into the primary cache state by mapping the *Shared* and *Dirty Shared* secondary states into the *Shared* primary state. The *Dirty Exclusive* primary state allows the primary cache to be written without a secondary access.

The 4-word primary data cache line is accessed using two PCLK cycles; the 8-word primary data cache line is accessed using four PCLK cycles.

Secondary Cache

The R4000 is designed to operate with an external secondary cache. The secondary cache is accessible to the processor and to the system interface. The cache contains data, cache tags and cache line state bits.

R4000 processors support an optional external secondary cache which can be configured at chip reset as either a one joint cache, or a separate I-cache and D-cache. This secondary cache is:

- Write-back.

- Direct-mapped.
- Indexed with a physical address.
- Checked with a physical tag.
- Organized with either a 4-word (16-byte), 8-word (32-byte), 16-word (64-byte), or 32-word (128-byte) cache line.

The secondary cache states indicate the following cache line attributes:

Invalid — The cache line does not contain valid information.

Shared — The cache line contains valid information and may be present in another cache. The cache line may or may not be consistent with memory, and is not owned.

Dirty Shared — The cache line contains valid information and may be present in another cache. The cache line is inconsistent with memory and is owned.

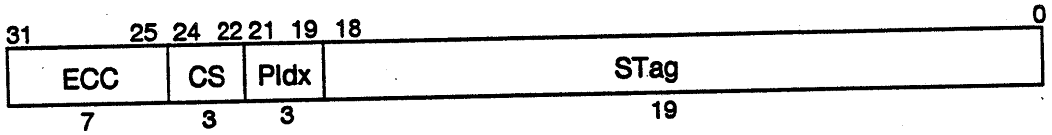
Clean Exclusive — The cache line contains valid information and is not present in any other cache. The cache line is consistent with memory and is not owned.

Dirty Exclusive — The cache line contains valid information and is not present in any other cache. The cache line is inconsistent with memory and is owned.

The primary cache state *shared* corresponds to the secondary cache states *shared* and *dirty shared.*

The secondary-cache line has an associated 19-bit tag that contains bits 35..17 of the physical address, a 3-bit primary cache index, and a 3-bit cache line state. These 25 bits are protected by a 7-bit error correction code (ECC).

Figure 11-3 shows the format of the R4000 secondary-cache line.



ECC ECC for secondary tag
CS secondary-cache state
0 Invalid
1 reserved
2 reserved
3 reserved
4 Clean Exclusive
5 Dirty Exclusive
6 Shared
7 Dirty Shared
PIdx primary cache index (bits 14..12 of the virtual address)
STag physical tag (bits 35..17 of the physical address)

Figure 11-3 Format of R4000 Secondary Cache Line

The secondary-cache state (*CS* bits) indicates whether

- The cache line data and tag are valid.
- The data is at least potentially present in the caches of other processors (*Shared* versus *Exclusive*).
- The processor is responsible for updating main memory (*Clean* versus *Dirty*).

The primary caches are a subset of the secondary cache. The processor maintains this subset property by checking and invalidating the primary caches, if necessary, when a secondary cache line is replaced.

The PIdx field provides the processor with an index to the virtual (not physical) address of primary cache lines that may contain data from the secondary cache line.

A second function of the PIdx field is to detect a cache alias. If the physical address tag matches during a data reference to the secondary cache (S-cache), but the PIdx field does not match the appropriate bits in the virtual address, the reference was made from a different virtual address than the one that created the secondary-cache line. Since this could create a cache alias, the processor signals this condition by taking a Virtual Coherency exception (see Chapter 5, *Exception Processing*).

Primary and Secondary Cache Interaction

The primary caches are proper subsets of the secondary cache. In the R4000PC, the Invalid and Dirty Exclusive states are used to describe the cache line. In the R4000SC, the Invalid, Clean Exclusive, and Dirty Exclusive states are used to describe the cache line. In the R4000MC, all four states are used to describe the cache line and to control whether load and store operations need to access the secondary cache for coherency purposes. The effects of the load and store operations for the four primary cache states are described in Table 11-2. This table can be better understood by realizing that there may be many primary cache lines for each secondary cache line.

Table 11-2 R4000MC Data Cache Coherency States

Primary Cache States	Secondary Cache States	Action on Load	Action on Store
Invalid	All	Miss	Miss
Shared	Shared	None	Read secondary tag. If the coherency algorithm is Update on Write, then send update and set the secondary cache state to Dirty Shared. If the coherency algorithm is Invalidate on Write, then send invalidate and set the primary and secondary cache states to Dirty Exclusive.
	Dirty Shared	None	
	Dirty Exclusive	None	If Dirty Exclusive, set the primary cache state to Dirty Exclusive.
Clean Exclusive	Clean Exclusive	None	Set the primary and secondary cache states to Dirty Exclusive.
	Dirty Exclusive	None	Set the primary data cache state to Dirty Exclusive.
Dirty Exclusive	Dirty Exclusive	None	None

Upon a cache miss in both the primary and the secondary cache, the missing secondary cache line is loaded from memory into the secondary cache first. The the appropriate subset is loaded into the primary cache. When the primary cache is filled from the secondary cache, the secondary cache state is mapped into the primary cache state by mapping the *Shared* and *Dirty Shared* secondary states into the *Shared* primary cache state. The *Dirty Exclusive* primary cache state allows the primary cache to be written without a secondary cache access.

Cache Line Ownership

The R4000 requires that cache lines have a single owner at all times. The owner of a cache line is responsible for providing the current contents in the cache line to any read requestor. The ownership of a cache line is set and maintained as follows:

- A processor assumes ownership of the cache line if the state of the secondary cache line is *dirty shared, dirty exclusive,* or if the state of the primary cache line is *dirty exclusive* when no secondary cache is present. For responses to processor coherent read requests in which the data is returned with an indication that it must be loaded in the *dirty shared* or *dirty exclusive* state, the cache state is set when the last word of read response data is returned. Therefore, the processor assumes ownership of the cache line when the last word of read response data is returned.
- The processor gives up ownership of a cache line when the state of the cache line changes to *invalid, shared,* or *clean exclusive.* For processor coherent write requests the state of the cache line changes to *invalid* if the cache line is replaced, or to *clean exclusive* or *shared* if the cache line is retained. In either case, the cache state transition occurs when the last word of write data is transmitted to the external agent. Therefore, the processor gives up ownership of the cache line when the last word of write data is transmitted to the external agent.
- For external requests, other than read responses, any cache state change associated with the external request, including a change of ownership, occurs at the completion of the external request.
- Clean cache lines are always owned by memory.

Cache Operation

This section describes the operation of the R4000 caches.

Cache Coherency

The R4000 processor manages its primary and secondary caches using a write-back methodology; that is, it stores write data into the caches. A modified cache line is not written back to memory until the cache line is replaced either in the course of satisfying a cache miss or during the execution of a Writeback or Writeback Invalidate cache instruction. When the contents of a cache line is not consistent with memory, it is said to be *dirty*. Many systems, in particular multi-processor systems, or systems that employ input/output (IO) devices that are capable of direct memory access (DMA), may require the system to behave as if the caches are always consistent with memory and each other. Schemes for maintaining consistency between multiple write-back caches or between write-back caches and memory are referred to as *cache coherency protocols.*

The R4000MC processor, in its secondary cache mode, provides a set of cache states and mechanisms for manipulating the contents and state of the cache that are sufficient to implement a variety of cache coherency protocols, both snoopy and directory-based. In particular, the processor supports both the *write-invalidate* and *write-update* protocols simultaneously.

The coherency protocol for lines in the cache is controlled by bits in the translation look-aside buffer (TLB) on a per-page basis. Specifically, the TLB contains three bits per entry that control the coherency attributes of a page. The three bits are encoded to provide five possible coherency attributes per page:

- *uncached,*
- *sharable,*
- *update,*
- *exclusive,* and
- *noncoherent.*

A processor in the no-secondary cache mode supports only the uncached and noncoherent coherency attributes.

If a page has the *uncached* coherency attribute, the processor issues a word or partial word read or write directly to main memory for any load or store to a location within that page. Lines within an uncached page are assumed never to be cache-resident.

If the coherency attribute is *sharable*, the processor issues a coherent block read for a load miss to a location within the page, and a coherent block read that requests exclusivity for a store miss to a location within the page. In most systems, coherent reads require snoops or directory checks to occur; noncoherent reads do not. A coherent read that requests exclusivity implies that the processor functions most efficiently if the requested cache line is returned to it in an exclusive state, but the processor still performs correctly if the cache line is returned in a shared state. Cache lines within the page are managed with a *write invalidate* protocol; that is, the processor issues an *invalidate* on a store hit to a shared cache line.

If the coherency attribute is *update*, the processor issues a coherent block read for a load or store miss to a location within the page. Cache lines within the page are managed with a *write update* protocol; that is, the processor issues an update on a store hit to a shared cache line.

If the coherency attribute is *exclusive*, the processor issues a coherent block read that requests exclusivity for a load or store miss to a location within the page. Cache lines within the page are managed with a *write invalidate* protocol. Load Linked Store Conditional instruction sequences must ensure that the link location is not in a page managed with the exclusive coherency attribute.

If the coherency attribute is *noncoherent*, the processor issues a noncoherent block read for a load or store miss to a location within the page.

The behavior of the processor on load misses, store misses, and store hits to shared cache lines for each of the coherency attributes is summarized in Table 11-3.

Table 11-3 Coherency Attributes and Processor Behavior

Attribute	Load Miss	Store Miss	Store Hit Shared
Uncached	Main memory read	Main memory write	NA
Noncoherent	Noncoherent read	Noncoherent read	Invalidate *
Exclusive	Coherent read exclusive	Coherent read exclusive	Invalidate *
Sharable	Coherent read	Coherent read exclusive	Invalidate
Update	Coherent read	Coherent read	Update

NOTE: *This should not occur under normal circumstances.

The following sections describe:

- The cache state transitions performed by the processor during execution.
- The mechanisms provided for an external agent to manipulate the state and contents of the primary and secondary cache.

Cache State Changes

The initial state of a cache line is specified by the external agent when it supplies the cache line. During the course of processor execution, the processor may change the state of a cache line. The following events cause changes to the state of the cache:

- A store to a *clean exclusive* cache line causes the state to be changed to *dirty exclusive* in both the primary and secondary caches.
- A store to a *shared* cache line, that is a line marked *shared* in the primary cache and either *shared* or *dirty shared* in the secondary cache, will cause the processor to issue either an invalidate request or an update request depending on the coherency attribute in the TLB entry for the page containing the cache line. Upon successful completion of an invalidate, the processor completes the store and changes the state of the cache line to *dirty exclusive* in both the primary and secondary caches. Upon successful completion of an update, the processor completes the store and changes the state of the cache line to *shared* in the primary cache and *dirty shared* in the secondary cache if *dirty shared mode* is enabled. *Dirty shared* mode is programmable via the boot-time mode control interface described in Chapter 12. If *dirty shared* mode is not enabled, the state of the primary and secondary caches will be left unchanged after successful completion of an update.

Figure 11-4 and Figure 11-5 are state diagrams of the Primary and Secondary Caches respectively.

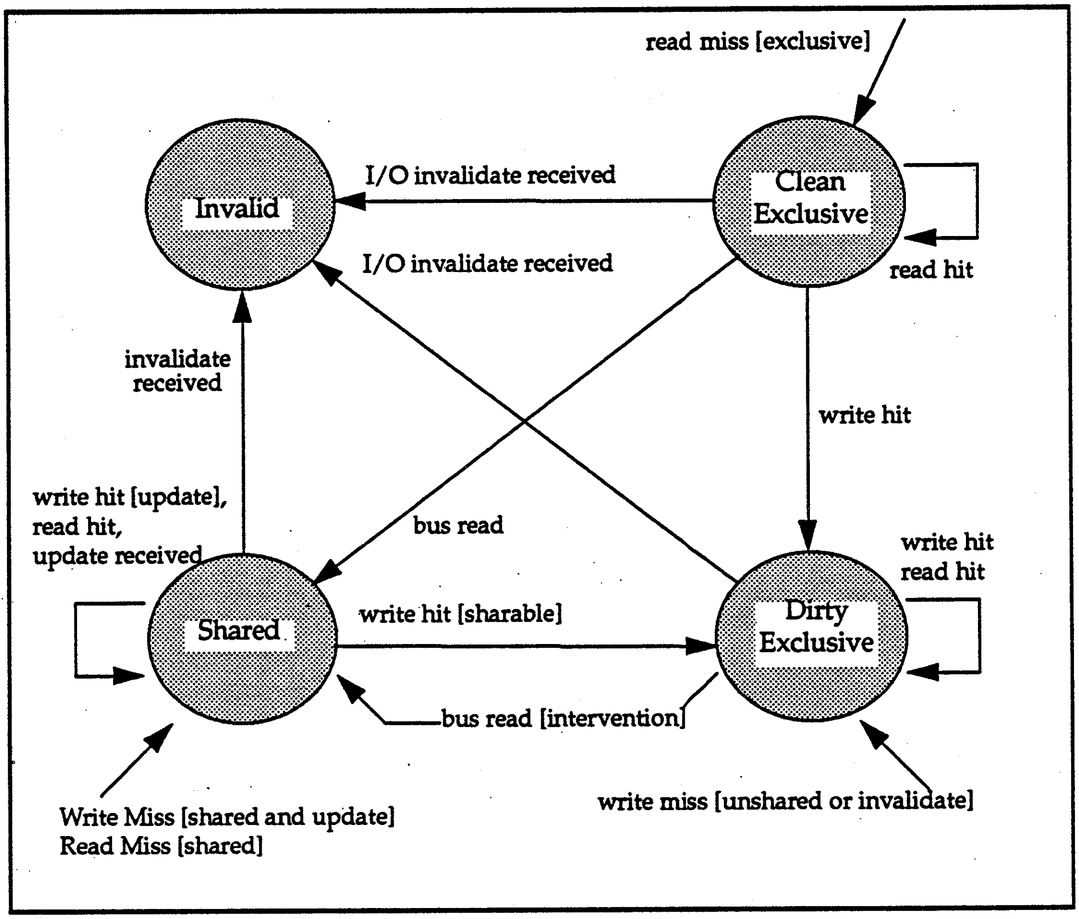


Figure 11-4 Primary Data Cache State Diagram

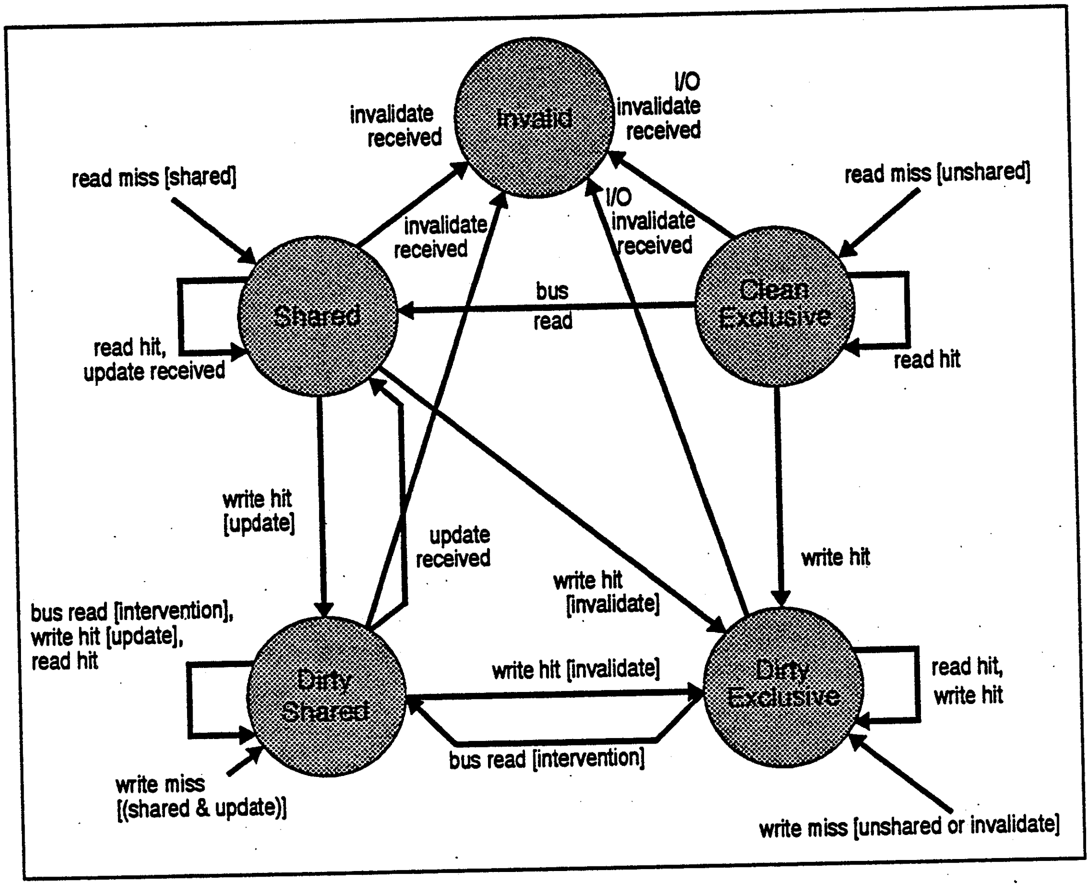


Figure 11-5 Secondary Cache State Diagram

Cache Line Write-Back

If the cache line is in the *dirty exclusive* or *dirty shared* state in the secondary cache, the processor writes a cache line back to memory when it is replaced, eitherin the course of satisfying a cache miss or during the execution of a Writeback or Writeback Invalidate cache instruction. When the processor writes a cache line back to memory, it does not ordinarily retain a copy of the cache line, and the state of the cache line is changed to *invalid*. However, under certain conditions related to load linked and store conditional, or if a cache line is written back by the Hit Writeback cache instruction, the processor retains a copy of the cache line. If the cache line is retained, the processor changes the cache line state to *clean exclusive* if the secondary cache state was *dirty exclusive* before the write, or *shared* if the secondary cache state was *dirty shared* before the write.

Whether or not the processor is retaining the line is signaled by the processor during a write.

Manipulation of the Caches by an External Agent

The R4000 provides the following mechanisms for an external agent to examine and manipulate the state and contents of the primary and secondary caches:

- An external agent must specify the state in which data, supplied in response to a processor read request, loads into the processor's caches. Data may be loaded in any of the four valid secondary cache states. Data returned by the external agent must not be marked invalid. The secondary cache state will be mapped by the processor to a primary cache state as previously described.
- An external agent may issue a snoop request to the processor causing the processor to return the secondary cache state of the specified cache line. At the same time and according to a function supplied by the external agent, it atomically changes the state of the specified cache line in both the primary and secondary caches.
- An external agent may issue an invalidate request or an update request to the processor. An invalidate request causes the processor to change the state of the specified cache line to invalid in both the primary and secondary caches. An update request causes the processor to write the specified data element into the specified cache line, and either change the state of the cache line to *shared* in both the primary and secondary caches, or leave the state of the cache line unchanged, depending on the nature of the update request. An external agent may issue updates—without changing the state of the cache line—to cache lines that are in either exclusive or shared states
- An external agent may issue an intervention request causing the processor to return the secondary cache state of the specified cache line, and, under certain conditions related to the state of the cache line and the nature of the intervention request, the contents of the specified secondary cache line. At the same time and according to a state change function specified by the external agent, the processor atomically changes the state of the specified cache line in both the primary and secondary caches.

Ordering Considerations

Many cache coherent multiprocessor systems must obey ordering constraints on stores to shared data; therefore, they exhibit the same behavior as a uniprocessor system in a multiprogramming environment. A multiprocessor system that exhibits such behavior is said to be *strongly ordered.*

A typical algorithm for testing strong ordering follows:

Given - Locations X and Y have no particular relationship; i.e., they are not in the same cache line.

Processor A performs a store to location X at the same time processor B performs a store to location Y. Next, processor A does a load from location Y at the same time that processor B does a load from location X. In order for the system to be considered strongly ordered, either processor A must load the new value of Y, or processor B must load the new value of B,. or both processors A and B must load the new values of Y and X, respectively, under all conditions. If both processors A and B load the old values of Y and X, respectively, under any conditions, the system does not meet the requirements for strong ordering.

The algorithm to test for strong ordering is summarized below.

Processor A	Processor B
Store to location X	Store to location Y
Load from location Y	Load from location X

In order for this strong ordering test algorithm to succeed, stores must have a global ordering in time; that is, every processor in the system must agree that either the store to location X preceded the store to location Y, or the store to location Y preceded the store to location X. If this global ordering is enforced, this test algorithm for strong ordering will succeed.

The requirements to achieve strong ordering translate into a need for precise control of when the processor restarts in relationship to a change in cache state initiated by an external coherence request. Specifically, before allowing the processor to restart after completion of a processor coherence request, system designers must ensure completion of any cache state changes resulting from an external coherence request that occurs before a processor coherence request.

The R4000 processor obeys the following paradigms for restart after issuing a coherence request.

For coherent read requests, the processor will restart after either of the following conditions, unless a processor invalidate or update request is unacknowledged:

- The requested double word is transmitted to the processor if sub-block ordering is enabled, or
- The last word in the block is transmitted to the processor if sequential ordering is enabled.

Any external requests that must be completed before the read is complete must be issued to the processor before the read response is issued.

For coherent write requests, the processor restarts after the write request is complete; that is, after the last double word of data associated with the write request has been transmitted to the external agent, ***unless*** a processor read request is pending or a processor invalidate or update request is unacknowledged.

For invalidate and update requests, the processor restarts after the assertion of **IvdAck*** or **IvdErr***, ***unless*** a processor read request is pending or it is processing an external request when **IvdAck*** or **IvdErr*** are asserted.

If **IvdAck*** or **IvdErr*** are asserted during or after the first cycle that the external agent asserts **ExtRqst***, the processor will accept the external request and complete any cache state changes associated with the external request before restarting.

If **IvdAck*** or **IvdErr*** are ***not*** asserted during or after the first cycle that the external agent asserts **ExtRqst***, the processor will restart before beginning the external request.

In summary, external requests that must be completed before a processor invalidate or update completes can be completed providing the processor receives an asserted **ExtRqst*** by the external agent before or during the same cycle either **IvdAck*** or **IvdErr*** are asserted.

Coherence Conflicts

This section explains how the R4000 handles coherence conflicts caused by competing coherence requests from the processor and an external source. The topics in this section are:

- How Coherence Conflicts Arise
- System Implications of Coherence Conflicts
- Handling Coherence Conflicts

This material applies only to the R4000MC which can issue processor coherence requests and accept external coherence requests.

How Coherency Conflicts Arise

The R4000MC processor issues processor coherence requests and accepts external coherence requests.

Processor coherence requests are:

- Processor coherent read requests
- Invalidate requests
- Update requests

External coherence requests are

- External invalidate
- Update
- Snoop
- Intervention requests

Because of the overlapped nature of the system interface it is possible for processor coherence requests and external coherence requests to *conflict*. That is, it is possible for an external coherence request to reference an address that targets the same cache line as a pending processor read request or an unacknowledged processor invalidate or update request. The processor does not contain comparators to detect such conflicts. The processor uses the secondary cache as the single point of reference to determine the coherency actions it takes, and only checks the state of the secondary cache at specific times.

For pending processor coherent read requests, conflicting external requests cannot affect the behavior of the processor. The processor only issues a read request for a particular cache line if it does not have a copy of that cache line. Therefore, any external coherence request that targets a cache line that is also the target of a pending processor coherent read request will not find the line present in the cache. External coherence requests do not change the state of the cache unless the cache line they target is present. Since no change can be made to the state of the cache for the line that is the target of the pending processor read request, no external coherence requests can effect the read request. Therefore, external coherence requests that conflict with a pending processor coherent read request may be issued to the processor and will effectively be discarded by the processor.

For processor invalidate and update requests the cancellation mechanism is provided to signal conflicts to the processor. If a conflicting external coherence request is submitted while a processor invalidate or update request has been issued but not yet acknowledged, an external agent may cancel the processor invalidate or update. This applies to compulsory updates and invalidates only. This is accomplished by setting the cancellation bit in the command for the coherence request. The processor, upon receiving an external coherence request with the cancellation bit set, will consider its invalidate or update request to be acknowledged and canceled, and will re-access the secondary cache and re-evaluate the cache state to determine the correct action. This may result in the invalidate or update request being re-issued, or it may result in the issue of a read request instead.

An external agent is only allowed to assert the cancellation bit with an external coherence request when a processor invalidate or compulsory update request is currently unacknowledged. If an external coherence request is issued with the cancellation bit set when there is no unacknowledged processor invalidate or update request, the behavior of the processor is undefined.

Processor potential update requests may not be cancelled. Potential updates are always issued under processor read requests and become compulsory only after the response to the processor read request is returned to the processor in one of the shared states. If an external coherence request is issued with the cancellation bit set when a processor invalidate or update request is still potential, in other words, while a processor read request is currently pending, the behavior of the processor is undefined.

An external agent may issue an external coherence request that is in conflict with an unacknowledged processor invalidate or update request without setting the cancellation bit. In this case, the processor will be unaware of the conflict and will not re-evaluate the cache state to determine the correct action. It will simply wait for an acknowledge to its invalidate or update request just as it would for any invalidate or update request. A system employing the R4000 may not behave correctly under these circumstances.

Note that it is not possible for external coherence requests to conflict with processor write requests since external requests are not accepted while a processor write request is in progress. The interactions between processor coherence requests and conflicting external coherence requests, tabulated by processor state, is summarized in Table 11-4 and Table 11-5.

The processor can be in one of the following states:

- Idle

  No processor transactions currently pending.

- Read Pending

  A processor coherent read request has been issued, but the read response has not yet been received.

- Potential Update Unacknowledged

  A processor update request has been issued while a processor coherent read request is pending but has not yet been acknowledged, and, by definition, the response to the coherent read request has not yet been received.

- Invalidate or Update Unacknowledged

  A processor invalidate or update request has been issued but has not yet been acknowledged and, by definition, there is not a processor coherent read request pending.

Table 11-4 Coherence Conflicts Summary

Processor State	Conflicting External Coherence Request			
	Invalidate	Invalidate w/Cancel	Update	Update w/Cancel
Idle	NA	Undefined	NA	Undefined
Read Pending	OK	Undefined	OK	Undefined
Potential Update Unacknowledged	OK	Undefined	OK	Undefined
Invalidate or Update Unacknowledged	OK *	OK	OK *	OK

* This may cause incorrect system operation and should not normally be allowed to occur.

Table 11-5 Coherence Conflicts Summary

Processor State	Conflicting External Coherence Request			
	Intervention	Intervention w/Cancel	Snoop	Snoop w/Cancel
Idle	NA	Undefined	NA	Undefined
Read Pending	OK	Undefined	OK	Undefined
Potential Update Unacknowledged	OK	Undefined	OK	Undefined
Invalidate or Update Unacknowledged	OK *	OK	OK *	OK

* This may cause incorrect system operation and should not normally be allowed to occur.

System Implications of Coherence Conflicts

The constraints that the processor places on the handling of conflicting coherency transactions have certain implications for the design of a multiprocessor system employing the R4000. This section will consider, as an example, a particular snoopy, split-read, bus-based system and the requirements for that system to correctly handle coherence conflicts.

System Model

The system model consists of the following components:

1. Four processor subsystems, each consisting of an R4000 processor, a secondary cache, and an external agent. The agent communicates with the R4000, accepting processor requests and issuing external requests, and with the system bus likewise issuing and receiving bus requests.
2. A memory subsystem that communicates with main memory and the system bus.

3. A system bus with the following characteristics:

   - It is a multiple master, request based, arbitrated bus in which an agent that wishes to perform a transaction on the bus must request the bus and wait for global arbitration logic to supply a grant signal before assuming mastership of the bus. Once mastership has been granted, the agent may begin a transaction.
   - It supports a read transaction, read exclusive transaction, write transaction, and invalidate transaction.
   - It is a split-read bus in that independent transactions may occur on the bus between a read request from a particular agent and the return of data by the target of the read request. The return of data by the target of the read request will be referred to as the read response.
   - It is a snoopy bus in that all agents connected to the bus must monitor all of the traffic on the bus to correctly maintain cache coherency.

I/O is not considered in this system model.

Coherency Model

The goal, for purposes of this example, is to implement a simple write invalidate cache coherency protocol for this system model that maintains consistency between all of the caches in the system and main memory. Attention is focused on the interactions between the system bus and the R4000, and the handling of coherence conflicts that arise in this system.

The coherency model for the system is as follows:

- All pages in the system are maintained either with the noncoherent coherency attribute or with the sharable coherency attribute.
- The handling of noncoherent data will not be considered.
- Using the sharable coherency attribute allows data to be shared between the four caches in the system with a write invalidate cache coherency protocol.
- The secondary cache states used are *invalid, shared, clean exclusive,* and *dirty exclusive.*
- The secondary cache state *dirty shared* is not used in this coherency model.

When a processor misses in both caches on a load it issues a read request. The external agent translates this to a read request on the bus. The returned data may be loaded in either the state *clean exclusive* or *shared* based on a shared indication returned on the bus with the read response. The shared indication is based on the result of an intervention request to the processor for the cache line of interest, and is supplied by the external agents that are a part of the other three processor subsystems. When a processor misses in both caches on a store, it issues a read request desiring exclusivity; this is translated to a read exclusive on the bus and the data is loaded in the state *dirty exclusive*. When a processor hits in the cache on a store to shared data, it issues an invalidate request which must be forwarded to the system bus before the store can be completed and the state changed to *dirty exclusive*.

When an external agent observes a coherent read request on the system bus it does not take any immediate action. Instead, the external agent issues an intervention request to the processor for the read request during the read response associated with the read request. This is referred to as a *response complete* read model; that is, the read is treated as complete only after the read response has occurred. This model requires that cache interrogation for a read must not occur until the read response occurs, as described, in order to maintain consistency.

At the end of the read response, each external agent supplies an indication on the bus of whether it was able to obtain the state of the cache line that is the target of the read via an intervention request, and if so, the external agent supplies an indication of sharing or *takeover*. Takeover occurs when an external agent discovers that its processor has a copy of the cache line that is the target of the read in the state *dirty exclusive*. If any external agent is unable to obtain the state of the cache line that is the target of the read because it is unable to initiate an intervention request, the read response is extended until all external agents have obtained the state of the cache line from their processors.

The response from an external agent at the end of a read response depends on whether the read request was an ordinary read request or a read exclusive request.

For an ordinary read request, an external agent indicates *shared* at the end the read response if it finds that its processor has a copy of the requested cache line in the state *clean exclusive* or *shared*. If the current state of the cache line is *clean exclusive*, the external agent causes the processor to change the state of the cache line to *shared*. An external agent will indicate both *shared* and *takeover* at the end of a read response to an ordinary read request if it finds that its processor has a copy of the requested cache line in the state *dirty exclusive*. Having indicated takeover, the external agent supplies the contents of the cache line (returned by the processor in response to its intervention request) over the bus to the read requester, and causes the processor to change the state of the cache line to *shared*. At the same time the cache line is supplied to the read requester, it is also written back to memory.

For a read exclusive request, an external agent never indicates *shared* at the end of the read response, regardless of the state its processor has the cache line in. If the current state of the cache line is *clean exclusive* or *shared*, the external agent causes the state of the cache line to be changed to invalid. If the current state of the cache line is *dirty exclusive*, the external agent indicates *takeover* but not *shared*. Having indicated *takeover*, the external agent supplies the contents of the cache line over the bus to the read requester, and causes the processor to change the state of the cache line to invalid. At the same time the cache line is supplied to the read requester, it is also written back to memory.

An invalidate request is considered complete as soon as it appears on the system bus. When an external agent observes an invalidate request on the system bus, it must react as if the invalidate has changed the state of all caches at that instant.

An external agent takes no action in response to the appearance of a write request on the bus.

Handling Coherence Conflicts

Coherence conflicts can be examined based on the current state of the processor. In particular, the processor may have a coherent read request pending, or it may have an invalidate request unacknowledged, or it may not have any requests pending or unacknowledged. Note that the read exclusive transaction on the system bus guarantees that the requested cache line is returned in an exclusive state. Therefore, the issue of potential updates is disabled through the boot-time mode control.

Coherent Read Conflicts

External coherence requests that conflict with pending processor coherent read requests may be issued to the processor without effecting the processor's behavior. Therefore, in this simple system model no conflict detection is performed by the external agent for processor coherent read requests. If an external intervention request or invalidate request is forwarded to the processor that is in conflict with a pending processor coherent read request, it will not effect the processor's cache since the target cache line is guaranteed to be absent from the cache. The processor effectively discards the conflicting external intervention request, responding with an invalid indication for the target cache line. Similarly, the processor will discard a conflicting external invalidate request since the target cache line is not present and therefore already invalid.

In a system model similar to the one described, conflict detection could be provided for pending processor coherent read requests. In this case, when the external agent sees a read response on the bus that conflicts with a pending processor coherent read request, it does not issue an intervention request to the processor. Rather, it simply reacts as if an intervention request has been completed and the cache line is not present in the processor's cache. Similarly, when the external agent sees an invalidate request on the bus that conflicts with a pending processor coherent read request, it does not forward the invalidate request to the processor since the target cache line is known to be absent from the processor's cache. Using this scheme for conflict detection on processor coherent read requests might slightly reduce the number of external intervention and invalidate requests issued to the processor. However, since the intervention and invalidate requests that would otherwise be issued to the processor would not result in any state modification within the processor, conflict detection for processor coherent read requests is not necessary.

Invalidate Conflicts

From the time the processor has issued an invalidate request until the request has been acknowledged, any external coherence request issued to the processor that is in conflict with the unacknowledged invalidate must include a cancellation. In this system, an acknowledge for the invalidate will be generated to the processor as soon as the invalidate is forwarded to the system bus. Therefore, while the external agent is waiting to acquire mastership of the system bus to forward an invalidate request, the external agent must detect, via comparators, any external coherence request that conflicts with the unacknowledged invalidate. If a conflict is detected, the external agent must not forward the invalidate request to the system bus. Instead, it must throw the invalidate request away and submit the conflicting external request to the processor with a cancellation.

If the response to a coherent read request conflicts with a waiting unacknowledged processor invalidate request appears on the bus, the external agent will detect the conflict and will not forward the processor invalidate request to the bus. Instead, it throws the processor invalidate request away and issues an intervention request to the processor that includes a cancellation. The processor then re-evaluates its cache state and reissues the invalidate request or issues a coherent read request instead.

If an invalidate request appears on the bus while the external agent has a processor invalidate request waiting and the external agent detects a conflict, the external agent will not forward the processor invalidate request to the bus. Instead, it throws the processor invalidate request away and issues an external invalidate request to the processor that includes a cancellation. The processor then re-evaluates its cache state and reissues the invalidate request or issues a coherent read request instead.

It is not possible for a write request to appear on the system bus that conflicts with a waiting processor invalidate request since, for an invalidate request to be issued, the state of the cache line must be shared in every cache in the system in which the line is present.

Coherent Write Conflicts

As soon as a write request has been issued to the external agent the external agent becomes responsible for the cache line. No conflicts are possible with a processor write request; however, the external agent must manage ownership of the cache line while it is waiting to acquire mastership of the system bus so that it may forward the write request. The external agent is responsible for the cache line from the time the issue cycle of the write request is accomplished until the write request is forwarded to the system bus.

If the response to a coherent read request conflicts with a waiting processor write request or with a processor write request that has issued and is transmitting data appears on the bus, the external agent will detect the conflict and will not issue an intervention request to the processor. Instead, it reacts as if an intervention request has been completed and the line is in the *dirty exclusive* state. The external agent indicates *takeover* and supplies the read data to the read requester itself without disturbing the processor. After providing the read data to the read requester, the external agent must throw the write request away if the read request was a read exclusive. In fact, the external agent may throw the write request away for either type of read since processor supplied read data is also written back to memory.

It is not possible for an invalidate request or a write request that conflicts with a waiting processor write request to appear on the system bus, since, for a processor write request to be issued, the state of the cache line must be *dirty exclusive* in that processor's cache.

Secondary Cache Interface

The R4000SC and R4000MC versions of the R4000 interface to an optional secondary cache. The secondary cache interface consists of a 128-bit data bus, a 25-bit tag bus, an 18-bit address bus and SRAM control signals. The 128-bit wide data bus minimizes cache miss penalty, and allows the use of standard low-cost SRAMs in the secondary cache design.

Secondary Cache Interface Signals

Following is a secondary cache interface signal summary.

SCData(127:0): (i/o) A 128-bit bus used to read or write cache data from/to the secondary cache.

SCDChk(15:0): (i/o) A 16-bit bus that carries two 8-bit ECC fields covering the 128 bits of the SCData from/to secondary cache. **SCDChk(15:8)** corresponds to **SCData(127:64)** and **SCDChk(7:0)** corresponds to **SCData(63:0).**

SCTag(24:0): (i/o) A 25-bit bus used to read or write cache tags from/to the secondary cache.

SCTChk(6:0): (i/o) A 7-bit bus that carries an ECC field covering the **SCTag** from/to the secondary cache.

SCAddr(17:1): (o) A 17-bit address bus for the secondary cache.

SCAddr0Z: (o) Bit 0 of the secondary cache address.

SCAddr0Y: (o) Bit 0 of the secondary cache address.

SCAddr0X: (o) Bit 0 of the secondary cache address.

SCAddr0W: (o) Bit 0 of the secondary cache address.

SCAPar(2:0): (o) A 3-bit bus that carries the parity of the SCAddr bus and the cache control lines **SCWR***, **SCDCS*** and **SCTCS***. The individual bit definitions are:
SCAPar2 - Even Parity for **SCAddr(17:12)** and **SCWR***
SCAPar1 - Even Parity for **SCAddr(11:6)** and **SCDCS***
SCAPar0 - Even Parity for **SCAddr(5:0)** and **SCTCS***

SCOE*: (o) Output enable for the secondary cache RAM.

SCWrZ*: (o) Write enable for the secondary cache RAM.

SCWrY*: (o) Write enable for the secondary cache RAM.

SCWrX*: (o) Write enable for the secondary cache RAM.

SCWrW*: (o) Write enable for the secondary cache RAM.

SCDCS*: (o) Chip select enable signal for the secondary cache RAM associated with **SCData** and **SCDChk**.

SCTCS*: (o) Chip select enable signal for the secondary cache RAM associated with **SCTag** and **SCTChk**.

The interface to the secondary cache is designed to maximize the efficiency of servicing primary cache misses. The width of the data portion of the secondary cache interface is 128 bits to support a data rate into the primary cache that is near the processor to primary cache bandwidth during normal operation. To assure that this bandwidth is maintained, each data, tag and check pin must be connected to only one static RAM device. The **SCAddr** bus and the **SCOE***, **SCDCS***, and **SCTCS*** signals drive a large number of static RAM devices; therefore, one level of external buffering between the R4000 and the cache array is necessary.

The speed of the secondary cache interface is limited by buffered control signals. Critical control signals are duplicated to minimize this effect. The **SCWR*** signal and **SCAddr(0)** are duplicated four times so that external buffering will not be required. When an 8-word (256-bit) primary cache line is used, these signals can be controlled more quickly; this reduces the time of the two back-to-back transfers. These duplicated control signals are specified to drive 11 parts each; therefore, a total of 44 RAM packages can be used in the cache array. This permits a cache design using 16 KByte by 64 bit, 64 KByte by 4 bit, or 256 KByte by 4 bit standard static RAMs. Other cache designs are also acceptable. For example, a smaller cache design using twenty-two 8 KByte by 8 bit static RAMs; this design presents less load on the address pins and control signals, and reduces the overall parts count.

Note that duplicated signals like **SCWRW***, **SCWRX***, **SCWRY*** and **SCWRZ*** will be described in this document as though they were a single signal. This signal is called **SCWR***.

The benefit of duplicating **SCAddr(0)** is greater in systems using fast sequential static cache RAMs and a primary cache line size of 8 words. If **SCAddr(0)** is attached to the static RAM address bit that affects column decode only, the read cycle time with respect to that pin should approximate the output enable time of the RAM. For fast static RAM it should be half that of the nominal read cycle time.

When the split instruction/data cache mode is enabled, assertion of the top SCAddr bit, SCAddr(17), enables the instruction half of the cache instead of the data half.

It is possible to design a cache that supports both joint and split instruction/data configurations with less than the maximum cache size. SCAddr(12:0) must be used to address the cache in all configurations. SCAddr(17) must be used to support the split instruction/data configuration. Any of SCAddr(16:14) may be omitted because of the fixed width of the physical tag array.

The SCDChk bus is divided into two fields to cover the upper and lower 64 bits of SCData. This form is required to keep the width of internal data paths to 64 bits.

The SCTag bus is divided into three fields, as shown in Figure 11-6.

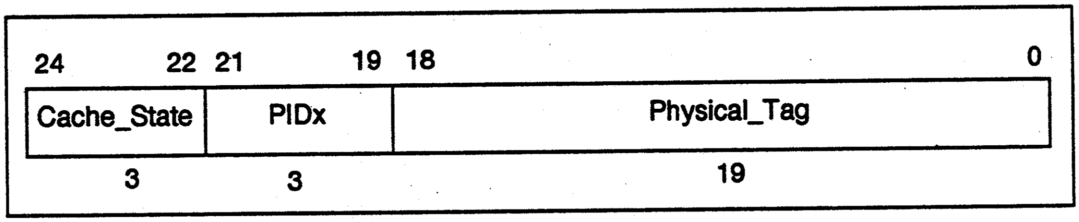


Figure 11-6 SCTag Fields

The SCDCS* and SCTCS* are needed to disable reads or writes of the data array or tag array when the other array is being accessed. These signals are useful for saving power on snoop and invalidate requests because accesses to the data array are not necessary. These signals are also useful for writing data from the data primary cache to the secondary cache because the secondary cache state is not always known by the primary cache.

Operation of the Secondary Cache Interface

The control of the secondary cache is configurable for various clock rates and static RAM speeds. All configurable parameters are specified in multiples of PClock, which runs at twice the frequency of the external system clock, *MasterClock*. Boot time mode control registers hold the various configuration parameters so they can be specified by software when initializing the processor.

Table 11-6 Secondary Cache Timing Parameter

tRd1Cyc:	4-15 PCycles
tRd2Cyc:	3-15 PCycles
tDis:	2-7 PCycles
tWr1Dly:	1-3 PCycles
tWr2Dly:	1-3 PCycles
tWrRC:	0-1 PCycles
tWrSUp:	3-15 PCycles

Read Cycles

Each secondary cache read sequence begins with the driving of the address pins. The output enable signal SCOE* is asserted at the same time.

There are two basic read cycles: a four-word read, and an eight-word read.

For the four-word read, there are two parameters of interest. The first parameter is read sequence cycle time, T_{Rd1Cyc}, which specifies the time from the driving of the SCAddr bus to the sampling of the SCData bus. The second parameter is the cache output disable time T_{Dis}, which specifies the time from the end of a read cycle to the start of the next write cycle. Figure 11-7 illustrates the four word read sequence.

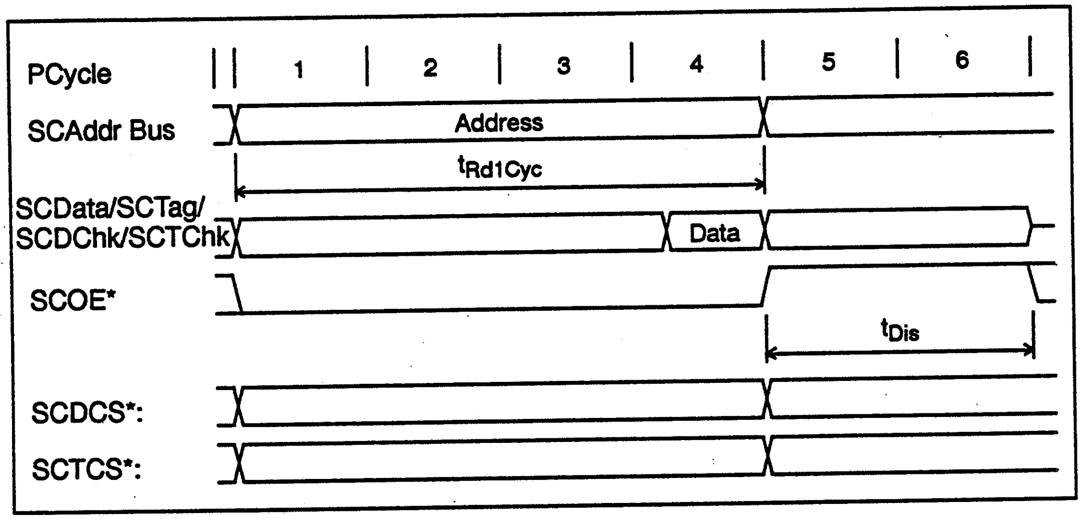


Figure 11-7 Four-Word Read Cycle

For the eight-word read, there is one additional parameter of interest: the time from the first sample point to the second sample point, T_{Rd2Cyc}. The lower-order address bit, **SCAddr(0)** is changed at the same time as the first read sample point. Figure 11-8 illustrates the eight word read sequence.

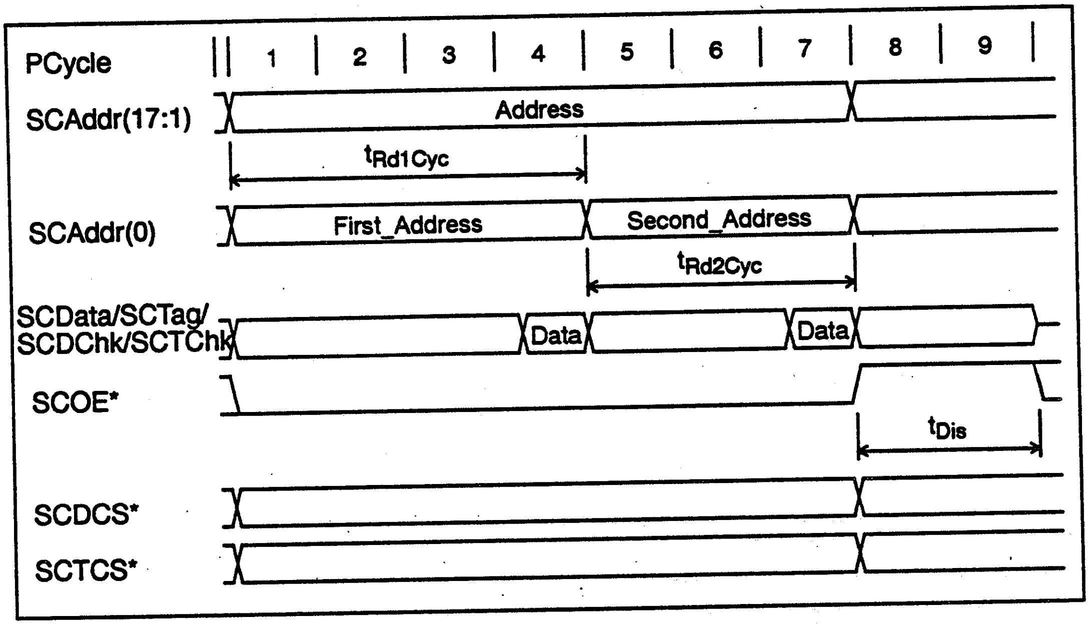


Figure 11-8 Eight-Word Read Cycle

All read cycles can be aborted by changing the address. A new cycle starts with the edge on which the address is changed. Additionally, the period t_{Dis} after a read cycle can be interrupted any time by the start of a new read cycle. If a read cycle is aborted by a write cycle, SCOE* must be deasserted for the t_{Dis} period, before the write cycle can commence. Read cycles can also be extended indefinitely. There is no requirement to change the address at the end of a read cycle.

Write Cycles

Like the read sequence, the secondary cache write sequence begins with the driving of the address pins.

There are two basic write cycles: a four-word write cycle and an eight-word write cycle.

For the four-word write, there are several parameters of interest.

T_{Wr1Dly} delay from the driving of the address to the assertion of SCWR*.

T_{WrSUp} delay from driving the second data double-word to the deassertion of SCWR*.

T_{WrRc} delay from the deassertion of SCWR* to the beginning of the next cycle.

T_{WrRc} will be zero for most cache designs. Note that the upper data double word and the lower data double word will normally be driven one cycle apart. This reduces the peak current consumption in the output drivers. Figure 11-9 illustrates the four word write sequence. The order of driving the upper versus the lower half on SCData is not fixed; either the upper or lower half may be driven first.

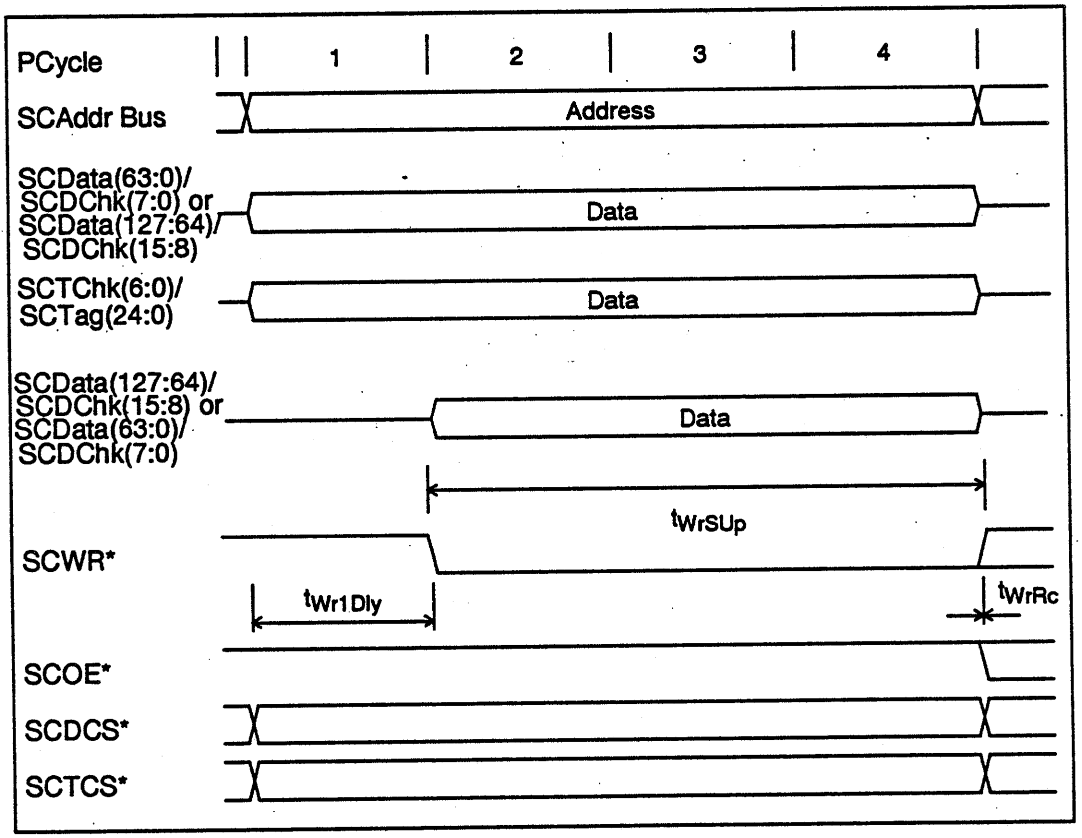

PCycle
1
2
3
4
SCAddr Bus
Address
SCData(63:0)/ SCDChk(7:0) or SCData(127:64)/ SCDChk(15:8)
Data
SCTChk(6:0)/ SCTag(24:0)
Data
SCData(127:64)/ SCDChk(15:8) or SCData(63:0)/ SCDChk(7:0)
Data
tWrSUp
SCWR*
tWr1Dly
tWrRc
SCOE*
SCDCS*
SCTCS*

Figure 11-9 Four-Word Write Cycle

The eight word write has one additional parameter: T_{Wr2Dly}. This is the time from changing the low-order address bit **SCAddr(0)** to the assertion of **SCWR*** for the second time. The lower half of SC Data will be driven out on the same edge as the change in **SCAddr(0)**. Figure 11-10 illustrates the eight word write sequence.

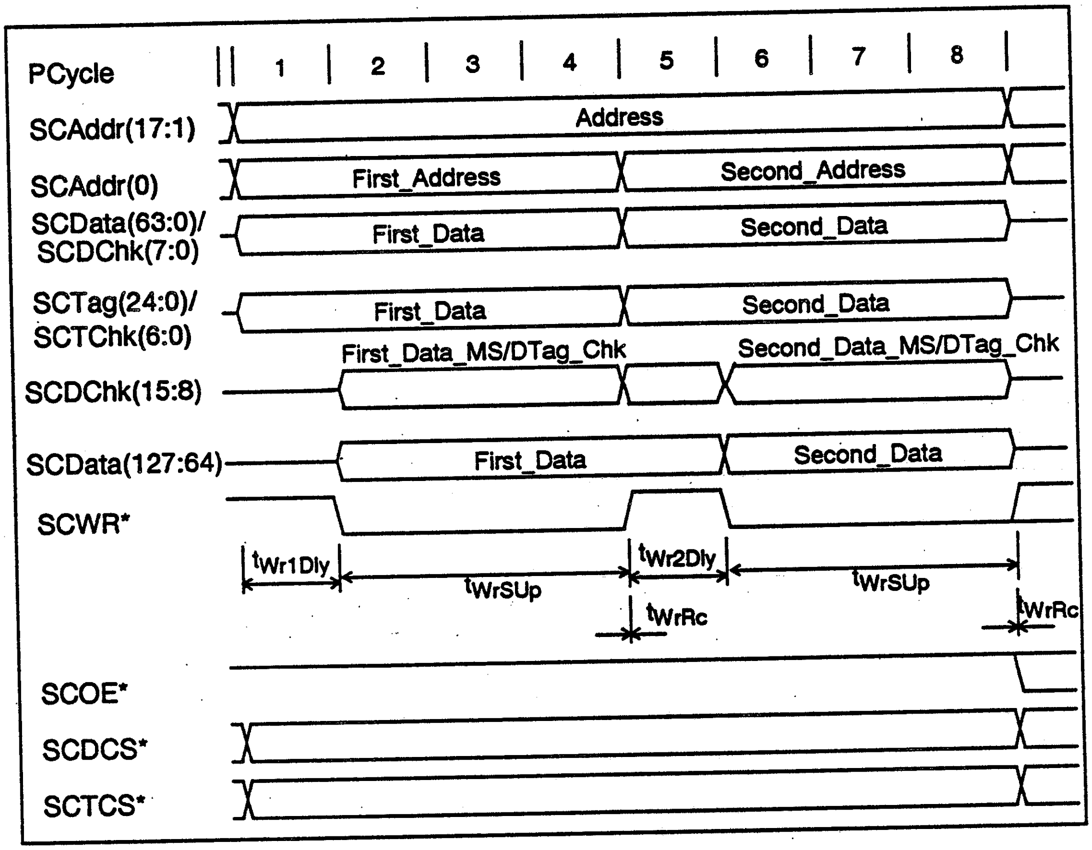


Figure 11-10 Eight-Word Write Cycle

When receiving data from the system interface, it is possible that the first data double word will arrive several cycles before the second data double word. In this case, the cache state machine simply waits in a state that extends the **SCWR*** until T_{WrSUp} after the driving of the second data item.

Initialization Interface

12

Functional Overview

The operation of the R4000 requires a multilevel reset sequence using the **VCCOk**, **ColdReset***, and **Reset*** inputs. A power-on or cold reset accomplish the same thing: they both completely reset the internal state machine of the R4000. A warm reset also resets the internal state machine; however, the processor internal state is preserved.

Fundamental operational modes for the processor are initialized by the initialization interface. The initialization interface is a serial interface operating at a frequency of MasterClock divided by 256. The low-frequency operation allows the initialization information to be stored in a low-cost EPROM.

Immediately after the **VCCOk** signal is asserted, the processor reads a serial bit stream of 256 bits on **ModeIn** to initialize all fundamental operational modes. After initialization is complete, the processor continues to drive the serial clock output, but no further initialization bits are read.

Initialization Interface Operation

1. While **VCCOk** is de-asserted, the **ModeClock** output is held asserted.
2. The processor synchronizes the **ModeClock** output at the time **VCCOk** is asserted, and the first rising edge of the **ModeClock** will occur 256 **MasterClock** clock cycles after **VCCOk** is asserted.

3. After each rising edge of the **ModeClock**, the next bit of the initialization bit stream must be presented at the **ModeIn** input. The processor will sample exactly 256 initialization bits from the **ModeIn** input.

ModeIn: (i) Serial boot mode data in.

ModeClock: (o) Serial boot mode data clock out at the **MasterClock** frequency divided by 256.

Refer to Figure 12-1 and Figure 12-2 for timing relationships.

Boot-Time Modes

The correspondence between bits of the initialization bit stream and processor mode settings is illustrated in Table 12-1. Bit 0 of the bit stream is the bit presented to the processor when **VCCOk** is asserted.

Table 12-1 Boot Time Modes

Serial Bit	Value	Mode Setting
0	**BlkOrder:** Secondary Cache Mode block read response ordering.	
	0	Sequential ordering.
	1	Sub-block ordering.
1	**EIBParMode:** Specifies nature of system interface check bus.	
	0	SECDED error checking and correcting mode.
	1	Byte parity.
2	**EndBIt:** Specifies byte ordering.	
	0	Little Endian ordering.
	1	Big Endian ordering.
3	**DShMdDis:** Dirty shared mode, enables transition to dirty shared state on processor update successful.	
	0	Dirty shared mode enabled.
	1	Dirty shared mode disabled.
4	**NoSCMode:** Specifies presence of secondary cache.	
	0	Secondary cache present.
	1	No secondary cache present.
5:6	**SysPort:** System Interface port width, bit 6 most significant.	
	0	64 bits.
	1-3	Reserved.
7	**SC64BitMd:** Secondary cache interface port width.	
	0	128 bits.
	1	Reserved.
8	**EISpltMd:** Specifies secondary cache organization	
	0	Secondary cache unified.
	1	Reserved.
9:10	**SCBlkSz:** Secondary cache line size, bit 10 most significant.	
	0	4 words.
	1	8 words.
	2	16 words.
	3	32 words.

Table 12-2 Boot Time Modes

Serial Bit	Value	Mode Setting
11:14	**XmitDatPat:** System interface data rate, bit 14 most significant.	
	0	D
	1	DDx
	2	DDxx
	3	DxDx
	4	DDxxx
	5	DDxxxx
	6	DxxDxx
	7	DDxxxxxx
	8	DxxxDxxx
	9-15	Reserved.
15:17	**SysCkRatio:** PClock to SClock divisor, frequency relationship between SClock, RClock, and TClock and PClock, bit 17 most significant.	
	0	Divide by 2.
	1	Divide by 3.
	2	Divide by 4.
	3-7	Reserved.
18	0	Reserved.
19	**TimIntDis:** Timer Interrupt enable allows timer interrupts, otherwise the interrupt used by the timer becomes a general-purpose interrupt.	
	0	Timer Interrupt enabled.
	1	Timer Interrupt disabled.
20	**PotUpdDis:** Potential update enable allows potential updates to be issued. Otherwise only compulsory updates are issued.	
	0	Potential updates enabled.
	1	Potential updates disabled.
21:24	**TWrSUp:** Secondary cache write deassertion delay, TWrSup in PCycles, bit 24 most significant.	
	0-2	Undefined.
	3-15	Number of PCLK cycles; Min 3; Max 15.
25:26	**TWr2Dly:** Secondary cache write assertion delay 2, TWr2Dly in PCycles, bit 26 most significant.	
	0	Undefined.
	1-3	Number of PCLK cycles; Min 1, Max 3

Table 12-3 Boot Time Modes

Serial Bit	Value	Mode Setting
27:28	**TWr1Dly:** Secondary cache write assertion delay 1, TWr1Dly in PCycles, bit 28 most significant.	
	0	Undefined.
	1-3	Number of PCLK cycles; Min 1, Max 3
29	**TWrRc:** Secondary cache write recovery time, TWrRc in PCycles, either 0 or 1 cycles.	
	0	0 cycle
	1	1 cycle
30:32	**TDis:** Secondary cache disable time, TDis in PCycles, bit 32 most significant.	
	0-1	Undefined.
	2-7	Number of PCLK cycles; Min 2, Max 7
33:36	**TRd2Cyc:** Secondary cache read cycle time 2, TRdCyc2 in PCycles, bit 36 most significant.	
	0-2	Undefined.
	3-15	Number of PCLK cycles; Min 3; Max 15.
37:40	**TRd1Cyc:** Secondary cache read cycle time 1, TRdCyc1 in PCycles, bit 40 most significant.	
	0-3	Undefined.
	4-15	Number of PCLK cycles; Min 4; Max 15.
41:45	0	Reserved.
46	**Pkg179:** R4000 Package type.	
	0	Large (447 pin).
	1	Small (179 pin).
47:49	**CycDivisor:** This mode determines the clock divisor for the reduced power mode. When the RP bit in the Status Register is set to one, the pipeline clock is divided by one of the following values. Bit 49 is most significant.	
	0	Divide by 2
	1	Divide by 4
	2	Divide by 8
	3	Divide by 16
	4-7	Reserved.
50:52	**Drv0_50, Drv0_75, Drv1_00:** Drive the outputs out in NxMasterClock period. Bit 52 most significant. Those combinations not defined below are reserved.	
	1	Drive at 0.50 x MasterClock period.
	2	Drive at 0.75 x MasterClock period.
	4	Drive at 1.00 x MasterClock period.

Table 12-4 Boot Time Modes

Serial Bit	Value	Mode Setting
53:56	**InitP:** Initial values for the state bits that determine the pull-down Δi/Δt and switching speed of the output buffers. Bit 53 is the most significant.	
	0	Fastest pull-down rate
	1-14	Intermediate pull-down rates.
	15	Slowest pull-down rate.
57:60	**InitN:** Initial values for the state bits that determine the pull-up Δi/Δt and switching speed of the output buffers. Bit 57 is the most significant.	
	0	Slowest pull-up rate
	1-14	Intermediate pull-up rates.
	15	Fastest pull-up rate.
61	**EnblDPLLR:** Enables the negative feedback loop that determines the Δi/Δt and switching speed of the output buffers only during ColdReset.	
	0	Disable Δi/Δt mechanism.
	1	Enable Δi/Δt mechanism.
62	**EnblDPLL:** Enables the negative feedback loop that determines the Δi/Δt and switching speed of the output buffers during ColdReset and during normal operation.	
	0	Disable Δi/Δt control mechanism.
	1	Enable Δi/Δt control mechanism.
63	**DsblPLL:** Enables PLLs that match MasterIn and produce RClock, TClock SClock and the internal clocks.	
	0	Enable PLLs.
	1	Disable PLLs.
64	**SRTristate:** Controls when output-only pins are tristated	
	0	Only when ColdReset* is asserted.
	1	When Reset* or ColdReset* are asserted
65:255	Reserved. Scan in zeros.	

- Selecting a reserved value results in undefined processor behavior.
- Bits 65 to 255 are reserved bits.
- Zeros must be scanned in for all reserved bits.

Reset Operation

The R4000 supports three types of resets:

- Power-on Reset: Starts from power supply turning on.
- Cold Reset: Restarts all clocks, but power supply remains stable. Processor operating parameters do not change.
- Warm Reset: Restarts processor, but does not affect clocks.

The operation of each type of reset is described in a subsection below.

Reset Signal Summary

VCCOk: When asserted, VCCOk indicates to the R4000 that the +5 volt power supply has been above 4.75 volts for more than 100 milliseconds and will remain stable. The assertion of VCCOk initiates the reading of the boot-time mode control serial stream.

ColdReset*: ColdReset* must be asserted for a power on reset or a cold reset. The clocks SClock, TClock, and RClock begin to cycle and are synchronized with the de-assertion edge of ColdReset*. ColdReset* must be deasserted synchronously with MasterClock.

Reset*: Reset* must be asserted for any reset sequence. It may be asserted synchronously or asynchronously for a cold reset, or synchronously to initiate a warm reset. Reset* must be deasserted synchronously with MasterClock.

Power-on Reset

The sequence for a power-on reset is:

1. Stable VCC of at least 4.75 volts from the +5 volt power supply is applied to the processor. A stable continuous system clock at the processor's desired operational frequency is also supplied.

2. After at least 100 milliseconds of stable VCC and MasterClock, the VCCOk input to the processor may be asserted. The assertion of VCCOk causes the processor to initialize the 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, the SyncOut-SyncIn clock path and the master clock output, MasterOut.

3. Once the boot-time mode control serial data stream had been read by the processor, the **ColdReset*** input may be de-asserted. **ColdReset*** must remain asserted for at least 64 MasterClock cycles after the assertion of **VCCOk**. **ColdReset*** must be de-asserted synchronously with MasterClock.
4. The de-assertion edge of **ColdReset*** is used to synchronize the edges of SClock, TClock, and RClock, potentially across multiple processors in a multiprocessor system.
5. After **ColdReset*** is de-asserted and SClock, TClock and RClock have stabilized, **Reset*** is de-asserted to allow the processor to begin to run. **Reset*** must be held asserted for at least 64 MasterClock cycles after the de-assertion of **ColdReset***. **Reset*** must be de-asserted synchronously with MasterClock.

ColdReset* must be asserted when **VCCOk** asserts. The behavior of the processor is undefined if **VCCOk** asserts while **ColdReset*** is de-asserted.

Cold Reset

A cold reset can begin once 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 power-on reset except that the power is presumed to have been stable before the assertion of the reset inputs and the deassertion of **VCCOk**. **VCCOk** must be deasserted for a minimum of 100 msec before reassertion, to begin the reset sequence.

Warm Reset

To affect a warm reset, the **Reset*** input may be asserted synchronously with MasterClock and held asserted for at least 64 MasterClock cycles before being de-asserted synchronously with MasterClock. The processor internal clocks, PClock and SClock, and the system interface clocks, TClock and RClock ,are not be affected by a warm reset, and the boot-time mode control serial data stream is not read by the processor on a warm reset. A Warm Reset causes processor to start with the Soft Reset Exception

The master clock output, **MasterOut**, is provided for use in generating the reset related signals for the processor that must be synchronous with **MasterClock**.

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 will depend on whether a cache miss sequence has been interrupted by resetting the processor state machines.

The following timing diagrams illustrate a power-on reset, cold reset, and warm reset.

Figure 12-1 Power-On Reset

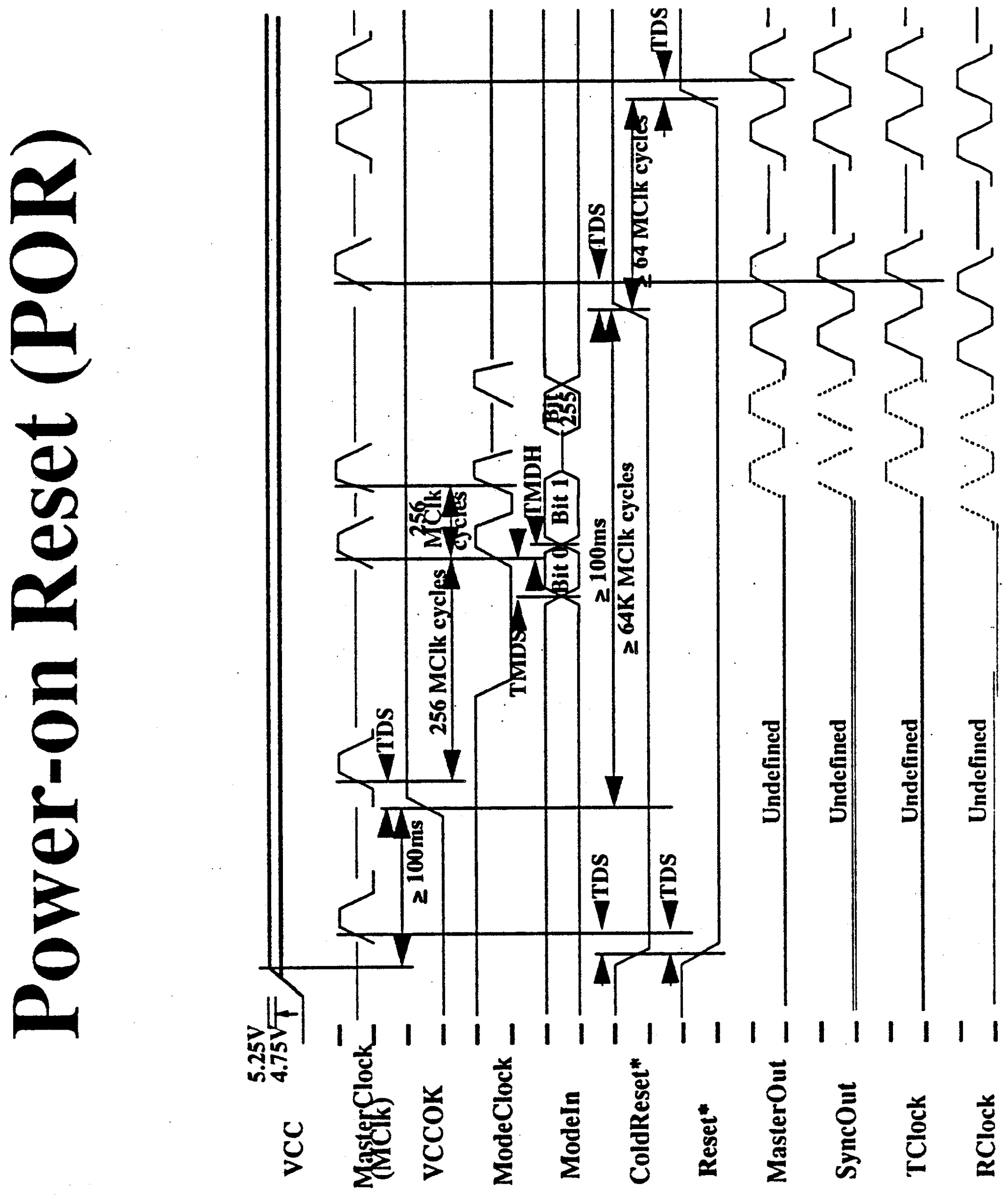


Figure 12-2 Cold Reset

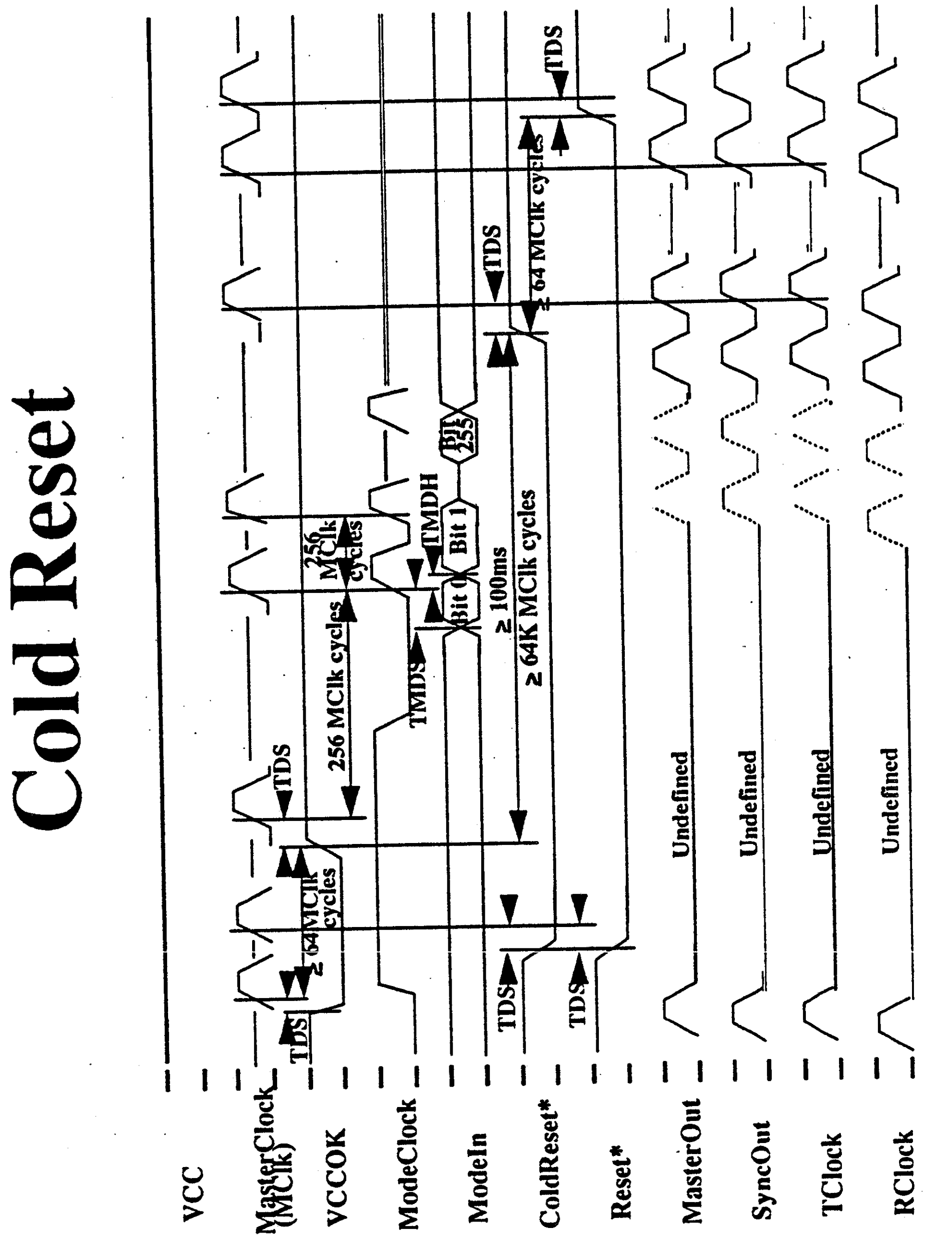
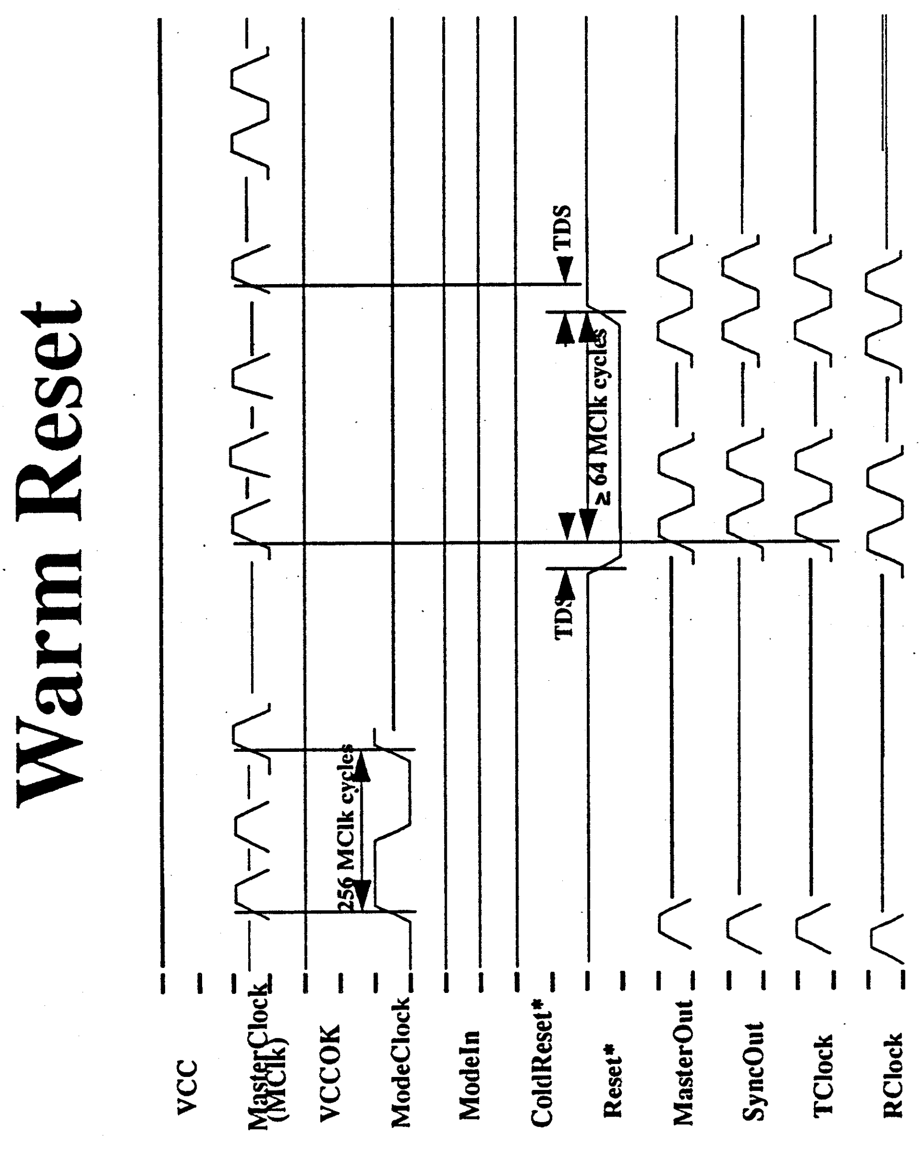


Figure 12-3 Warm Reset



JTAG Interface

13

The R4000 processor provides a boundary scan interface using the industry standard JTAG protocol.

JTAG Interface Signal Summary

JTDI:	(i) JTAG serial data in.
JTDO:	(o) JTAG serial data out.
JTMS:	(i) JTAG command signal.
JTCK:	(i) JTAG serial clock input.

JTAG Functionality

The JTAG boundary scan mechanism provides a capability for testing the interconnect between the R4000 processor, the printed circuit board to which it is attached, and the other components on the board. In addition the JTAG boundary scan mechanism provides a rudimentary capability for low-speed logical testing of the secondary cache RAMs. The JTAG boundary scan mechanism does not provide any capability for testing the R4000 processor itself.

The JTAG boundary scan mechanism is compatible with JTAG specifications. The R4000 processor contains the JTAG registers—TAP controller, JTAG Instruction Register, JTAG Boundary Scan Register and a JTAG Bypass Register—and executes the standard JTAG EXTEST operation associated with external test functionality testing.

JTAG Test Access Port (TAP)

The JTAG Test Access Port (TAP) consists of the 4 pins described above. Data is serially scanned into one of the three registers (Instruction Register, Bypass Register, Boundary San Register) from the JTDI pin, and is scanned out from the selected one of these registers onto the JTDO pin. The JTDI input feeds the LSB of the selected register, and the MSB of the selected register appears on the JTDO output. The JTMS input controls the state transitions of the main TAP controller state machine.

Data on the JTDI and JTMS pins is sampled on the rising edge of the JTCK input clock signal. Data on the JTDO pin changes on the falling edge of the JTCK clock signal.

JTAG TAP Controller

The R4000 implements the 16-state JTAG TAP controller as defined in the IEEE JTAG specification.

The TAP controller state machine can be put in its Reset state in one of two ways. Deassertion of the VCCOk input will reset the TAP controller. Keeping the JTMS input signal asserted through five consecutive rising edges of the JTCK clock input will also send the TAP controller state machine into its Reset state. In either case, keeping JTMS asserted will maintain the Reset state.

Instruction Register

The R4000's JTAG Instruction Register is three bits wide and is encoded as follows:

MSB . . . LSB	Selected Data Register
0 0 0	Boundary Scan Register (external test only)
x x 1	Bypass Register
x 1 x	Bypass Register
1 x x	Bypass Register

The instruction register comprises two stages; the shift register stage and the parallel output latch. When the TAP controller is in the Reset state, the value 7 (111) is loaded into the parallel output latch, thus selecting the Bypass Register as the default. When the TAP controller is in the Capture-IR state, the value 4 (100) is loaded into the shift register stage. When the TAP controller is in the Shift-IR state, data is serially shifted into the shift register stage of the Instruction Register from the **JTDI** input pin, and the MSB of the Instruction Register's shift register state is shifted out onto the **JTDO** pin. When the TAP controller is in the Update-IR state, the current data in the shift register stage is loaded into the parallel output latch.

Bypass Register

The Bypass Register is one bit wide. When the TAP controller is in the Shift-DR (Bypass) state, the data on the **JTDI** pin is shifted into the Bypass Register, and the Bypass Register's output is shifted out onto the **JTDO** output pin.

Boundary Scan Register

The Boundary Scan Register is 319 bits wide. The three most-significant bits control the output enables on the various bidirectional buses. The most-significant bit is the JTAG output enable bit for the **SysAD**, **SysADC**, **SysCmd**, and **SysCmdP** buses. The next most significant bit is the JTAG output enable for the **SCData** and **SCDChk** buses. The third most-significant bit is the JTAG output enable for the **SCTag** and **SCTChk** buses. The remaining 316 bits correspond to 316 signal pads of the R4000.

The scan order of these 316 scan bits is listed below starting from JTDI and ending with JTDO.

1. SCDChk[13]
2. SysADC[1]
3. SCDChk[1]
4. SysADC[5]
5. SCDChk[5]
6. Status[0]
7. Status[1]
8. Status[2]
9. Status[3]
10. IvdErrB
11. Status[4]
12. IvdAckB
13. Status[5]
14. Status[6]
15. Status[7]
16. SCDChk[7]
17. SysADC[7]
18. SCDChk[3]
19. SysADC[3]
20. SCDChk[15]
21. VCCOk
22. SCTag[16]
23. SCDChk[11]
24. SCData[63]
25. SysAD[63]
26. SCData[31]
27. SysAD[31]
28. SCData[127]
29. SCTag[17]
30. SCData[95]
31. SCData[62]
32. SysAD[62]
33. SCData[30]
34. SysAD[30]
35. SCData[126]
36. SCTag[18]
37. SCData[94]
38. RClock[1..0] (share the same JTAG bit)
39. SCTag[19]
40. SCData[61
41. SysAD[61]
42. SCData[29]
43. SysAD[29]
44. SCData[125]
45. ResetB
46. SCTag[20]
47. SCData[93]
48. SCData[60]
49. SysAD[60]
50. SCData[28]
51. SysAD[28]
52. SCData[124]
53. ColdResetB
54. SCTag[21]
55. SCData[92]
56. SCData[59]
57. SysAD[59]
58. SCData[27]
59. SysAD[27]
60. SCData[123]
61. IOIn
62. SCTag[22]
63. SCData[91]
64. SCData[58]
65. SysAD[58]
66. SCData[26]
67. SysAD[26]
68. SCData[122]
69. IOOut
70. SCTag[23]
71. SCData[90]
72. SCData[57]
73. SysAD[57]
74. SCData[25]
75. SysAD[25]
76. SCData[121]
77. GrpRunB
78. SCTag[24]
79. SCData[89]
80. SCData[56]
81. SysAD[56]
82. SCData[24]
83. SysAD[24]
84. SCData[120]
85. GrpStallB
86. SCTChk[0]
87. SCData[88]
88. SCDChk[6]
89. SysADC[6]
90. SCDChk[2]
91. SysADC[2]
92. SCDChk[14]
93. NMIB
94. SCTChk[1]
95. SCDChk[10]
96. SCData[55]
97. SysAD[55]
98. SCData[23]
99. SysAD[23]
100. SCData[119]
101. ReleaseB
102. SCTChk[2]
103. SCData[87]
104. SCData[54]
105. SysAD[54]
106. SysAD[22]
107. ModeIn
108. SCData[22]
109. RdRdyB
110. SCData[118]
111. SCData[86]
112. SCData[53]
113. SysAD[53]
114. SCData[21]
115. SysAD[21]
116. SCData[117]
117. ExtRqstB
118. SCTChk[3]
119. SCData[85]
120. SCData[52]
121. SysAD[52]
122. SCData[20]
123. SysAD[20]
124. SCData[116]
125. ValidOutB
126. SCTChk[4]
127. SCData[84]
128. SCData[51]
129. SysAD[51]
130. SCData[19]
131. SysAD[19]
132. SCData[115]
133. ValidInB
134. SCTChk[5]
135. SCData[83]
136. SCAddr0W,X (share the same JTAG bit)
137. SCAddr0Y,Z (share the same JTAG bit)
138. SCAddr[1]
139. SCData[50]
140. SysAD[50]
141. SCData[18]
142. SysAD[18]
143. SCData[114]
144. IntB[0]
145. SCTChk[6]
146. SCData[82]
147. SCData[49]
148. SysAD[49]
149. SCData[17]
150. SysAD[17]
151. SCData[113]
152. SCAddr[2]IntB[1]
153. SCAddr[3]
154. SCData[81]
155. SCData[48]
156. SysAD[48]
157. SCData[16]
158. SysAD[16]
159. SCData[112]
160. SCAddr[4]IntB[2]
161. SCAddr[5]
162. SCData[80]
163. SCAddr[6]
164. SCAddr[7]
165. SCAddr[8]
166. SCAddr[9]
167. SCAddr[10]
168. SCAddr[11]
169. SC64Addr
170. SCAddr[12]
171. SCAddr[13]
172. SCAddr[14]
173. SCAddr[15]
174. SCAddr[16]
175. SCAddr[17]
176. SCData[64]
177. SCAPar[0]
178. SCAPar[1]/IntB[3]
179. SCData[96]
180. SysAD[0]
181. SCData[0]
182. SysAD[32]
183. SCData[32]
184. SCData[65]
185. SCAPar[2]
186. SCOEB/IntB[4]
187. SCData[97]
188. SysAD[1]
189. SCData[1]
190. SysAD[33]
191. SCData[33]
192. SCData[66]

193. SCDCSB
194. SCTCSB/IntB[5]
195. SCData[98]
196. SysAD[2]
197. SCData[2]
198. SysAD[34]
199. SCData[34]
200. SCTag[0]
201. SCWrBW,X (share the same JTAG bit)
202. SCWrBY,Z (share the same JTAG bit)
203. SCData[67]
204. SCTag[1]
205. SysCmd[0]
206. SCData[99]
207. SysAD[3]
208. SCData[3]
209. SysAD[35]
210. SCData[35]
211. SCData[68]
212. SCTag[2]
213. SysCmd[1]
214. SCData[100]
215. SysAD[4]
216. SCData[4]
217. SysAD[36]
218. SCData[36]
219. SCData[69]
220. SCTag[3]
221. SysCmd[2]
222. SCData[101]
223. SysAD[5]
224. SCData[5]
225. SysAD[37]
226. SCData[37]
227. SCData[70]
228. WrRdyB
229. ModeClock
230. SCData[102]
231. SysAD[6]
232. SCData[6]
233. SysAD[38]
234. SCData[38]
235. SCData[71]
236. SCTag[4]
237. SysCmd[3]
238. SCData[103]
239. SysAD[7]
240. SCData[7]
241. SysAD[39]
242. SCData[39]
243. SCDChk[8]
244. SCTag[5]
245. SysCmd[4]
246. SCDChk[12]
247. SysADC[0]
248. SCDChk[0]
249. SysADC[4]
250. SCDChk[4]
251. SCData[72]
252. SCTag[6]
253. SysCmd[5]
254. SCData[104]
255. SysAD[8]
256. SCData[8]
257. SysAD[40]
258. SCData[40]
259. SCData[73]
260. SCTag[7]
261. SysCmd[6]
262. SCData[105]
263. SysAD[9]
264. SCData[9]
265. SysAD[41]
266. SCData[41]
267. SCData[74]
268. SCTag[8]
269. SysCmd[7]
270. SCData[106]
271. SysAD[10]
272. SCData[10]
273. SysAD[42]
274. SCData[42]
275. SCData[75]
276. SCTag[9]
277. SysCmd[8]
278. SCData[107]
279. SysAD[11]
280. SCData[11]
281. SysAD[43]
282. SCData[43]
283. SCData[76]
284. SCTag[10]
285. SysCmdP
286. SCData[108]
287. SysAD[12]
288. SCData[12]
289. SysAD[44]
290. SCData[44]
291. SCData[77]
292. SCTag[11]
293. FaultB
294. SCData[109]
295. SysAD[13]
296. SCData[13]
297. SysAD[45]
298. SCData[45]
299. SCTag[12]
300. TClock[1..0] (share the same JTAG bit)
301. SCData[78]
302. SCTag[13]
303. SCData[110]
304. SysAD[14]
305. SCData[14]
306. SysAD[46]
307. SCData[46]
308. SCData[79]
309. SCTag[14]
310. SCData[111]
311. SysAD[15]
312. SCData[15]
313. SysAD[47]
314. SCData[47]
315. SCDChk[9]
316. SCTag[15]
317. SCTag_OE (JTAG output enable control for SCTag and SCTChk buses)
318. SCData_OE (JTAG output enable control for SCData and SCDChk buses)
319. SysAD_OE (JTAG output enable control for SysAD, SysADC, SysCmd and SysCmdP buses)

When the TAP controller is in the Reset state, the three most significant bits of the Boundary Scan Register are set to "0" (the default JTAG output enable control on all the bidirectional pins is to disable the outputs). When the TAP controller is in the Capture-DR (Boundary Scan) state, the data currently present on all the R4000's input and I/O pins are latched into the Boundary Scan Register. The Boundary Scan Register bits corresponding to output pins are arbitrary in this state and must not be checked during the scan out process. When the TAP controller is in the Shift-DR (Boundary Scan) state, data is serially shifted into the Boundary Scan Register from the **JTDI** pin, and the contents of the Boundary Scan Register are shifted out onto the **JTDO** pin. When the TAP controller is in the Update-DR (Boundary Scan) state, the current data in the Boundary Scan Register is latched into its parallel output latch, and the bits corresponding to output pins and those I/O pins whose outputs are enabled (by the three MSBs of the Boundary Scan Register) are enabled onto the R4000's pins.

Implementation Specific Details

- The **MasterClock, MasterOut, SyncIn,** and **SyncOut** pads do not have JTAG.
- Some pairs of output pads share a single JTAG bit. These are:

  SCAddr0W and **SCAddr0X**
  SCAddr0Y and **SCAddr0Z**
  SCWrBW and **SCWrBX**
  SCWrBY and **SCWrBZ**
  TClcok[0] and **TClock[1]**
  RClock[0] and **RClock[1]**

- All input pads data are first latched into a Processor Clock-based register in the pad cell before they are captured into the Boundary Scan Register in the Capture-DR (Boundary Scan) state. When the phase-locked loop is disabled, the processor clock is half the frequency of **MasterClock**. Therefore, the data setup required at the input pads is greater than two **MasterClock** periods before the rising edge of the **JTCK** when the TAP controller is in the Capture-DR (Boundary Scan) sate.
- The output enable controls generated from the three most significant bits of the Boundary Scan Register are latched into a Processor Clock-based register before they actually enable the data onto the pads. Therefore, the delay from the rising edge of **JTCK** in the Update-DR (Boundary Scan) state to data valid at the output pins of the chip is greater than two **MasterClock** periods.

Processor Interrupts

14

The R4000 processor supports six hardware interrupts, two software interrupts, and a non-maskable interrupt. The processor's six hardware interrupts are accessible via external write requests in the R4000SC, R4000MC and R4000PC configurations, and by dedicated pins as well in the R4000PC configuration. The non-maskable interrupt is accessible via external write requests and a dedicated pin in the R4000SC, R4000MC and R4000PC configurations.

External writes to the processor are directed, based on a processor internal address map, to various processor internal resources. An external write to any address with **SysAD[6..4]** = 0 writes to an architecturally transparent register called the *Interrupt* Register. During the data cycle, **SysAD[22..16]** are the write enables for the 6 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 set and clear with a single write request.

- In the R4000SC and R4000MC, bit 5 of the *Interrupt* register is multiplexed with the TimerInterrupt signal and the result is directly readable as bit 15 of the *Cause* register. Bits 4:1 of the Interrupt register are directly readable as bits 14:11 of the *Cause* register. Bit 0 of the *Interrupt* register is ORed with the **Int*[0]** pins, and the result is directly readable as bit 10 of the *Cause* register.
- In the R4000PC, bit 5 of the *Interrupt* register is ORed with the **Int*[5]** pin and then multiplexed with the TimerInterrupt signal and 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 the interrupt pins **Int*[4:0]** and the result is directly readable as bits 14:10 of the *Cause* register.

- In both configurations, bit 6 of the Interrupt Register is ORed with the inverted value of the non-maskable interrupt pin **NMI***** to form the non-maskable interrupt input to the processor.

Error Checking and Correcting

15

The processor provides sixteen check bits for the secondary cache data bus **SCDChk(15:0)**, seven check bits for the secondary cache tag bus **SCTChk(6:0)**, and eight check bits for the system interface address and data bus **SysADC(7:0)**. The sixteen check bits for the secondary cache data bus are organized as eight check bits for the upper sixty-four bits of the data bus, and eight check bits for the lower sixty-four bits of the data bus. In addition, a single check bit is provided for the system interface command bus **SysCmdP**.

The eight check bits for the system interface address and data bus provide either even-byte parity or are generated in accordance with a single error correcting double error detecting (SECDED) code that also detects any three or four bit error in a nibble. (See Appendix C for details.) The eight check bits for each half of the secondary cache data bus are always generated in accordance with the SECDED code.

The processor checks data using parity or the SECDED code as it passes from the system interface to the secondary cache and as it is moved from the secondary cache to the primary cache or to the system interface. The processor passes the check bits for data accessed from the secondary cache directly to the system interface without change as it checks it. The processor does not check data received from the system interface for external updates and external writes. It is possible to force the processor to not check data from the system interface for read responses using a bit in the data identifier. The processor does generate correct check bits for double word, word, or partial word data transmitted to the system interface. The processor does not check addresses received from the system interface, but does generate correct 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 an error is detected based on the data check bits. Software, in conjunction with an off-processor data corrector, is responsible for correcting the data when the SECDED code is employed.

The seven check bits for the secondary cache tag bus are generated in accordance with a single error correcting double error detecting (SECDED) code that also detects any three or four bit error in a nibble. The processor generates check bits for the tag when it is written into the secondary cache and checks the tag whenever the secondary cache is accessed. The processor contains a corrector for the secondary cache tag. The tag corrector is not in-line for processor accesses due to primary cache misses. When a tag error is detected on a processor access due to a primary cache miss, the processor will trap. Software, using the processor cache management primitives, corrects the tag. When executing the cache management primitives, the processor uses the corrected tag to generate write back addresses and cache state. For external accesses, the tag corrector is in-line; that is, the response to external accesses is based on the corrected tag. The processor still traps on tag errors detected during external accesses to allow software to repair the contents of the cache if possible.

The check bit for the system interface command bus provides even parity over the nine bits of the system interface command bus. This parity bit is generated correctly when the system interface is in master state, but is not checked when the system interface is in slave state.

The busses that are covered by check bits, their contents, and whether or not they are checked for various processor internal and external transactions are summarized in Table 15-1, Table 15-2, Table 15-3, and Table 15-4.

Table 15-1 Error Checking and Correcting Summary for Internal Transactions

Bus	Secondary Cache to Primary Cache	Primary Cache to Secondary Cache	Uncached Load	Uncached Store
Processor or Secondary Cache Data	Checked, Trap on Error	Primary Cache parity	From System Interface	Not Checked
Secondary Cache Data Check Bits	Checked, Trap on Error	Generated	NA	NA
Secondary Cache Tag and Check Bits	Checked, not corrected Trap on error	Generated	NA	NA
System Internal Addr/ Cmd and Check Bits: Transmit	NA	NA	Generated	Generated
System Internal Addr/ Cmd and Check Bits: Receive	NA	NA	Not Checked	NA
System Internal Data	NA	NA	Not Checked	From Processor
System Internal Data Check Bits	NA	NA	Not Checked	Generated

Table 15-2 Error Checking and Correcting Summary for Internal Transactions

Bus	Store to Shared Cache Line	Cache Instruction	Secondary Cache Load from System Int	Secondary Cache Write to System Int
Processor or Secondary Cache Data	Checked on read part of RMW, Trap on Error	Not Checked	From System Int unchanged	Checked, Trap on Error
Secondary Cache Data Check Bits	Checked on read part of RMW, Trap on Error	Not Checked	From System Int unchanged	Checked, Trap on Error
Secondary Cache Tag & Check Bits	Checked on read part of RMW, Trap on Error	Checked and corrected	Generated	Checked, not corrected, Trap on Error
System Internal Addr/Cmd and Check Bits: Transmit	Generated	Generated	Generated	Generated
System Internal Addr/Cmd and Check Bits: Receive	NA	NA	Not Checked	NA
System Internal Data	From Processor	From Secondary	Checked, Trap on Error	From Secondary
System Internal Data Check Bits	Generated	From Secondary Cache	Checked, Trap on Error	From Secondary Cache

Table 15-3 Errror Checking and Correcting Summary for External Transactions

Bus	Read Request	Write Request	Invalidate Request	Update Request
Processor or Secondary Cache Data	NA	NA	Not Checked	Checked on read part of RMW, Trap on Error[1]
Secondary Cache Data Check Bits	NA	NA	Not Checked	Checked on read part of RMW, Trap on Error[1], Generation on write part of RMW if written
Secondary Cache Tag & Check Bits	NA	NA	Checked on read part of RMW, Trap on Error, Generation on write part of RMW if written	Checked on read part of RMW, Trap on Error, Generation on write part of RMW if written
System Internal Addr/ Cmd and Check Bits: Transmit	Generated	NA	NA	NA
System Internal Addr/ Cmd and Check Bits: Receive	Not Checked	Not Checked	Not Checked	Not Checked
System Internal Data	From Processor	Not Checked	Not Checked	Not Checked
System Internal Data Check Bits	Generated	Not Checked	Not Checked	Not Checked

Table 15-4 Error Checking and Correcting Summary for External Transactions

Bus	Intervention Request Data Returned	Intervention Request State Returned	Snoop Request
Processor or Secondary Cache Data	Checked, Trap on Error	Not Checked	Not Checked
Secondary Cache Data Check Bits	Checked, Trap on Error	Not Checked	Not Checked
Secondary Cache Tag & Check Bits	Checked and corrected on read part of RMW, Trap on Error, Generation on write part of RMW if written.	Checked and corrected on read part of RMW, Trap on Error, Generation on write part of RMW if written.	Checked and corrected on read part of RMW, Trap on Error, Generation on write part of RMW if written.
System Internal Addr/Cmd and Check Bits: Transmit	Generated	Generated	Generated
System Internal Addr/Cmd and Check Bits: Receive	Not Checked	Not Checked	Not Checked
System Internal Data	From Secondary	NA	NA
System Internal Data Check Bits	From Secondary Cache	NA	NA

Specifications

16

Electrical Characteristics

NOTE: The designer is advised to consult the vendor-specific data sheets for the exact information on electrical characteristics. The information in this chapter is provided as a reference only.

Maximum Ratings

Operation beyond the limits set forth in Table 16-1 may impair the useful life of the device.

Table 16-1 Maximum Ratings

Parameter	Symbol	Min	Max	Units
Supply Voltage	VCC	-0.5	+7.0	Volts
Input Voltage	VIN	-0.5[1]	+7.0	Volts
Storage Temperature	TST	-65.0	+150.0	Degrees C
Operating Temperature	TC	0	+85.0	Degrees C

(1) VIN Min. = -3.0V for pulse width less than 15 ns.

NOTE: No more than one output should be shorted at a time. Duration of the short should not exceed 30 seconds.

Operating Range

Table 16-2 Operating Range

Range	Case (TC)	VCC
Commercial	0C to 85C	5V ± 5%

Operating Parameters

Table 16-3 Operating Parameters

Parameter	Symbol	Conditions	50MHz		Units
			Min	Max	
Output HIGH Voltage	VOH	VCC = Min.	3.5		V
Clock Output HIGH Voltage[3]	VOHC	VCC = Min.	4.0		V
Output LOW Voltage	VOL	VCC = Min.		.4	V
Input HIGH Voltage[2]	VIH		2	VCC+.5	V
Input LOW Voltage[1,2]	VIL		-.5[(1)]	.8	V
MasterClock Input					
HIGH Voltage	VIHC		0.8VCC	VCC+.5	V
MasterClock Input					
LOW Voltage	VILC		-.5[(1)]	0.2VCC	V
Input Capacitance	CIn			10	pF
Output Capacitance	COut			10	pF
Operating Current	ICC	VCC = 5V,TC=0C		3	A
Input Leakage	ILeak			10	µA
Input/Output Leakage	IOLeak			20	µA

Notes:

(1) VIL Min. = -3.0V for pulse width less than 15 ns.

(2) Except for MasterClock and SyncIn input

(3) Applies to TClock, RClock, MasterOut, ModeClock and SyncOut outputs

MasterClock and Clock Parameters

Table 16-4 MasterClock and Clock Parameters

Parameter	Symbol	Test Conditions	50 MHz		Units
			Min	Max	
MasterClock High	t_{MCHigh}	Transition ≤ 5ns	4		ns
MasterClock Low	t_{MCLow}	Transition ≤ 5ns	4		ns
MasterClock Freq[1]			25	50	MHz
MasterClock Period	t_{MCP}		20	40	ns
Clock Jitter	$t_{MCJitter}$			500	ps
MasterClock Rise Time	t_{MCRise}			5	ns
MasterClock Fall Time	t_{MCFall}			5	ns
ModeClock Period	$t_{ModeCKP}$			$256{*}t_{MCP}$	ns

Notes:

(1) Operation of the R4000 is only guaranteed with the Phase Lock Loop enabled.

System Interface Parameters

Table 16-5 System Interface Parameters

Parameter	Symbol	Test Conditions	50MHz		Units
			Min	Max	
Data Output[1,2,3]	t_{DO}	Maximum Slew Rate	2	10	ns
		Modebits[53:56]=0			
		Modebits[57:60]=15			
		Minimum Slew Rate	6	16	ns
		Modebits[53:56]=15			
		Modebits[57:60]=0			
		MC*0.5 Drive Time	TBD	TBD	ns
		Modebit[50:52]=100			
		MC*0.75 Drive Time	TBD	TBD	ns
		Modebit[50:52]=010			
		MC*1.0 Drive Time	TBD	TBD	ns
		Modebit[50:52]=001			
Data Setup	t_{DS}		5		ns
Data Hold	t_{DH}		2		ns

Notes:

(1) When the dynamic output slew rate control Mode bits [61] or [62] are enabled, the initial values for the pull-up and pull-down rates should be set to the slowest value, Modebits[53:56]=15, Modebits[57:60]=0.

(2) Timings are measured from 1.5V of the clock to 1.5V of signal.

(3) Capacitive load for all output timings is 50 pf.

(4) Data Output, Data Setup and Data Hold apply to all logic signals driven out of or driven into the R4000 on the system interface. Secondary cache signals are specified separately.

NOTE: All output timing specifications given assume 50 pf of capacitive load. Output timing specifications should be derated where appropriate as shown in Table 16-6 below.

Secondary Cache Interface Parameters

Table 16-6 Secondary Cache Interface Parameters

Parameter	Symbol	Test Conditions	50MHz		Units
			Min	Max	
MasterClock to Output[1,2,3]	t_{SCO}	Maximum Slew Rate	2	10	ns
		Modebits[53:56]=0			
		Modebits[57:60]=15			
		Minimum Slew Rate	6	16	ns
		Modebits[53:56]=15			
		Modebits[57:60]=0			
		MC*0.5 Drive Time	TBD	TBD	ns
		Modebit[50:52]=100			
		MC*0.75 Drive Time	TBD	TBD	ns
		Modebit[50:52]=010			
		MC*1.0 Drive Time	TBD	TBD	ns
		Modebit[50:52]=001			
Data Setup	t_{SCDS}		5		ns
Data Hold	t_{SCDH}		2		ns
Cycle length of 4-word read	t_{Rd1Cyc}[4]		4	15	cycles
Cycles between read and write	t_{Dis}[4]		2	7	cycles
Cycle length of 8-word read	t_{Rd2Cyc}[4]		3	15	cycles
Cycles between Address and SCWr*	t_{Wr1Dly}[4]		1	3	cycles
Cycles between deassertion of SCWr* to the start of the next cycle	t_{WrRc}[4]		0	1	cycles
Cycles from second doubleword to SCWr*	t_{WrSUp}[4]		3	15	cycles
Cycles between first and second data word in 8-word write	t_{Wr2Dly}[4]		1	3	cycles

Notes:

(1) When the dynamic output slew rate control Mode bits [61] or [62] are enabled, the initial values for the pull-up and pull-down rates should be set to the slowest value, Modebits[53:56]=15, Modebits[57:60]=0.

(2) Timings are measured from 1.5V of the clock to 1.5V of signal.

(3) Capacitive load for all output timings is 50pf.

(4) Number of cycles is configured through the boot time mode control. Section 9.0 specifies the boot time mode interface.

Capacitive Load Deration

Table 16-7 Capacitive Load Deration

Parameter	Symbol	50 MHz		Units
		Min	Max	
Load Derate	CLD		2	ns/25pF

Physical Specifications

Signal to Pin Correlation of R4000PC

Table 16-8 lists the PC package signal layout; signals are listed alphabetically left to right and run from the top row on down.

Table 16-8 R4000 PC Package Layout

R4000 Function	PC Pkg Pin	R4000 Function	PC Pkg Pin	R4000 Function	PC Pkg Pin
ColdResetB	T14	ExtRqstB	U2	FaultB	B16
IOIn	T13	IOOut	U12	IntB0	N2
IntB1	L3	IntB2	K3	IntB3	J3
IntB4	H3	IntB5	F2	JTCK	H17
JTDI	G16	JTDO	F16	JTMS	E16
MasterClock	J17	MasterOut	P17	ModeClock	B4
ModeIn	U4	NMIB	U7	NoConnect	U10
PLLCap0	****	PLLCap1	****	RClock0	T17
RClock1	R16	RdRdyB	T5	ReleaseB	V5
ResetB	U16	SyncIn	J16	SyncOut	P16
SysAD0	J2	SysAD1	G2	SysAD2	E1
SysAD3	E3	SysAD4	C2	SysAD5	C4
SysAD6	B5	SysAD7	B6	SysAD8	B9
SysAD9	B11	SysAD10	C12	SysAD11	B14
SysAD12	B15	SysAD13	C16	SysAD14	D17
SysAD15	E18	SysAD16	K2	SysAD17	M2
SysAD18	P1	SysAD19	P3	SysAD20	T2
SysAD21	T4	SysAD22	U5	SysAD23	U6
SysAD24	U9	SysAD25	U11	SysAD26	T12
SysAD27	U14	SysAD28	U15	SysAD29	T16
SysAD30	R17	SysAD31	M16	SysAD32	H2
SysAD33	G3	SysAD34	F3	SysAD35	D2
SysAD36	C3	SysAD37	B3	SysAD38	C6
SysAD39	C7	SysAD40	C10	SysAD41	C11
SysAD42	B13	SysAD43	A15	SysAD44	C15

R4000 Function	PC Pkg Pin	R4000 Function	PC Pkg Pin	R4000 Function	PC Pkg Pin
SysAD45	B17	SysAD46	E17	SysAD47	F17
SysAD48	L2	SysAD49	M3	SysAD50	N3
SysAD51	R2	SysAD52	T3	SysAD53	U3
SysAD54	T6	SysAD55	T7	SysAD56	T10
SysAD57	T11	SysAD58	U13	SysAD59	V15
SysAD60	T15	SysAD61	U17	SysAD62	N16
SysAD63	N17	SysADC0	C8	SysADC1	G17
SysADC2	T8	SysADC3	L16	SysADC4	B8
SysADC5	H16	SysADC6	U8	SysADC7	L17
SysCmd0	E2	SysCmd1	D3	SysCmd2	B2
SysCmd3	A5	SysCmd4	B7	SysCmd5	C9
SysCmd6	B10	SysCmd7	B12	SysCmd8	C13
SysCmdP	C14	TClock0	C17	TClock1	D16
VCCOk	M17	ValidInB	P2	ValidOutB	R3
WrRdyB	C5	VccP	K17	VssP	K16
Vcc	A2	Vcc	A4	Vcc	A7
Vcc	A9	Vcc	A11	Vcc	A13
Vcc	A16	Vcc	B18	Vcc	C1
Vcc	D18	Vcc	F1	Vcc	G18
Vcc	H1	Vcc	J18	Vcc	K1
Vcc	L18	Vcc	M1	Vcc	N18
Vcc	R1	Vcc	T9	Vcc	T18
Vcc	U1	Vcc	V3	Vcc	V6
Vcc	V8	Vcc	V10	Vcc	V12
Vcc	V14	Vcc	V17	Vss	A3
Vss	A6	Vss	A8	Vss	A10
Vss	A12	Vss	A14	Vss	A17
Vss	A18	Vss	B1	Vss	C18
Vss	D1	Vss	F18	Vss	G1
Vss	H18	Vss	J1	Vss	K18
Vss	L1	Vss	M18	Vss	N1

R4000 Function	PC Pkg Pin	R4000 Function	PC Pkg Pin	R4000 Function	PC Pkg Pin
Vss	P18	Vss	R18	Vss	T1
Vss	U18	Vss	V1	Vss	V2
Vss	V4	Vss	V7	Vss	V9
Vss	V11	Vss	V13	Vss	V16
Vss	V18				

Pinout of R4000PC

Figure 16-1 shows the physical pinout of the R4000PC.

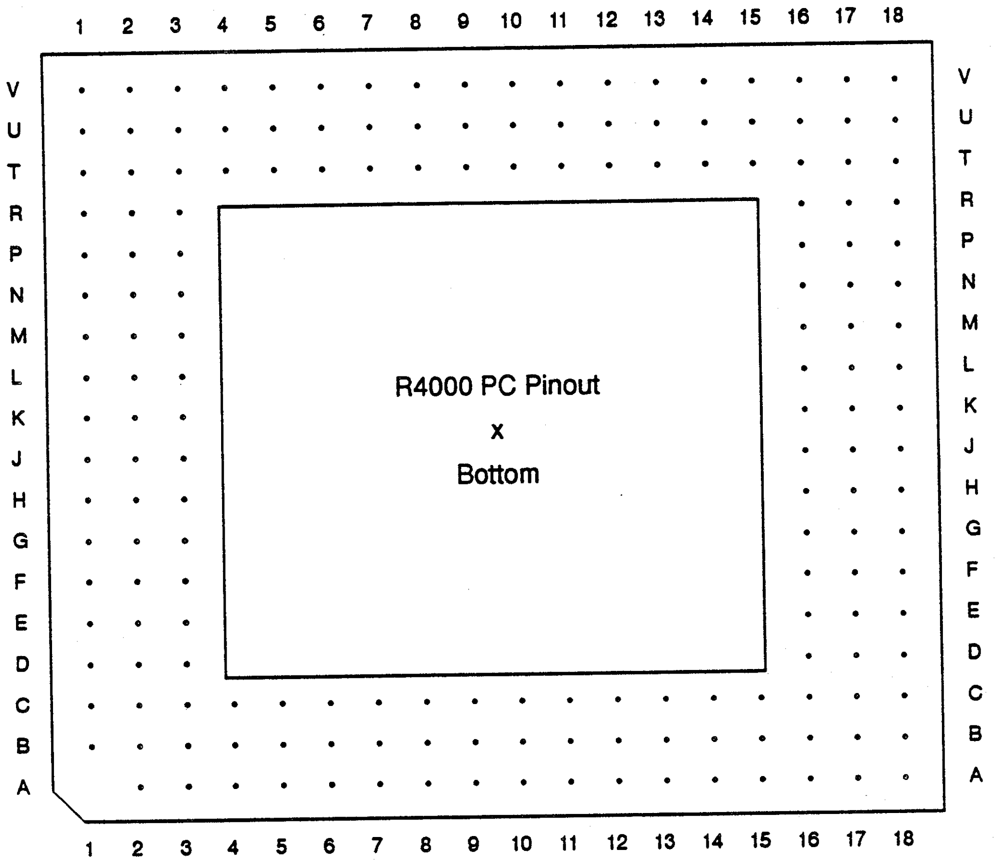


Figure 16-1 Pinout of R4000PC

Signal to Pin Correlation of R4000MC/SC

Table 16-8 lists the PC package signal layout; signals are listed alphabetically left to right and run from the top row on down.

Table 16-9 R4000MC/SC Package Layout

R4000 Function	MC/SC Pkg Pin	R4000 Function	MC/SC Pkg Pin	R4000 Function	MC/SC Pkg Pin
ColdResetB	AW37	ExtRqstB	AV2	FaultB	C39
IOIn	AV32	IOOut	AV28	IntB0	AL1
IvdAckB	AA35	IvdErrB	AA39	JTCK	U39
JTDI	N39	JTDO	J39	JTMS	G37
MasterClock	AA37	MasterOut	AJ39	ModeClock	B8
ModeIn	AV8	NMIB	AV16	NoConnect	AV24
NoConnect	Y2	PLLCap0	****	PLLCap1	****
RClock0	AM34	RClock1	AL33	RdRdyB	AW7
ReleaseB	AV12	ResetB	AU39	SCAPar0	U5
SCAPar1	U1	SCAPar2	P4	SCAddr1	AL5
SCAddr2	AG1	SCAddr3	AE7	SCAddr4	AC1
SCAddr5	AC5	SCAddr6	AC3	SCAddr7	AA1
SCAddr8	AB4	SCAddr9	AA5	SCAddr10	AA7
SCAddr11	AA3	SCAddr12	W3	SCAddr13	Y6
SCAddr14	W5	SCAddr15	W7	SCAddr16	W1
SCAddr17	U3	SCAddr0W	AN7	SCAddr0X	AN5
SCAddr0Y	AM6	SCAddr0Z	AL7	SCDCSB	M6
SCDChk0	G19	SCDChk1	T34	SCDChk2	AP20
SCDChk3	AD34	SCDChk4	C19	SCDChk5	R37
SCDChk6	AU19	SCDChk7	AE37	SCDChk8	C17
SCDChk9	N37	SCDChk10	AU17	SCDChk11	AG37
SCDChk12	E19	SCDChk13	R35	SCDChk14	AR19
SCDChk15	AE35	SCData0	R3	SCData1	R7
SCData2	L5	SCData3	F8	SCData4	C9
SCData5	F12	SCData6	G15	SCData7	E17
SCData8	G21	SCData9	C25	SCData10	G25
SCData11	E29	SCData12	G31	SCData13	C35

R4000 Function	MC/SC Pkg Pin	R4000 Function	MC/SC Pkg Pin	R4000 Function	MC/SC Pkg Pin
SCData14	K36	SCData15	N35	SCData16	AE3
SCData17	AG5	SCData18	AK4	SCData19	AN9
SCData20	AU9	SCData21	AN13	SCData22	AT14
SCData23	AR17	SCData24	AT22	SCData25	AU25
SCData26	AN27	SCData27	AR29	SCData28	AN31
SCData29	AR35	SCData30	AK36	SCData31	AG35
SCData32	T6	SCData33	L3	SCData34	L7
SCData35	E7	SCData36	G11	SCData37	E13
SCData38	E15	SCData39	G17	SCData40	C23
SCData41	F24	SCData42	E27	SCData43	D30
SCData44	C33	SCData45	E35	SCData46	L35
SCData47	R33	SCData48	AF4	SCData49	AJ3
SCData50	AJ7	SCData51	AP8	SCData52	AT10
SCData53	AR13	SCData54	AR15	SCData55	AT18
SCData56	AU23	SCData57	AT26	SCData58	AR27
SCData59	AN29	SCData60	AP32	SCData61	AN35
SCData62	AJ35	SCData63	AE33	SCData64	V4
SCData65	R5	SCData66	N5	SCData67	E5
SCData68	G9	SCData69	E11	SCData70	G13
SCData71	D14	SCData72	C21	SCData73	D22
SCData74	E25	SCData75	G27	SCData76	C31
SCData77	F32	SCData78	J35	SCData79	M34
SCData80	AC7	SCData81	AE5	SCData82	AG7
SCData83	AR5	SCData84	AR9	SCData85	AR11
SCData86	AN15	SCData87	AP16	SCData88	AU21
SCData89	AN23	SCData90	AR25	SCData91	AP28
SCData92	AU31	SCData93	AR33	SCData94	AL35
SCData95	AH34	SCData96	U7	SCData97	N3
SCData98	N7	SCData99	C5	SCData100	E9
SCData101	C11	SCData102	C13	SCData103	F16
SCData104	E21	SCData105	G23	SCData106	C27
SCData107	F28	SCData108	E31	SCData109	G33

R4000	MC/SC Pkg	R4000	MC/SC Pkg	R4000	MC/SC Pkg
Function	Pin	Function	Pin	Function	Pin
SCData110	J37	SCData111	N33	SCData112	AD6
SCData113	AG3	SCData114	AJ5	SCData115	AU5
SCData116	AN11	SCData117	AU11	SCData118	AU13
SCData119	AN17	SCData120	AR21	SCData121	AP24
SCData122	AU27	SCData123	AT30	SCData124	AU33
SCData125	AN33	SCData126	AL37	SCData127	AG33
SCOEB	N1	SCTCSB	J1	SCTChk0	AN21
SCTChk1	AN19	SCTChk2	AU15	SCTChk3	AP12
SCTChk4	AU7	SCTChk5	AR7	SCTChk6	AH6
SCTag0	K4	SCTag1	G7	SCTag2	C7
SCTag3	D10	SCTag4	C15	SCTag5	D18
SCTag6	F20	SCTag7	E23	SCTag8	D26
SCTag9	C29	SCTag10	G29	SCTag11	E33
SCTag12	G35	SCTag13	L33	SCTag14	L37
SCTag15	P36	SCTag16	AF36	SCTag17	AJ37
SCTag18	AJ33	SCTag19	AN37	SCTag20	AU35
SCTag21	AR31	SCTag22	AU29	SCTag23	AN25
SCTag24	AR23	SCWrWB	J5	SCWrXB	J7
SCWrYB	H6	SCWrZB	G5	Status0	U33
Status1	U35	Status2	V36	Status3	W35
Status4	W37	Status5	AC37	Status6	AC35
Status7	AC33	SyncIn	W39	SyncOut	AN39
SysAD0	T2	SysAD1	M2	SysAD2	J3
SysAD3	G3	SysAD4	C1	SysAD5	A3
SysAD6	A9	SysAD7	A13	SysAD8	A21
SysAD9	A25	SysAD10	A29	SysAD11	A33
SysAD12	B38	SysAD13	E37	SysAD14	G39
SysAD15	L39	SysAD16	AD2	SysAD17	AH2
SysAD18	AL3	SysAD19	AN3	SysAD20	AU1
SysAD21	AW3	SysAD22	AW9	SysAD23	AW13
SysAD24	AW21	SysAD25	AW25	SysAD26	AW29
SysAD27	AW33	SysAD28	AV38	SysAD29	AR37

R4000 Function	MC/SC Pkg Pin	R4000 Function	MC/SC Pkg Pin	R4000 Function	MC/SC Pkg Pin
SysAD30	AM38	SysAD31	AH38	SysAD32	R1
SysAD33	L1	SysAD34	H2	SysAD35	E1
SysAD36	C3	SysAD37	A5	SysAD38	A11
SysAD39	A15	SysAD40	A23	SysAD41	A27
SysAD42	A31	SysAD43	A35	SysAD44	C37
SysAD45	E39	SysAD46	H38	SysAD47	M38
SysAD48	AE1	SysAD49	AJ1	SysAD50	AM2
SysAD51	AR1	SysAD52	AU3	SysAD53	AW5
SysAD54	AW11	SysAD55	AW15	SysAD56	AW23
SysAD57	AW27	SysAD58	AW31	SysAD59	AW35
SysAD60	AU37	SysAD61	AR39	SysAD62	AL39
SysAD63	AG39	SysADC0	A17	SysADC1	R39
SysADC2	AW17	SysADC3	AD38	SysADC4	A19
SysADC5	T38	SysADC6	AW19	SysADC7	AC39
SysCmd0	G1	SysCmd1	E3	SysCmd2	B2
SysCmd3	B12	SysCmd4	B16	SysCmd5	B20
SysCmd6	B24	SysCmd7	B28	SysCmd8	A32
SysCmdP	A37	TClock0	H34	TClock1	J33
VCCOk	AE39	ValidInB	AN1	ValidOutB	AR3
WrRdyB	A7	VccSense	W33	VssSense	U37
VccP	AA33	VssP	Y34	Vcc	A39
Vcc	B6	Vcc	B10	Vcc	B18
Vcc	B26	Vcc	B34	Vcc	D4
Vcc	D8	Vcc	D16	Vcc	D24
Vcc	D32	Vcc	D36	Vcc	F2
Vcc	F14	Vcc	F22	Vcc	F30
Vcc	F38	Vcc	H4	Vcc	H36
Vcc	K6	Vcc	K38	Vcc	P2
Vcc	P34	Vcc	T4	Vcc	T36
Vcc	V6	Vcc	V38	Vcc	Y38
Vcc	AB2	Vcc	AB34	Vcc	AD4
Vcc	AD36	Vcc	AF6	Vcc	AF38

R4000 Function	MC/SC Pkg Pin	R4000 Function	MC/SC Pkg Pin	R4000 Function	MC/SC Pkg Pin
Vcc	AK2	Vcc	AK34	Vcc	AM4
Vcc	AM36	Vcc	AP2	Vcc	AP10
Vcc	AP18	Vcc	AP26	Vcc	AP38
Vcc	AT4	Vcc	AT8	Vcc	AT16
Vcc	AT24	Vcc	AT32	Vcc	AT36
Vcc	AV6	Vcc	AV14	Vcc	AV20
Vcc	AV22	Vcc	AV30	Vcc	AV34
Vcc	AW1	Vcc	AW39	Vss	B4
Vss	B14	Vss	B22	Vss	B30
Vss	B36	Vss	D2	Vss	D6
Vss	D12	Vss	D20	Vss	D28
Vss	D34	Vss	D38	Vss	F4
Vss	F6	Vss	F10	Vss	F18
Vss	F26	Vss	F34	Vss	F36
Vss	K2	Vss	K34	Vss	M4
Vss	M36	Vss	P6	Vss	P38
Vss	V2	Vss	V34	Vss	Y4
Vss	Y36	Vss	AB6	Vss	AB36
Vss	AB38	Vss	AF2	Vss	AF34
Vss	AH4	Vss	AH36	Vss	AK6
Vss	AK38	Vss	AP4	Vss	AP6
Vss	AP14	Vss	AP22	Vss	AP30
Vss	AP34	Vss	AP36	Vss	AT2
Vss	AT6	Vss	AT12	Vss	AT20
Vss	AT28	Vss	AT34	Vss	AT38
Vss	AV4	Vss	AV10	Vss	AV18
Vss	AV26	Vss	AV36		

Pinout of R4000MC/SC Package

Figure 16-2 shows the physical pinout of the R4000MC and SC.

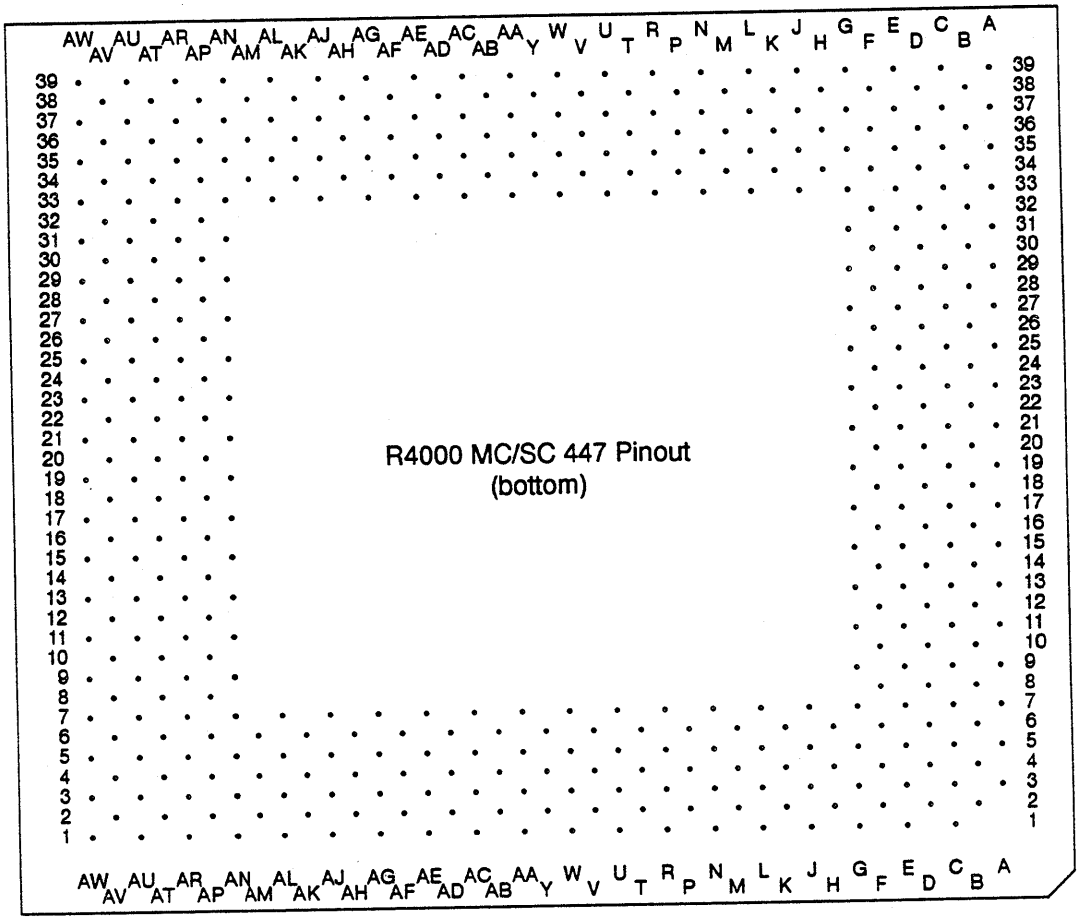


Figure 16-2 Pinout of the R4000MC and R4000SC

CPU Instruction Set Details

A

This appendix provides a detailed description of the operation of each R4000 instruction in both 32- and 64-bit modes. The instructions are listed in alphabetical order.

Refer to Appendix B for a detailed description of the FPU instructions.

The exceptions that may occur due to the execution of each instruction are listed after the description of each instruction. The description of the immediate causes and manner of handling exceptions is omitted from the instruction descriptions in this chapter. Refer to Chapter 5 for detailed descriptions of exceptions and handling.

Figures at the end of this appendix list the bit encoding for the constant fields of each instruction, and the bit encoding for each individual instruction is included with that instruction.

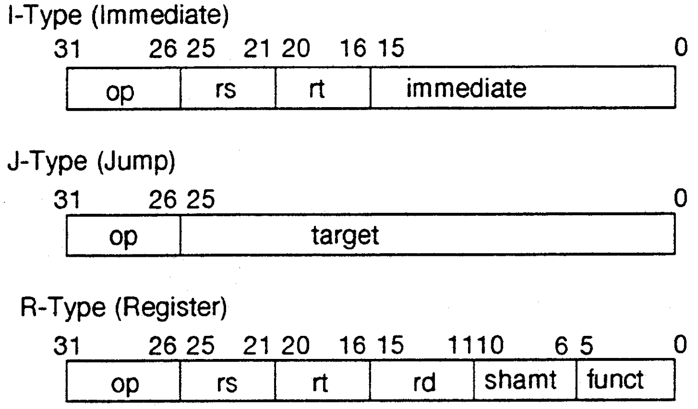
Instruction Classes

CPU instructions are divided into the following classes:

- **Load** and **Store** instructions move data between memory and general registers. They are all I-type instructions, since the only addressing mode supported is *base register + 16-bit immediate offset*.
- **Computational** instructions perform arithmetic, logical and shift operations on values in registers. They occur in both R-type (both operands are registers) and I-type (one operand is a 16-bit immediate) formats.
- **Jump** and **Branch** instructions change the control flow of a program. Jumps are always to absolute 26-bit word addresses (J-type format), or register addresses (R-type, for returns and dispatches). Branches have 16-bit offsets relative to the program counter (I-type). **Jump and Link** instructions save a return address in Register 31.
- **Coprocessor** instructions perform operations in the coprocessors. Coprocessor loads and stores are I-type. Coprocessor computational instructions have coprocessor-dependent formats (see the FPU instructions). Coprocessor zero (CP0) instructions manipulate the memory management and exception handling facilities of the processor.
- **Special** instructions perform a variety of tasks, including movement of data between special and general registers, trap, and breakpoint. They are always R-type.

Instruction Formats

Every CPU instruction consists of a single word (32 bits) aligned on a word boundary and the major instruction formats are shown in Figure A-1.



where:

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

Figure A-1 CPU Instruction Formats

Instruction Notation Conventions

In this appendix, all variable subfields in an instruction format (such as *rs, rt, immediate,* etc.) are shown in lowercase names.

For the sake of clarity, we sometimes use an alias for a variable subfield in the formats of specific instructions. For example, we use *rs* = *base* in the format for load and store instructions. Such an alias is always lower case, since it refers to a variable subfield.

Figures with the actual bit encoding for all the mnemonics are located at the end of this Appendix, and the bit encoding also accompanies each instruction.

In the instruction descriptions that follow, the *Operation* section describes the operation performed by each instruction using a high-level language notation. The R4000 can operate as either a 32- or 64-bit microprocessor. The operation for both modes is included with the instruction description. Special symbols used in the notation are described in Table A-1.

Table A-1 CPU Instruction Operation Notations

Symbol	Meaning
←	Assignment.
$\Vert$	Bit string concatenation.
x^y	Replication of bit value *x* into a *y*-bit string. Note: *x* is always a single-bit value.
$x_{y..z}$	Selection of bits *y* through *z* of bit string *x*. Little-endian bit notation is always used. If *y* is less than *z*, this expression is an empty (zero length) bit string.
+	Two's complement or floating-point addition.
-	Two's complement or floating-point subtraction.
*	Two's complement or floating-point multiplication.
div	Two's complement integer division.
mod	Two's complement modulo.
/	Floating-point division.
<	Two's complement less than comparison.
and	Bitwise logic AND.
or	Bitwise logic OR.
xor	Bitwise logic XOR.
nor	Bitwise logic NOR.
GPR[*x*]	General-Register x. The content of GPR[0] is always zero. Attempts to alter the content of GPR[0] have no effect.
CPR[*z,x*]	Coprocessor unit *z*, general register *x*.
CCR[*z,x*]	Coprocessor unit *z*, control register *x*.
COC[*z*]	Coprocessor unit *z* condition signal.
BigEndianMem	Big-endian mode as configured at reset (0 → Little, 1 → Big). Specifies the endianess of the memory interface (see LoadMemory and StoreMemory), and the endianess of Kernel and Supervisor mode execution.
ReverseEndian	Signal to reverse the endianess of load and store instructions. This feature is available in User mode only, and is effected by setting the *RE* bit of the *Status* register. Thus, ReverseEndian may be computed as (SR_{25} and User mode).
BigEndianCPU	The endianess for load and store instructions (0 → Little, 1 → Big). In User mode, this endianess may be reversed by setting SR_{25}. Thus, BigEndianCPU may be computed as BigEndianMem XOR ReverseEndian.
LLbit	Bit of state to specify synchronization instructions. Set by LL, cleared by ERET and Invalidate and read by SC.
T+*i*:	Indicates the time steps between operations. Each of the statements within a time step are defined to be executed in sequential order (as modified by conditional and loop constructs). Operations which are marked *T+i:* are executed at instruction cycle *i* relative to the start of execution of the instruction. Thus, an instruction which starts at time *j* executes operations marked T+*i:* at time *i + j*. The interpretation of the order of execution between two instructions or two operations which execute at the same time should be pessimistic; the order is not defined.

Instruction Notation Examples

The following examples illustrate the application of some of the instruction notation conventions:

Example #1:
$\text{GPR[rt]} \leftarrow \text{immediate} \parallel 0^{16}$
Sixteen zero bits are concatenated with an immediate value (typically 16 bits), and the 32-bit string (with the lower 16 bits set to zero) is assigned to General-Purpose Register rt.
Example #2:
$(\text{immediate}_{15})^{16} \parallel \text{immediate}_{15..0}$
Bit 15 (the sign bit) of an immediate value is extended for 16 bit positions, and the result is concatenated with bits 15 through 0 of the immediate value to form a 32-bit sign extended value.

Load and Store Instructions

In the R4000 implementation, the instruction immediately following a load may use the contents of the register loaded. In such cases, the hardware interlocks, requiring additional real cycles, so scheduling load delay slots is still desirable, although not required for functional code.

Two special instructions are provided in the R4000 implementation of the MIPS ISA, Load Linked and Store Conditional. These instructions are used in carefully coded sequences to provide one of several synchronization primitives, including test-and-set, bit-level locks, semaphores, and sequencers/event counts.

In the load and store operation descriptions, the functions listed in Table A-2 are used to summarize the handling of virtual addresses and physical memory.

Table A-2 Load and Store Common Functions

Function	Meaning
AddressTranslation	Uses the TLB to find the physical address given the virtual address. The function fails and an exception is taken if the required translation is not present in the TLB.
LoadMemory	Uses the cache and main memory to find the contents of the word containing the specified physical address. The low-order two bits of the address and the access type field indicates which of each of the four bytes within the data word need to be returned. If the cache is enabled for this access, the entire word is returned and loaded into the cache.
StoreMemory	Uses the cache, write buffer, and main memory to store the word or part of word specified as data in the word containing the specified physical address. The low-order two bits of the address and the access type field indicates which of each of the four bytes within the data word should be stored.

The access type field indicates the size of the data item to be loaded or stored as shown in Table A-3. Regardless of access type or byte-numbering order (endianness), the address specifies the byte which has the smallest byte address of the bytes in the addressed field. For a Big-endian machine, this is the leftmost byte and contains the sign for a 2's-complement number; for a Little-endian machine, this is the rightmost byte and contains the lowest precision byte.

Table A-3 Access Type Specifications for Loads/Stores

Access type Mnemonic	Value	Meaning
DOUBLEWORD	7	doubleword (64 bits)
SEPTIBYTE	6	seven bytes (56 bits)
SEXTIBYTE	5	six bytes (48 bits)
QUINTIBYTE	4	five bytes (40 bits)
WORD	3	word (32 bits)
TRIPLEBYTE	2	triple-byte (24 bits)
HALFWORD	1	halfword (16 bits)
BYTE	0	byte (8 bits)

The bytes within the addressed doubleword which are used can be determined directly from the access type and the three low-order bits of the address, as shown in Chapter 3.

Jump and Branch Instructions

All jump and branch instructions have an architectural delay of exactly one instruction. That is, the instruction immediately following a jump or branch (i.e., occupying the delay slot) is always executed while the target instruction is being fetched from storage. It is not valid for a delay slot to be occupied itself by a jump or branch instruction; however, this error is not detected, and the results of such an operation are undefined.

If an exception or interrupt prevents the completion of a legal instruction during a delay slot, the hardware sets the *EPC* register to point at the jump or branch instruction which precedes it. When the code is restarted, both the jump or branch instructions and the instruction in the delay slot are reexecuted.

Because jump and branch instructions may be restarted after exceptions or interrupts, they must be restartable. Therefore, when a jump or branch instruction stores a return link value, register 31 (the register in which the link is stored) may not be used as a source register.

Since instructions must be word-aligned, a Jump Register or Jump and Link Register instruction must use a register whose two low-order bits are zero. If these low-order bits are not zero, an address exception will occur when the jump target instruction is subsequently fetched.

Coprocessor Instructions

The MIPS architecture provides four coprocessor units, or classes. Coprocessors are alternate execution units, which have separate register files from the CPU. R-Series coprocessors have 2 register spaces, each with thirty-two 32-bit registers. The first space, *coprocessor general* registers, may be directly loaded from memory and stored into memory, and their contents may be transferred between the coprocessor and processor. The second, *coprocessor control* registers, may only have their contents transferred directly between the coprocessor and processor. Coprocessor instructions may alter registers in either space.

Normally, by convention, *Coprocessor Control Register 0* is interpreted as a *Coprocessor Implementation And Revision* register. However, the system control coprocessor (CP0) uses *Coprocessor General Register 15* for the processor/coprocessor revision register. The register's low-order byte (bits 7..0) is interpreted as a coprocessor unit revision number. The second byte (bits 15..8) is interpreted as a coprocessor unit implementation descriptor. The revision number is a value of 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 contents of the high-order halfword of the register are not defined (currently read as 0 and should be 0 when written).

System Control Coprocessor (CP0) Instructions

There are some special limitations imposed on operations involving CP0 that is incorporated within the CPU. Although load and store instructions to transfer data to and from coprocessors and move control to/from coprocessor instructions are generally permitted by the MIPS architecture, CP0 is given a somewhat protected status since it has responsibility for exception handling and memory management. Therefore, the move to/from coprocessor instructions are the only valid mechanism for reading from and writing to the CP0 registers.

Several coprocessor operation instructions are defined for CP0 to directly read, write, and probe TLB entries and to modify the operating modes in preparation for returning to User mode or interrupt-enabled states.

ADD Add ADD

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	ADD 100000
6	5	5	5	5	6

Format:

ADD rd, rs, rt

Description:

The contents of general register *rs* and the contents of general register *rt* are added to form the result. The result is placed into general register *rd*. In 64-bit mode, the operands must be valid sign-extended, 32-bit values.

An overflow exception occurs if the carries out of bits 30 and 31 differ (2's-complement overflow). The destination register *rd* is not modified when an integer overflow exception occurs.

Operation:

```
32    T:    $GPR[rd] \leftarrow GPR[rs] + GPR[rt]$

64    T:    $temp \leftarrow GPR[rs] + GPR[rt]$
            $GPR[rd] \leftarrow (temp_{31})^{32} \,||\, temp_{31..0}$
```

Exceptions:

Integer overflow exception

ADDI Add Immediate ADDI

31 26	25 21	20 16	15 0
ADDI 001000	rs	rt	immediate
6	5	5	16

Format:

ADDI rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and added to the contents of general register *rs* to form the result. The result is placed into general register *rt*. In 64-bit mode, the operand must be valid sign-extended, 32-bit values.

An overflow exception occurs if carries out of bits 30 and 31 differ (2's-complement overflow). The destination register *rt* is not modified when an integer overflow exception occurs.

Operation:

```
32   T:   $GPR[rt] \leftarrow GPR[rs] + (immediate_{15})^{16} \| immediate_{15..0}$

64   T:   $temp \leftarrow GPR[rs] + (immediate_{15})^{48} \| immediate_{15..0}$
          $GPR[rt] \leftarrow (temp_{31})^{32} \| temp_{31..0}$
```

Exceptions:

Integer overflow exception

ADDIU Add Immediate Unsigned ADDIU

31 26	25 21	20 16	15 0
ADDIU 001001	rs	rt	immediate
6	5	5	16

Format:

ADDIU rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and added to the contents of general register *rs* to form the result. The result is placed into general register *rt*. No integer overflow exception occurs under any circumstances. In 64-bit mode, the operand must be valid sign-extended, 32-bit values.

The only difference between this instruction and the ADDI instruction is that ADDIU never causes an overflow exception.

Operation:

```
32    T:    $GPR\ [rt] \leftarrow GPR[rs] + (immediate_{15})^{16} \,||\, immediate_{15..0}$

64    T:    $temp \leftarrow GPR[rs] + (immediate_{15})^{48} \,||\, immediate_{15..0}$
            $GPR[rt] \leftarrow (temp_{31})^{32} \,||\, temp_{31..0}$
```

Exceptions:

None

ADDU Add Unsigned ADDU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	ADDU 100001
6	5	5	5	5	6

Format:

ADDU rd, rs, rt

Description:

The contents of general register *rs* and the contents of general register *rt* are added to form the result. The result is placed into general register *rd*. No overflow exception occurs under any circumstances. In 64-bit mode, the operands must be valid sign-extended, 32-bit values.

The only difference between this instruction and the ADD instruction is that ADDU never causes an overflow exception.

Operation:

```
32    T:    GPR[rd] ←GPR[rs] + GPR[rt]

64    T:    temp ← GPR[rs] + GPR[rt]
            GPR[rd] ← $(temp_{31})^{32}$ || $temp_{31..0}$
```

Exceptions:

None

AND And AND

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	AND 100100
6	5	5	5	5	6

Format:

AND rd, rs, rt

Description:

The contents of general register *rs* are combined with the contents of general register *rt* in a bit-wise logical AND operation. The result is placed into general register *rd*.

Operation:

```
32    T:    GPR[rd] ← GPR[rs] and GPR[rt]

64    T:    GPR[rd] ← GPR[rs] and GPR[rt]
```

Exceptions:

None

ANDI — And Immediate — ANDI

31 26	25 21	20 16	15 0
ANDI 001100	rs	rt	immediate
6	5	5	16

Format:

ANDI rt, rs, immediate

Description:

The 16-bit *immediate* is zero-extended and combined with the contents of general register *rs* in a bit-wise logical AND operation. The result is placed into general register *rt*.

Operation:

32 T: $\text{GPR[rt]} \leftarrow 0^{16} \,||\, (\text{immediate } \textit{and } \text{GPR[rs]}_{15..0})$

64 T: $\text{GPR[rt]} \leftarrow 0^{48} \,||\, (\text{immediate } \textit{and } \text{GPR[rs]}_{15..0})$

Exceptions:

None

BCzF Branch On Coprocessor z False BCzF

31 26	25 21	20 16	15 0
COPz 0 1 0 0 x x*	BC 0 1 0 0 0	BCF 0 0 0 0 0	offset
6	5	5	16

Format:

BCzF offset

Description:

A 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. If coprocessor z's condition signal (CpCond), as sampled during the previous instruction, is false, then the program branches to the target address with a delay of one instruction.

Because the condition line is sampled during the previous instruction, there must be at least one instruction between this instruction and a coprocessor instruction that changes the condition line.

Operation:

```
32    T–1:  condition ← not COC[z]
      T:    target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
      T+1:  if condition then
                PC ← PC + target
            endif

64    T–1:  condition ← not COC[z]
      T:    target ← $(\text{offset}_{15})^{38}$ || offset || $0^2$
      T+1:  if condition then
                PC ← PC + target
            endif
```

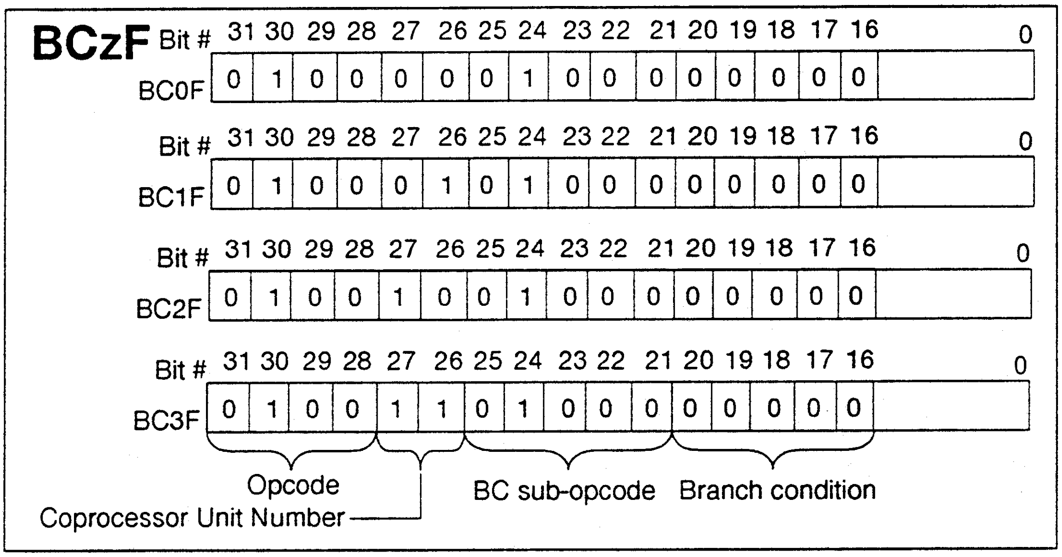
*See the table "Opcode Bit Encoding" on next page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

BCzF Branch On Coprocessor z False (continued) BCzF

Exceptions:

Coprocessor unusable exception

Opcode Bit Encoding:



BCzF

Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC0F	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	
BC1F	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	
BC2F	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	
BC3F	0	1	0	0	1	1	0	1	0	0	0	0	0	0	0	0	

BCzFL Branch On Coprocessor z False Likely BCzFL

31 — 26	25 — 21	20 — 16	15 — 0
COPz 0 1 0 0 x x*	BC 0 1 0 0 0	BCFL 0 0 0 1 0	offset
6	5	5	16

Format:

BCzFL offset

Description:

A 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. If the contents of coprocessor *z*'s condition line, as sampled during the previous instruction, is false, the target address is branched to with a delay of one instruction.

If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

Because the condition line is sampled during the previous instruction, there must be at least one instruction between this instruction and a coprocessor instruction that changes the condition line.

*See the table "Opcode Bit Encoding" on next page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

BCzFL Branch On Coprocessor z False Likely (continued) BCzFL

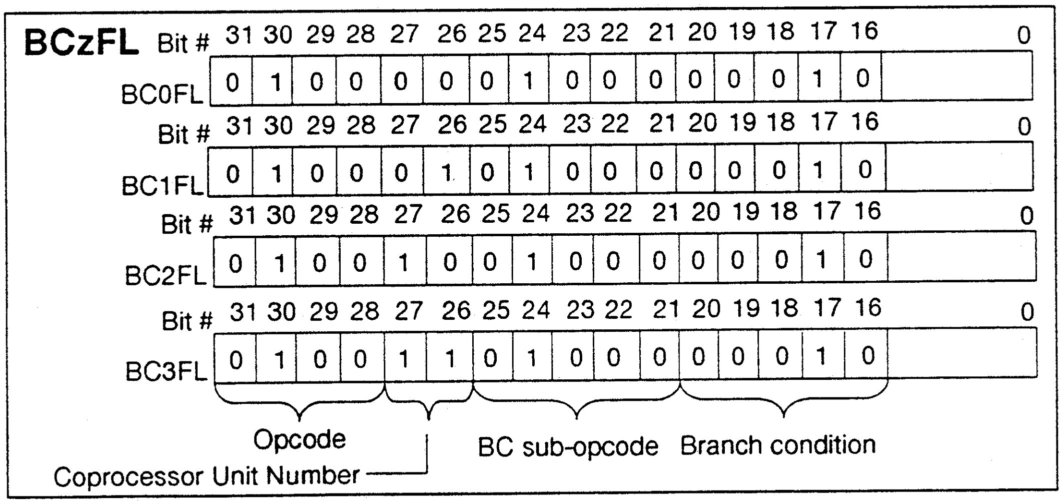
Operation:

```
32    T–1:  condition ← not COC[z]
      T:    target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
      T+1:  if condition then
                PC ← PC + target
            else
                NullifyCurrentInstruction
            endif

64    T–1:  condition ← not COC[z]
      T:    target ← $(\text{offset}_{15})^{38}$ || offset || $0^2$
      T+1:  if condition then
                PC ← PC + target
            else
                NullifyCurrentInstruction
            endif
```

Exceptions:

Coprocessor unusable exception

Opcode Bit Encoding:



BCzFL

Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC0FL	0	1	0	0	0	0	0	1	0	0	0	0	0	0	1	0	
BC1FL	0	1	0	0	0	1	0	1	0	0	0	0	0	0	1	0	
BC2FL	0	1	0	0	1	0	0	1	0	0	0	0	0	0	1	0	
BC3FL	0	1	0	0	1	1	0	1	0	0	0	0	0	0	1	0	

BCzT Branch On Coprocessor z True BCzT

31 26	25 21	20 16	15 0
COPz 0 1 0 0 x x*	BC 0 1 0 0 0	BCT 0 0 0 0 1	offset
6	5	5	16

Format:

BCzT offset

Description:

A 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. If the coprocessor z's condition signal (CpCond) is true, then the program branches to the target address, with a delay of one instruction.

Because the condition line is sampled during the previous instruction, there must be at least one instruction between this instruction and a coprocessor instruction that changes the condition line.

Operation:

```
32   T–1:  condition ← COC[z]
     T:    target ← $(offset_{15})^{14}$ || offset || $0^2$
     T+1:  if condition then
                PC ← PC + target
           endif
64   T–1:  condition ← COC[z]
     T:    target ← $(offset_{15})^{38}$ || offset || $0^2$
     T+1:  if condition then
                PC ← PC + target
           endif
```

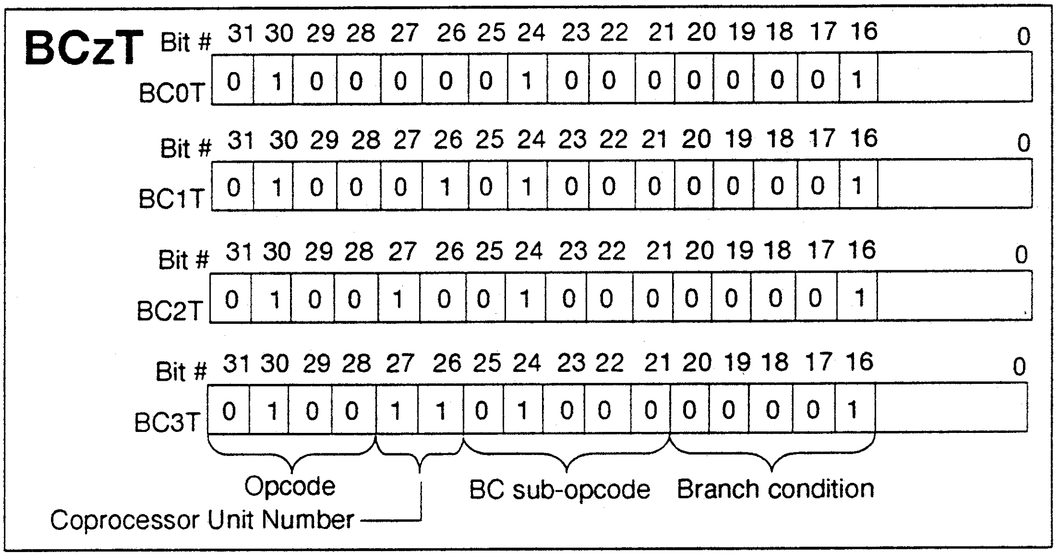
*See the table "Opcode Bit Encoding" on next page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

BCzT Branch On Coprocessor z True (continued) BCzT

Exceptions:

Coprocessor unusable exception

Opcode Bit Encoding:



BCzT

Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC0T	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1	
BC1T	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	1	
BC2T	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	1	
BC3T	0	1	0	0	1	1	0	1	0	0	0	0	0	0	0	1	

BCzTL Branch On Coprocessor z True Likely BCzTL

31 26	25 21	20 16	15 0
COPz 0 1 0 0 x x*	BC 0 1 0 0 0	BCTL 0 0 0 1 1	offset
6	5	5	16

Format:

BCzTL offset

Description:

A 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. If the contents of coprocessor z's condition line, as sampled during the previous instruction, is true, the target address is branched to with a delay of one instruction.

If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

Because the condition line is sampled during the previous instruction, there must be at least one instruction between this instruction and a coprocessor instruction that changes the condition line.

Operation:

```
32   T–1:  condition ← COC[z]
     T:    target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
     T+1:  if condition then
                PC ← PC + target
           else
                NullifyCurrentInstruction
           endif
64   T–1:  condition ← COC[z]
     T:    target ← $(\text{offset}_{15})^{38}$|| offset || $0^2$
     T+1:  if condition then
                PC ← PC + target
           else
                NullifyCurrentInstruction
           endif
```

*See the table "Opcode Bit Encoding" on next page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

BCzTL Branch On Coprocessor z True Likely (continued) BCzTL

Exceptions:

Coprocessor unusable exception

Opcode Bit Encoding:

BCzTL Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC0TL	0	1	0	0	0	0	0	1	0	0	0	0	0	0	1	1	
Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC1TL	0	1	0	0	0	1	0	1	0	0	0	0	0	0	1	1	
Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC2TL	0	1	0	0	1	0	0	1	0	0	0	0	0	0	1	1	
Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC3TL	0	1	0	0	1	1	0	1	0	0	0	0	0	0	1	1	

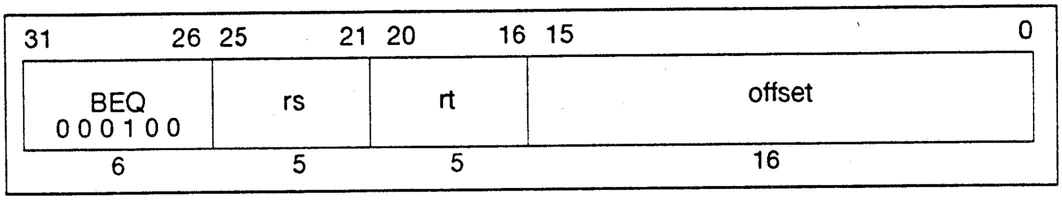
Opcode

Coprocessor Unit Number

BC sub-opcode

Branch condition

BEQ Branch On Equal BEQ



Format:

BEQ rs, rt, offset

Description:

A 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. The contents of general register *rs* and the contents of general register *rt* are compared. If the two registers are equal, then the program branches to the target address, with a delay of one instruction.

Operation:

```
32   T:     target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
            condition ← (GPR[rs] = GPR[rt])
     T+1:   if condition then
                 PC ← PC + target
            endif
64   T:     target ← $(\text{offset}_{15})^{38}$ || offset || $0^2$
            condition ← (GPR[rs] = GPR[rt])
     T+1:   if condition then
                 PC ← PC + target
            endif
```

Exceptions:

None

BEQL Branch On Equal Likely BEQL

31 26	25 21	20 16	15 0
BEQL 010100	rs	rt	offset
6	5	5	16

Format:

BEQL rs, rt, offset

Description:

A 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. The contents of general register *rs* and the contents of general register *rt* are compared. If the two registers are equal, the target address is branched to, with a delay of one instruction. If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

Operation:

```
32    T:    target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
            condition ← (GPR[rs] = GPR[rt])
      T+1:  if condition then
                PC ← PC + target
            else
                NullifyCurrentInstruction
            endif
64    T:    target ← $(\text{offset}_{15})^{38}$ || offset || $0^2$
            condition ← (GPR[rs] = GPR[rt])
      T+1:  if condition then
                PC ← PC + target
            else
                NullifyCurrentInstruction
            endif
```

Exceptions:

None

BGEZ — Branch On Greater Than Or Equal To Zero — BGEZ

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BGEZ 00001	offset
6	5	5	16

Format:

BGEZ rs, offset

Description:

A 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. If the contents of general register *rs* have the sign bit cleared, then the program branches to the target address, with a delay of one instruction.

Operation:

```
32    T:    target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
            condition ← $(\text{GPR[rs]}_{31} = 0)$
      T+1:  if condition then
                  PC ← PC + target
            endif
64    T:    target ← $(\text{offset}_{15})^{38}$ || offset || $0^2$
            condition ← $(\text{GPR[rs]}_{63} = 0)$
      T+1:  if condition then
                  PC ← PC + target
            endif
```

Exceptions:

None

BGEZAL Branch On Greater Than Or Equal To Zero And Link BGEZAL

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BGEZAL 10001	offset
6	5	5	16

Format:

BGEZAL rs, offset

Description:

A 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. Unconditionally, the address of the instruction after the delay slot is placed in the link register, *r31*. If the contents of general register *rs* have the sign bit cleared, then the program branches to the target address, with a delay of one instruction.

General register *rs* may not be general register *31*, because such an instruction is not restartable. An attempt to execute this instruction is *not* trapped, however.

Operation:

```
32    T:    target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
            condition ← $(\text{GPR[rs]}_{31} = 0)$
            GPR[31] ← PC + 8
      T+1:  if condition then
                 PC ← PC + target
            endif
64    T:    target ← $(\text{offset}_{15})^{38}$ || offset || $0^2$
            condition ← $(\text{GPR[rs]}_{63} = 0)$
            GPR[31] ← PC + 8
      T+1:  if condition then
                 PC ← PC + target
            endif
```

Exceptions:

None

BGEZALL Branch On Greater Than Or Equal To Zero And Link Likely BGEZALL

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BGEZALL 10011	offset
6	5	5	16

Format:

BGEZALL rs, offset

Description:

A 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. Unconditionally, the address of the instruction after the delay slot is placed in the link register, *r31*. If the contents of general register *rs* have the sign bit cleared, then the program branches to the target address, with a delay of one instruction.

General register *rs* may not be general register *31*, because such an instruction is not restartable. An attempt to execute this instruction is *not* trapped, however. If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

Operation:

```
32   T:    target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
           condition ← $(\text{GPR[rs]}_{31} = 0)$
           GPR[31] ← PC + 8
     T+1:  if condition then
               PC ← PC + target
           else
               NullifyCurrentInstruction
           endif
64   T:    target ← $(\text{offset}_{15})^{38}$ || offset || $0^2$
           condition ← $(\text{GPR[rs]}_{63} = 0)$
           GPR[31] ← PC + 8
     T+1:  if condition then
               PC ← PC + target
           else
               NullifyCurrentInstruction
           endif
```

Exceptions:

None

BGEZL — Branch On Greater Than Or Equal To Zero Likely — BGEZL

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BGEZL 00011	offset
6	5	5	16

Format:

BGEZL rs, offset

Description:

A 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. If the contents of general register *rs* have the sign bit cleared, then the program branches to the target address, with a delay of one instruction. If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

Operation:

```
32   T:    target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
           condition ← $(\text{GPR[rs]}_{31} = 0)$
     T+1:  if condition then
                PC ← PC + target
           else
                NullifyCurrentInstruction
           endif
64   T:    target ← $(\text{offset}_{15})^{38}$ || offset || $0^2$
           condition ← $(\text{GPR[rs]}_{63} = 0)$
     T+1:  if condition then
                PC ← PC + target
           else
                NullifyCurrentInstruction
           endif
```

Exceptions:

None

BGTZ Branch On Greater Than Zero BGTZ

31 26	25 21	20 16	15 0
BGTZ 0 0 0 1 1 1	rs	0 0 0 0 0 0	offset
6	5	5	16

Format:

BGTZ rs, offset

Description:

A 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. The contents of general register *rs* are compared to zero. If the contents of general register *rs* have the sign bit cleared and are not equal to zero, then the program branches to the target address, with a delay of one instruction.

Operation:

```
32   T:    target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
           condition ← $(\text{GPR}[rs]_{31} = 0)$ and $(\text{GPR}[rs] \neq 0^{32})$
     T+1:  if condition then
               PC ← PC + target
           endif
64   T:    target ← $(\text{offset}_{15})^{38}$ || offset || $0^2$
           condition ← $(\text{GPR}[rs]_{63} = 0)$ and $(\text{GPR}[rs] \neq 0^{64})$
     T+1:  if condition then
               PC ← PC + target
           endif
```

Exceptions:

None

BGTZL Branch On Greater Than Zero Likely BGTZL

31 26	25 21	20 16	15 0
BGTZL 0 1 0 1 1 1	rs	0 0 0 0 0 0	offset
6	5	5	16

Format:

BGTZL rs, offset

Description:

A 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. The contents of general register *rs* are compared to zero. If the contents of general register *rs* have the sign bit cleared and are not equal to zero, then the program branches to the target address, with a delay of one instruction. If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

Operation:

```
32   T:    target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
           condition ← $(\text{GPR[rs]}_{31} = 0)$ and $(\text{GPR[rs]} \neq 0^{32})$
     T+1:  if condition then
                PC ← PC + target
           else
                NullifyCurrentInstruction
           endif
64   T:    target ← $(\text{offset}_{15})^{38}$ || offset || $0^2$
           condition ← $(\text{GPR[rs]}_{63} = 0)$ and $(\text{GPR[rs]} \neq 0^{64})$
     T+1:  if condition then
                PC ← PC + target
           else
                NullifyCurrentInstruction
           endif
```

Exceptions:

None

BLEZ Branch on Less Than Or Equal To Zero BLEZ

31 26	25 21	20 16	15 0
BLEZ 000110	rs	0 00000	offset
6	5	5	16

Format:

BLEZ rs, offset

Description:

A 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. The contents of general register *rs* are compared to zero. If the contents of general register *rs* have the sign bit set, or are equal to zero, then the program branches to the target address, with a delay of one instruction.

Operation:

```
32   T:    target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
           condition ← $(\text{GPR[rs]}_{31} = 1)$ or $(\text{GPR[rs]} = 0^{32})$
     T+1:  if condition then
                PC ← PC + target
           endif
64   T:    target ← $(\text{offset}_{15})^{38}$ || offset || $0^2$
           condition ← $(\text{GPR[rs]}_{63} = 1)$ and $(\text{GPR[rs]} = 0^{64})$
     T+1:  if condition then
                PC ← PC + target
           endif
```

Exceptions:

None

BLEZL Branch on Less Than Or Equal To Zero Likely BLEZL

31 26	25 21	20 16	15 0
BLEZL 010110	rs	0 00000	offset
6	5	5	16

Format:

BLEZL rs, offset

Description:

A 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. The contents of general register *rs* is compared to zero. If the contents of general register *rs* have the sign bit set, or are equal to zero, then the program branches to the target address, with a delay of one instruction.

If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

Operation:

```
32    T:    target ← $(offset_{15})^{14}$ || offset || $0^2$
            condition ← $(GPR[rs]_{31} = 1)$ and $(GPR[rs] = 0^{32})$
      T+1:  if condition then
                 PC ← PC + target
            else
                 NullifyCurrentInstruction
            endif
64    T:    target ← $(offset_{15})^{38}$ || offset || $0^2$
            condition ← $(GPR[rs]_{63} = 1)$ and $(GPR[rs] = 0^{64})$
      T+1:  if condition then
                 PC ← PC + target
            else
                 NullifyCurrentInstruction
            endif
```

Exceptions:

None

BLTZ Branch On Less Than Zero BLTZ

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BLTZ 00000	offset
6	5	5	16

Format:

BLTZ rs, offset

Description:

A 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. If the contents of general register *rs* have the sign bit set, then the program branches to the target address, with a delay of one instruction.

Operation:

```
32   T:    target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
           condition ← $(\text{GPR}[rs]_{31} = 0)$
     T+1:  if condition then
                PC ← PC + target
           endif
64   T:    target ← $(\text{offset}_{15})^{38}$ || offset || $0^2$
           condition ← $(\text{GPR}[rs]_{63} = 0)$
     T+1:  if condition then
                PC ← PC + target
           endif
```

Exceptions:

None

BLTZAL Branch On Less Than Zero And Link BLTZAL

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BLTZAL 10000	offset
6	5	5	16

Format:

BLTZAL rs, offset

Description:

A 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. Unconditionally, the address of the instruction after the delay slot is placed in the link register, *r31*. If the contents of general register *rs* have the sign bit set, then the program branches to the target address, with a delay of one instruction.

General register *rs* may not be general register *31*, because such an instruction is not restartable. An attempt to execute this instruction with register 31 specified as *rs* is *not* trapped, however.

Operation:

```
32    T:    $target \leftarrow (offset_{15})^{14} \| offset \| 0^2$
            $condition \leftarrow (GPR[rs]_{31} = 1)$
            $GPR[31] \leftarrow PC + 8$
      T+1:  if condition then
                  $PC \leftarrow PC + target$
            endif
64    T:    $target \leftarrow (offset_{15})^{38} \| offset \| 0^2$
            $condition \leftarrow (GPR[rs]_{63} = 1)$
            $GPR[31] \leftarrow PC + 8$
      T+1:  if condition then
                  $PC \leftarrow PC + target$
            endif
```

Exceptions:

None

BLTZALL Branch On Less Than Zero And Link Likely BLTZALL

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BLTZALL 10010	offset
6	5	5	16

Format:

BLTZALL rs, offset

Description:

A 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. Unconditionally, the address of the instruction after the delay slot is placed in the link register, *r31*. If the contents of general register *rs* have the sign bit set, then the program branches to the target address, with a delay of one instruction.

General register *rs* may not be general register *31*, because such an instruction is not restartable. An attempt to execute this instruction with register 31 specified as *rs* is *not* trapped, however. If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

Operation:

```
32    T:    target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
            condition ← $(\text{GPR}[rs]_{31} = 1)$
            GPR[31] ← PC + 8
      T+1:  if condition then
                PC ← PC + target
            else
                NullifyCurrentInstruction
            endif
64    T:    target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
            condition ← $(\text{GPR}[rs]_{63} = 1)$
            GPR[31] ← PC + 8
      T+1:  if condition then
                PC ← PC + target
            else
                NullifyCurrentInstruction
            endif
```

Exceptions:

None

BLTZL Branch On Less Than Zero Likely BLTZL

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BLTZL 00010	offset
6	5	5	16

Format:

BLTZ rs, offset

Description:

A 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. If the contents of general register *rs* have the sign bit set, then the program branches to the target address, with a delay of one instruction. If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

Operation:

```
32    T:    target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
            condition ← $(\text{GPR[rs]}_{31} = 1)$
      T+1:  if condition then
                  PC ← PC + target
            else
                  NullifyCurrentInstruction
            endif
64    T:    target ← $(\text{offset}_{15})^{38}$ || offset || $0^2$
            condition ← $(\text{GPR[rs]}_{63} = 1)$
      T+1:  if condition then
                  PC ← PC + target
            else
                  NullifyCurrentInstruction
```

Exceptions:

None

BNE Branch On Not Equal BNE

31 26	25 21	20 16	15 0
BNE 0 0 0 1 0 1	rs	rt	offset
6	5	5	16

Format:

BNE rs, rt, offset

Description:

A 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. The contents of general register *rs* and the contents of general register *rt* are compared. If the two registers are not equal, then the program branches to the target address, with a delay of one instruction.

Operation:

```
32   T:     target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
            condition ← (GPR[rs] ≠ GPR[rt])
     T+1:   if condition then
                  PC ← PC + target
            endif
64   T:     target ← $(\text{offset}_{15})^{38}$ || offset || $0^2$
            condition ← (GPR[rs] ≠ GPR[rt])
     T+1:   if condition then
                  PC ← PC + target
            endif
```

Exceptions:

None

BNEL Branch On Not Equal Likely BNEL

31 26	25 21	20 16	15 0
BNEL 010101	rs	rt	offset
6	5	5	16

Format:

BNEL rs, rt, offset

Description:

A 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. The contents of general register *rs* and the contents of general register *rt* are compared. If the two registers are not equal, then the program branches to the target address, with a delay of one instruction.

If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

Operation:

```
32   T:     target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
            condition ← (GPR[rs] ≠ GPR[rt])
     T+1:   if condition then
                 PC ← PC + target
            else
                 NullifyCurrentInstruction
            endif
64   T:     target ← $(\text{offset}_{15})^{38}$ || offset || $0^2$
            condition ← (GPR[rs] ≠ GPR[rt])
     T+1:   if condition then
                 PC ← PC + target
            else
                 NullifyCurrentInstruction
            endif
```

Exceptions:

None

BREAK Breakpoint BREAK

31 26	25 6	5 0
SPECIAL 000000	code	BREAK 001101
6	20	6

Format:

BREAK

Description:

A breakpoint trap occurs, immediately and unconditionally transferring control to the exception handler.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
32, 64    T:    BreakpointException
```

Exceptions:

Breakpoint exception

CACHE Cache CACHE

31 26	25 21	20 16	15 0
CACHE 1 0 1 1 1 1	base	op	offset
6	5	5	16

Format:

CACHE op, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The virtual address is translated to a physical address using the TLB, and the 5-bit sub-opcode specifies a cache operation for that address.

If CP0 is not usable (User or Supervisor mode) the CP0 enable bit in the *Status* register is clear, and a coprocessor unusable exception is taken. The operation of this instruction on any operation/cache combination not listed below, or on a secondary cache when none is present, is undefined. The operation of this instruction on uncached addresses is also undefined.

The Index operation uses part of the virtual address to specify a cache block.

For a primary cache of $2^{CACHESIZE}$ bytes with $2^{BLOCKSIZE}$ bytes per tag, $vAddr_{CACHESIZE..BLOCKSIZE}$ specifies the block. For a secondary cache of $2^{CACHESIZE}$ bytes with $2^{BLOCKSIZE}$ bytes per tag, $pAddr_{CACHESIZE..BLOCKSIZE}$ specifies the block.

Index Load Tag also uses $vAddr_{BLOCKSIZE...3}$ to select the doubleword for reading ECC or parity. When the *CE* bit of the Status register is set, Hit WriteBack, Hit WriteBack Invalidate, Index WriteBack Invalidate, and Fill also use $vAddr_{BLOCKSIZE...3}$ to select the doubleword that has its ECC or parity modified. This operation is performed unconditionally.

The Hit operation accesses the specified cache as normal data references, and performs the specified operation if the cache block contains valid data with the specified physical address (a hit). If the cache block is invalid or contains a different address (a miss), no operation is performed.

CACHE Cache (continued) CACHE

Write back from a primary cache goes to the secondary cache (if there is one), otherwise to memory. Write back from a secondary cache always goes to memory. A secondary write back always writes the most recent data; the data comes from the primary data cache, if present, and modified (the *W* bit is set). Otherwise the data comes from the specified secondary cache. The address to be written is specified by the cache tag and not the translated physical address.

TLB Refill and TLB Invalid exceptions can occur on any operation. For Index operations (where the physical address is used to index the cache but need not match the cache tag) unmapped addresses may be used to avoid TLB exceptions. This operation never causes TLB Modified or Virtual Coherency exceptions. Bits 17..16 of the instruction specify the cache as follows:

Code	Name	Cache
0	I	primary instruction
1	D	primary data
2	SI	secondary instruction
3	SD	secondary data (or combined instruction/data)

Bits 20..18 of the instruction specify the operation as follows:

Code	Caches	Name	Operation
0	I, SI	Index Invalidate	Set the cache state of the cache block to Invalid.
0	D	Index WriteBack Invalidate	Examine the cache state and *W* bit of the primary data cache block at the Invalidate index specified by the virtual address. If the state is not Invalid and the *W* bit is set, then write back the block to the secondary cache (if present) or to memory (if no secondary cache). The address to write is taken from the primary cache tag. When a secondary cache is present, and the *CE* bit of the *Status* register is set, the content of the *ECC* register is XORed into the computed check bits during the write to the secondary cache for the addressed doubleword. Set cache state of primary cache block to Invalid.
0	SD	Index WriteBack Invalidate	Examine the cache state of the secondary data cache block at the index specified by the physical address. If the state is Dirty Exclusive or Dirty Shared, then write back the block to memory and set the cache state to Invalid. The address to write is taken from the secondary cache tag, which is not necessarily the physical address used to index the cache. Like all secondary write-backs, the operation writes any modified data (*W* bit set) from the primary data cache. Unlike Hit Write-back Invalidate the operation does not invalidate or clear the *W* bit in the primary D-cache. In all cases, the secondary cache block state is set to Invalid.
1	all	Index Load Tag	Read the tag for the cache block at the specified index and place it into the *TagLo* and *TagHi* CP0 registers, ignoring ECC and parity errors. Also load the data ECC or parity bits into the *ECC* register.
2	all	Index Store Tag	Write the tag for the cache block at the specified index from the *TagLo* and *TagHi* CP0 registers.

CACHE Cache (continued) CACHE

Code	Caches	Name	Operation
3	SD	Create Dirty Exclusive	This operation is used to avoid loading data needlessly from memory when writing new contents into an entire cache block. If the cache block is valid but does not contain the specified address (a valid miss) the secondary block is vacated. The data is written back to memory if dirty and all matching blocks in both primary caches are invalidated. As usual during a secondary write-back, if the primary data cache contains modified data (matching blocks with *W* bit set) that modified data is written to memory. If the cache block is valid and does contain the specified physical address (a hit), then the operation cleans up the primary caches to avoid any virtual alias problems: all blocks in both primary caches that match the secondary line are invalidated without write back. Note that the search for matching primary blocks uses the virtual index of the PIdx field of the secondary cache tag (the virtual index to the location last used) and not the virtual index of the virtual address used in the operation (the virtual index to the location now being used). If the secondary tag and address do not match (miss), or the tag and address do match (hit) and the block is in a shared state, send an invalidate for the specified address on the system interface. In all cases, set the cache block tag to the specified physical address, set the cache state to Dirty Exclusive, and set the virtual index field from the virtual address. The *CH* bit in the *Status* register is set or cleared to indicate a hit or miss.
3	D	Create Dirty Exclusive	This operation is used to avoid loading data needlessly from secondary cache or memory when writing new contents into an entire cache block. If the cache block does not contain the specified address, and the block is dirty, write it back to the secondary cache or memory. In all cases, set the cache block tag to the specified physical address, set the cache state to Dirty Exclusive.
4	I, D	Hit Invalidate	If the cache block contains the specified address, mark the cache block invalid.
4	SI, SD	Hit Invalidate	If the cache block contains the specified address, mark the cache block invalid and also invalidate all matching blocks, if present, in the primary caches (the PIdx field of the secondary tag is used to determine the locations in the primaries to search). The *CH* bit in the *Status* register is set or cleared to indicate a hit or miss.
5	D	Hit WriteBack Invalidate	If the cache block contains the specified address, write back the data if it is dirty, and mark the cache block invalid. When a secondary cache is present and the *CE* bit of the *Status* register is set, contents of the *ECC* register is XORed into the computed check bits during the write to the secondary cache for the addressed doubleword.

Code	Caches	Name	Operation
5	SD	Hit WriteBack Invalidate	If the cache block contains the specified address, write back the data if it is dirty, and mark the secondary cache block and all matching blocks in both primary caches invalid. As usual with secondary write-backs, modified data in the primary data cache (matching block with the *W* bit set) is used during the write-back. The PIdx field of the secondary tag is used to determine the locations in the primaries to check for matching primary blocks. The *CH* bit in the *Status* register is set or cleared to indicate a hit or miss.
5	I	Fill	Fill the primary instruction cache block from secondary or memory. If the *CE* bit of the *Status* register is set, the contents of the *ECC* register is used instead of the computed parity bits for addressed doubleword when written to the instruction cache.
6	D	Hit WriteBack	If the cache block contains the specified address, and the *W* bit is set, write back the data to memory or the secondary cache, and clear the *W* bit. When a secondary cache is present, and the *CE* bit of the *Status* register is set, the contents of the *ECC* register is XORed into the computed check bits during the write to the secondary cache for the addressed doubleword.
6	SD	Hit WriteBack	If the cache block contains the specified address, and the cache state is Dirty Exclusive or Dirty Shared, write back the data to memory, and change the cache state to Clean Exclusive or Shared, respectively. The *CH* bit in the *Status* register is set or cleared to indicate a hit or miss. The write back looks in the primary data cache for modified data, but does *not* invalidate or clear the *W* bit in the primary data cache. This state, although perhaps not intuitive, is consistent since the primary block contains data that is at least as current as that in memory or secondary cache. A subsequent write-back of the primary line without further modification would be redundant, but not incorrect.
6	I	Hit WriteBack	If the cache block contains the specified address, write back the data unconditionally. When a secondary cache is present, and the *CE* bit of the *Status* register is set, the contents of the *ECC* register is XORed into the computed check bits during the write to the secondary cache for the addressed doubleword.

CACHE Cache (continued) CACHE

Code	Caches	Name	Operation
7	SI, SD	Hit Set Virtual	This operation is used to change the virtual index of secondary cache contents avoiding unnecessary memory operations. If the cache block contains the specified address, invalidate matching blocks in the primary caches at the index formed by concatenating PIdx in the secondary cache tag (not the virtual address of the operation) and $vAddr_{11..4}$, then set the virtual index field of the secondary cache tag from the specified virtual address. Modified data in the primary data cache is not preserved by the operation and should be explicitly written back before this operation. The CH bit in the Status register is set or cleared to indicate a hit or miss.

Operation:

```
32, 64    T:    $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
                $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
```

Exceptions:

Coprocessor unusable exception

CFCz Move Control From Coprocessor CFCz

31 26	25 21	20 16	15 11	10 0
COPz 0 1 0 0 x x*	CF 0 0 0 1 0	rt	rd	0 0 0 0 0 0
6	5	5	5	11

Format:

CFCz rt, rd

Description:

The contents of coprocessor control register *rd* of coprocessor unit *z* are loaded into general register *rt*.

This instruction is not valid for CP0.

Operation:

```
32      T:        data ← CCR[z,rd]
        T + 1:    GPR[rt] ← data

64      T:        data ← $(CCR[z,rd]_{31})^{32}$ || CCR[z.rd]
        T + 1:    GPR[rt] ← data
```

Exceptions:

Coprocessor unusable exception

*Opcode Bit Encoding:

CFCz

Bit #	31	30	29	28	27	26	25	24	23	22	21	0
CFC1	0	1	0	0	0	1	0	0	0	1	0	

Bit #	31	30	29	28	27	26	25	24	23	22	21	0
CFC2	0	1	0	0	1	0	0	0	0	1	0	

Bit #	31	30	29	28	27	26	25	24	23	22	21	0
CFC3	0	1	0	0	1	1	0	0	0	1	0	

Opcode (bits 31–28) | Coprocessor Unit Number (bits 27–26) | Coprocessor Suboperation (bits 25–21)

COPz Coprocessor Operation COPz

31 26	25	24 0
COPz 0 1 0 0 x x*	CO 1	cofun
6	1	25

Format:

COPz cofun

Description:

A coprocessor operation is performed. The operation may specify and reference internal coprocessor registers, and may change the state of the coprocessor condition line, but does not modify state within the processor or the cache/memory system. Details of coprocessor operations are contained in Appendix B.

Operation:

32, 64	T:	CoprocessorOperation (z, co-

Exceptions:

Coprocessor unusable exception
Coprocessor interrupt or Floating-Point Exception (R4000 CP1 only)

*Opcode Bit Encoding:

COPz

Bit #	31	30	29	28	27	26	25	0
COP0	0	1	0	0	0	0	1	
COP1	0	1	0	0	0	1	1	
COP2	0	1	0	0	1	0	1	
COP3	0	1	0	0	1	1	1	

Opcode (bits 31–28)
Coprocessor Unit Number (bits 27–26)
CO sub-opcode (see end of Appendix A) (bit 25)

CTCz Move Control toCoprocessor CTCz

31 26	25 21	20 16	15 11	10 0
COPz 0 1 0 0 x x *	CT 0 0 1 1 0	rt	rd	0 000 0000 0000
6	5	5	5	11

Format:

CTCz rt, rd

Description:

The contents of general register *rt* are loaded into control register *rd* of coprocessor unit z.

This instruction is not valid for CP0.

Operation:

```
32,64    T:      data ← GPR[rt]
         T + 1:  CCR[z,rd] ← data
```

Exceptions:

Coprocessor unusable

*See "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

DADD Doubleword Add DADD

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	DADD 1 0 1 1 0 0
6	5	5	5	5	6

Format:

DADD rd, rs, rt

Description:

The contents of general register *rs* and the contents of general register *rt* are added to form the result. The result is placed into general register *rd*.

An overflow exception occurs if the carries out of bits 62 and 63 differ (2's-complement overflow). The destination register *rd* is not modified when an integer overflow exception occurs.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
64    T:    GPR[rd] ←GPR[rs] + GPR[rt]
```

Exceptions:

Integer overflow exception

Reserved instruction exception (R4000 in 32-bit mode)

DADDI Doubleword Add Immediate DADDI

31 26	25 21	20 16	15 0
DADDI 0 1 1 0 0 0	rs	rt	immediate
6	5	5	16

Format:

DADDI rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and added to the contents of general register *rs* to form the result. The result is placed into general register *rt*.

An overflow exception occurs if carries out of bits 62 and 63 differ (2's-complement overflow). The destination register *rt* is not modified when an integer overflow exception occurs.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
64    T:    GPR [rt] ← GPR[rs] + $(\text{immediate}_{15})^{48}$ || $\text{immediate}_{15..0}$
```

Exceptions:

Integer overflow exception
Reserved instruction exception (R4000 in 32-bit mode)

DADDIU Doubleword Add Immediate Unsigned DADDIU

31 26	25 21	20 16	15 0
DADDIU 011001	rs	rt	immediate
6	5	5	16

Format:

DADDIU rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and added to the contents of general register *rs* to formthe result. The result is placed into general register *rt*. No integer overflow exception occurs under any circumstances.

The only difference between this instruction and the DADDI instruction is that DADDIU never causes an overflow exception.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
64    T:    $GPR [rt] \leftarrow GPR[rs] + (immediate_{15})^{48} \| immediate_{15..0}$
```

Exceptions:

Reserved instruction exception (R4000 in 32-bit mode)

DADDU Doubleword Add Unsigned DADDU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DADDU 101101
6	5	5	5	5	6

Format:

DADDU rd, rs, rt

Description:

The contents of general register *rs* and the contents of general register *rt* are added to form the result. The result is placed into general register *rd*.

No overflow exception occurs under any circumstances.

The only difference between this instruction and the DADD instruction is that DADDU never causes an overflow exception.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
    64     T:     GPR[rd] ←GPR[rs] + GPR[rt]
```

Exceptions:

Reserved instruction exception (R4000 in 32-bit mode)

DDIV Doubleword Divide DDIV

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 00 0000 0000	DDIV 011110
6	5	5	10	6

Format:

DDIV rs, rt

Description:

The contents of general register *rs* are divided by the contents of general register *rt*, treating both operands as 2's-complement values. No overflow exception occurs under any circumstances, and the result of this operation is undefined when the divisor is zero.

This instruction is typically followed by additional instructions to check for a zero divisor and for overflow.

When the operation completes, the quotient word of the double result is loaded into special register *LO*, and the remainder word of the double result is loaded into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. Correct operation requires separating reads of *HI* or *LO* from writes by two or more instructions.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
64    T–2:    LO    ← undefined
              HI    ← undefined
      T–1:    LO    ← undefined
              HI    ← undefined
      T:      LO    ← GPR[rs] div GPR[rt]
              HI    ← GPR[rs] mod GPR[rt]
```

Exceptions:

Reserved instruction exception (R4000 in 32-bit mode)

DDIVU Doubleword Divide Unsigned DDIVU

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 000000 0000	DDIVU 011111
6	5	5	10	6

Format:

DDIVU rs, rt

Description:

The contents of general register *rs* are divided by the contents of general register *rt,* treating both operands as unsigned values. No integer overflow exception occurs under any circumstances, and the result of this operation is undefined when the divisor is zero.

This instruction is typically followed by additional instructions to check for a zero divisor.

When the operation completes, the quotient word of the double result is loaded into special register *LO,* and the remainder word of the double result is loaded into special register *HI.*

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. Correct operation requires separating reads of *HI* or *LO* from writes by two or more instructions.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
64    T–2:    LO    ← undefined
              HI    ← undefined
      T–1:    LO    ← undefined
              HI    ← undefined
      T:      LO    ← (0 || GPR[rs]) div  (0 || GPR[rt])
              HI    ← (0 || GPR[rs]) mod  (0 || GPR[rt])
```

Exceptions:

Reserved instruction exception (R4000 in 32-bit mode)

DIV Divide DIV

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 00 0000 0000	DIV 011010
6	5	5	10	6

Format:

DIV rs, rt

Description:

The contents of general register *rs* are divided by the contents of general register *rt*, treating both operands as 2's-complement values. No overflow exception occurs under any circumstances, and the result of this operation is undefined when the divisor is zero. In 64-bit mode, the operands must be valid sign-extended, 32-bit values.

This instruction is typically followed by additional instructions to check for a zero divisor and for overflow.

When the operation completes, the quotient word of the double result is loaded into special register *LO*, and the remainder word of the double result is loaded into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. Correct operation requires separating reads of *HI* or *LO* from writes by two or more instructions.

DIV Divide (continued) DIV

Operation:

```
32   T–2:   LO   ← undefined
            HI   ← undefined
     T–1:   LO   ← undefined
            HI   ← undefined
     T:     LO   ← GPR[rs] div GPR[rt]
            HI   ← GPR[rs] mod GPR[rt]
64   T–2:   LO   ← undefined
            HI   ← undefined
     T–1:   LO   ← undefined
            HI   ← undefined
     T:     q    ← $GPR[rs]_{31..0}$ div $GPR[rt]_{31..0}$
            r    ← $GPR[rs]_{31..0}$ mod $GPR[rt]_{31..0}$
            LO   ← $(q_{31})^{32} \| q_{31..0}$
            HI   ← $(r_{31})^{32} \| r_{31..0}$
```

Exceptions:

None

DIVU Divide Unsigned DIVU

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 000000 0000	DIVU 011011
6	5	5	10	6

Format:

DIVU rs, rt

Description:

The contents of general register *rs* are divided by the contents of general register *rt*, treating both operands as unsigned values. No integer overflow exception occurs under any circumstances, and the result of this operation is undefined when the divisor is zero. In 64-bit mode, the operands must be valid sign-extended, 32-bit values. In 64-bit mode, the operands must be valid sign-extended, 32-bit values.

This instruction is typically followed by additional instructions to check for a zero divisor.

When the operation completes, the quotient word of the double result is loaded into special register *LO*, and the remainder word of the double result is loaded into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. Correct operation requires separating reads of *HI* or *LO* from writes by two or more instructions.

DIVU Divide Unsigned (continued) DIVU

Operation:

```
32    T–2:    LO    ← undefined
              HI    ← undefined
      T–1:    LO    ← undefined
              HI    ← undefined
      T:      LO    ← (0 || GPR[rs]) div (0 || GPR[rt])
              HI    ← (0 || GPR[rs]) mod (0 || GPR[rt])
64    T–2:    LO    ← undefined
              HI    ← undefined
      T–1:    LO    ← undefined
              HI    ← undefined
      T:      q     ← $(0 \| GPR[rs]_{31..0})$ div $(0 \| GPR[rt]_{31..0})$
              r     ← $(0 \| GPR[rs]_{31..0})$ mod $(0 \| GPR[rt]_{31..0})$
              LO    ← $(q_{31})^{32} \| q_{31..0}$
              HI    ← $(r_{31})^{32} \| r_{31..0}$
```

Exceptions:

None

DMFC0 Doubleword Move From System Control Coprocessor DMFC0

31 26	25 21	20 16	15 11	10 0
COP0 0 1 0 0 0 0	DMF 0 0 0 0 1	rt	rd	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format:

DMFC0 rt, rd

Description:

The contents of coprocessor register *rd* of the CP0 are loaded into general register *rt*.

This operation is defined for the R4000 operating in 64-bit mode and in 32-bit kernal mode. Execution of this instruction in 32-bit user or supervisor mode causes a reserved instruction exception. All 64-bits of the general regiser destination are written from the coprocessor register source. The operation of DMFC0 on a 32-bit coprocessor 0 register is undefined.

Operation:

```
64      T:        data ←CPR[0,rd]
        T+1:      GPR[rt] ← data
```

Exceptions:

Coprocessor unusable exception
Reserved instruction exception (R4000 in 32-bit user mode
R4000 in 32-bit supervisor mode)

DMTC0 Doubleword Move To System Control Coprocessor DMTC0

31 26	25 21	20 16	15 11	10 0
COP0 0 1 0 0 0 0	DMT 0 0 1 0 1	rt	rd	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format:

DMTC0 rt, rd

Description:

The contents of general register *rt* are loaded into coprocessor register *rd* of the CP0.

This operation is defined for the R4000 operating in 64-bit mode or in 32-bit kernal mode. Execution of this instruction in 32-bit user or supervisor mode causes a reserved instruction exception. All 64-bits of the coprocessor 0 regiser are written from the general register source. The operation of DMTC0 on a 32-bit coprocessor 0 register is undefined.

Because the state of the virtual address translation system may be altered by this instruction, the operation of load, store instructions and TLB operations immediately prior to and after this instruction are undefined.

Operation:

```
64   T:    data ← GPR[rt]
     T+1:  CPR[0,rd] ← data
```

Exceptions:

Coprocessor unusable exception (R4000 in 32-bit user mode
R4000 in 32-bit supervisor mode)

DMULT Doubleword Multiply DMULT

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 00 0000 0000	DMULT 011100
6	5	5	10	6

Format:

DMULT rs, rt

Description:

The contents of general registers *rs* and *rt* are multiplied, treating both operands as 2's-complement values. No integer overflow exception occurs under any circumstances.

When the operation completes, the low-order word of the double result is loaded into special register *LO*, and the high-order word of the double result is loaded into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results of these instructions are undefined. Correct operation requires separating reads of *HI* or *LO* from writes by a minimum of two other instructions.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
64   T–2:   LO ← undefined
            HI ← undefined
     T–1:   LO ← undefined
            HI ← undefined
     T:     t ← GPR[rs] * GPR[rt]
            LO ← $t_{63..0}$
            H I← $t_{127..64}$
```

Exceptions:

Reserved instruction exception (R4000 in 32-bit mode)

DMULTU Doubleword Multiply Unsigned DMULTU

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 00 0000 0000	DMULTU 011101
6	5	5	10	6

Format:

DMULTU rs, rt

Description:

The contents of general register *rs* and the contents of general register *rt* are multiplied, treating both operands as unsigned values. No overflow exception occurs under any circumstances.

When the operation completes, the low-order word of the double result is loaded into special register *LO*, and the high-order word of the double result is loaded into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results of these instructions are undefined. Correct operation requires separating reads of *HI* or *LO* from writes by a minimum of two instructions.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
32    T–2:   LO ← undefined
             HI ← undefined
      T–1:   LO ← undefined
             HI ← undefined
      T:     t ← (0 || GPR[rs]) * (0 || GPR[rt])
             LO ← $t_{63..0}$
             HI ← $t_{127..64}$
```

Exceptions:

Reserved instruction exception (R4000 in 32-bit mode)

DSLL Doubleword Shift Left Logical DSLL

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSLL 111000
6	5	5	5	5	6

Format:

DSLL rd, rt, sa

Description:

The contents of general register *rt* are shifted left by *sa* bits, inserting zeros into the low-order bits. The result is placed in register *rd*.

Operation:

```
64    T:    $s \leftarrow 0 \,||\, sa$
            $GPR[rd] \leftarrow GPR[rt]_{63-s..0} \,||\, 0^{s}$
```

Exceptions:

Reserved instruction exception (R4000 in 32-bit mode)

DSLLV Doubleword Shift Left Logical Variable DSLLV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DSLLV 010100
6	5	5	5	5	6

Format:

DSLLV rd, rt, rs

Description:

The contents of general register *rt* are shifted left by the number of bits specified by the low-order six bits contained as contents of general register *rs*, inserting zeros into the low-order bits. The result is placed in register *rd*.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
64    T:    $s \leftarrow GP[rs]_{5..0}$
            $GPR[rd] \leftarrow GPR[rt]_{(63-s)..0} \| 0^s$
```

Exceptions:

Reserved instruction exception (R4000 in 32-bit mode)

DSLL32 Doubleword Shift Left Logical + 32 DSLL32

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSLL32 111100
6	5	5	5	5	6

Format:

DSLL32 rd, rt, sa

Description:

The contents of general register *rt* are shifted left by *32+sa* bits, inserting zeros into the low-order bits. The result is placed in register *rd*.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
64   T:   s ← 1 || sa
          GPR[rd]← $\text{GPR[rt]}_{63-s..0}$ || $0^s$
```

Exceptions:

Reserved instruction exception (R4000 in 32-bit mode)

DSRA Doubleword Shift Right Arithmetic DSRA

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSRA 111011
6	5	5	5	5	6

Format:

DSRA rd, rt, sa

Description:

The contents of general register *rt* are shifted right by *sa* bits, sign-extending the high-order bits. The result is placed in register *rd*.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
64    T:   $s \leftarrow 0 \,||\, sa$
           $GPR[rd] \leftarrow (GPR[rt]_{63})^{s} \,||\, GPR[rt]_{63..s}$
```

Exceptions:

Reserved instruction exception (R4000 in 32-bit mode)

DSRAV Doubleword Shift Right Arithmetic Variable DSRAV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DSRAV 010111
6	5	5	5	5	6

Format:

DSRAV rd, rt, rs

Description:

The contents of general register *rt* are shifted right by the number of bits specified by the low-order six bits of general register *rs*, sign-extending the high-order bits. The result is placed in register *rd*.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
64    T:    $s \leftarrow GPR[rs]_{5..0}$
            $GPR[rd] \leftarrow (GPR[rt]_{63})^{s} \,||\, GPR[rt]_{63..s}$
```

Exceptions:

Reserved instruction exception (R4000 in 32-bit mode)

DSRA32 Doubleword Shift Right Arithmetic + 32 DSRA32

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSRA32 111111
6	5	5	5	5	6

Format:

DSRA32 rd, rt, sa

Description:

The contents of general register *rt* are shifted right by *32+sa* bits, sign-extending the high-order bits. The result is placed in register *rd*.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
64    T:    s ← 1 || sa
            GPR[rd] ← (GPR[rt]_63)^s || GPR[rt]_63..s
```

Exceptions:

Reserved instruction exception (R4000 in 32-bit mode)

DSRL Doubleword Shift Right Logical DSRL

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSRL 111010
6	5	5	5	5	6

Format:

DSRL rd, rt, sa

Description:

The contents of general register *rt* are shifted right by *sa* bits, inserting zeros into the high-order bits. The result is placed in register *rd*.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
64    T:   $s \leftarrow 0 \,||\, sa$
           $GPR[rd] \leftarrow 0^{s} \,||\, GPR[rt]_{63..s}$
```

Exceptions:

Reserved instruction exception (R4000 in 32-bit mode)

DSRLV — Doubleword Shift Right Logical Variable — DSRLV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DSRLV 010110
6	5	5	5	5	6

Format:

DSRLV rd, rt, rs

Description:

The contents of general register *rt* are shifted right by the number of bits specified by the low-order six bits of general register *rs*, inserting zeros into the high-order bits. The result is placed in register *rd*.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
64    T:    $s \leftarrow GPR[rs]_{5..0}$
            $GPR[rd] \leftarrow 0^{s} \, || \, GPR[rt]_{63..s}$
```

Exceptions:

Reserved instruction exception (R4000 in 32-bit mode)

DSRL32 Doubleword Shift Right Logical + 32 DSRL32

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSRL32 111110
6	5	5	5	5	6

Format:

DSRL32 rd, rt, sa

Description:

The contents of general register *rt* are shifted right by *32+sa* bits, inserting zeros into the high-order bits. The result is placed in register *rd*.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
64    T:   $s \leftarrow 1 \| sa$
           $GPR[rd] \leftarrow 0^{s} \| GPR[rt]_{63..s}$
```

Exceptions:

Reserved instruction exception (R4000 in 32-bit mode)

DSUB Doubleword Subtract DSUB

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DSUB 101110
6	5	5	5	5	6

Format:

DSUB rd, rs, rt

Description:

The contents of general register *rt* are subtracted from the contents of general register *rs* to form a result. The result is placed into general register *rd*.

The only difference between this instruction and the DSUBU instruction is that DSUBU never traps on overflow.

An integer overflow exception takes place if the carries out of bits 62 and 63 differ (2's-complement overflow). The destination register *rd* is not modified when an integer overflow exception occurs.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
    64     T:    GPR[rd] ← GPR[rs] – GPR[rt]
```

Exceptions:

Integer overflow exception
Reserved instruction exception (R4000 in 32-bit mode)

DSUBU Doubleword Subtract Unsigned DSUBU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DSUBU 101111
6	5	5	5	5	6

Format:

DSUBU rd, rs, rt

Description:

The contents of general register *rt* are subtracted from the contents of general register *rs* to form a result. The result is placed into general register *rd*.

The only difference between this instruction and the DSUB instruction is that DSUBU never traps on overflow. No integer overflow exception occurs under any circumstances.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
    64    T:    GPR[rd] ← GPR[rs] – GPR[rt]
```

Exceptions:

Reserved instruction exception (R4000 in 32-bit mode)

ERET Exception Return ERET

31 26	25	24 6	5 0
COP0 0 1 0 0 0 0	CO 1	0 000 0000 0000 0000 0000	ERET 0 1 1 0 0 0
6	1	19	6

Format:

ERET

Description:

ERET is the R4000 instruction for returning from an interrupt, exception, or error trap. Unlike a branch or jump instruction, ERET does not execute the next instruction.

ERET must not itself be placed in a branch delay slot.

If the processor is servicing an error trap ($SR_2 = 1$), then load the PC from the *ErrorEPC* and clear the *ERL* bit of the *Status* register (SR_2). Otherwise ($SR_2 = 0$), load the PC from the *EPC*, and clear the *EXL* bit of the *Status* register (SR_1).

An ERET executed between a LL and SC also causes the SC to fail.

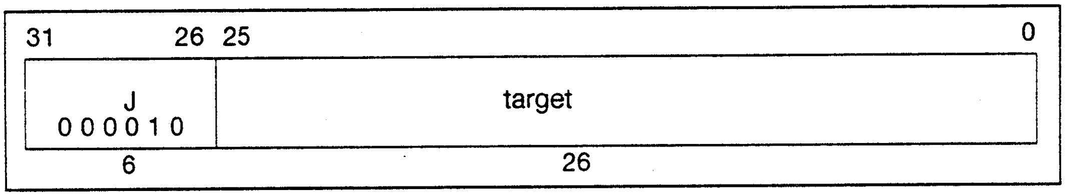
Operation:

```
32, 64   T:   if $SR_2 = 1$ then
                   PC ← ErrorEPC
                   SR ← $SR_{31..3}$ || 0 || $SR_{1..0}$
              else
                   PC ← EPC
                   SR ← $SR_{31..2}$ || 0 || $SR_0$
              endif
              LLbit ← 0
```

Exceptions:

Coprocessor unusable exception

J Jump J



Format:

J target

Description:

The 26-bit target address is shifted left two bits and combined with the high-order bits of the address of the delay slot. The program unconditionally jumps to this calculated address with a delay of one instruction.

Operation:

```
32   T:    $temp \leftarrow target$
     T+1:  $PC \leftarrow PC_{31..28} \,||\, temp \,||\, 0^2$

64   T:    $temp \leftarrow target$
     T+1:  $PC \leftarrow PC_{63..28} \,||\, temp \,||\, 0^2$
```

Exceptions:

None

JAL Jump And Link JAL

31 26	25 0
JAL 0 0 0 0 1 1	target
6	26

Format:

JAL target

Description:

The 26-bit target address is shifted left two bits and combined with the high-order bits of the address of the delay slot. The program unconditionally jumps to this calculated address with a delay of one instruction. The address of the instruction after the delay slot is placed in the link register, *r31*.

Operation:

```
32    T:     temp ← target
             GPR[31] ← PC + 8
      T+1:   PC ← $PC_{31..28}$ || temp || $0^2$

64    T:     temp ← target
             GPR[31] ← PC + 8
      T+1:   PC ← $PC_{63..28}$ || temp || $0^2$
```

Exceptions:

None

JALR Jump And Link Register JALR

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	0 00000	rd	0 00000	JALR 001001
6	5	5	5	5	6

Format:

JALR rs
JALR rd, rs

Description:

The program unconditionally jumps to the address contained in general register *rs*, with a delay of on instruction. The address of the instruction after the delay slot is placed in general register *rd*. The default value of *rd*, if omitted in the assembly language instruction, is 31.

Register specifiers *rs* and *rd* may not be equal, because such an instruction does not have the same effect when reexecuted. However, an attempt to execute this instruction is *not* trapped, and the result of executing such an instruction is undefined.

Since instructions must be word-aligned, a *Jump and Link Register* instruction must specify a target register (rs) whose two low-order bits are zero. If these low-order bits are not zero, an address exception will occur when the jump target instruction is subsequently fetched.

Operation:

```
32, 64     T:      temp ← GPR [rs]
                   GPR[rd] ← PC + 8
           T+1:    PC ← temp
```

Exceptions:

None

JR Jump Register JR

31 26	25 21	20 6	5 0
SPECIAL 000000	rs	0 000 0000 0000 0000	JR 001000
6	5	15	6

Format:

JR rs

Description:

The program unconditionally jumps to the address contained in general register *rs*, with a delay of one instruction.

Since instructions must be word-aligned, a *Jump Register* instruction must specify a target register (*rs*) whose two low-order bits are zero. If these low-order bits are not zero, an address exception will occur when the jump target instruction is subsequently fetched.

Operation:

```
32, 64   T:     temp ← GPR[rs]
         T+1:   PC ← temp
```

Exceptions:

None

LB Load Byte LB

31 26	25 21	20 16	15 0
LB 100000	base	rt	offset
6	5	5	16

Format:

LB rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the byte at the memory location specified by the effective address are sign-extended and loaded into general register *rt*.

Operation:

```
32   T:   $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1..2} \| (pAddr_{1..0}\ xor\ ReverseEndian^2)$
          $mem \leftarrow LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)$
          $byte \leftarrow vAddr_{1..0}\ xor\ BigEndianCPU^2$
          $GPR[rt] \leftarrow (mem_{7+8*byte})^{24} \| mem_{7+8*byte..8*byte}$
64   T:   $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0}\ xor\ ReverseEndian^3)$
          $mem \leftarrow LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)$
          $byte \leftarrow vAddr_{2..0}\ xor\ BigEndianCPU^3$
          $GPR[rt] \leftarrow (mem_{7+8*byte})^{56} \| mem_{7+8*byte..8*byte}$
```

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

LBU   Load Byte Unsigned   LBU

31 26	25 21	20 16	15 0
LBU 100100	base	rt	offset
6	5	5	16

Format:

LBU rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the byte at the memory location specified by the effective address are zero-extended and loaded into general register *rt*.

Operation:

```
32  T:  $vAddr \leftarrow ((offset_{15})^{16} \,||\, offset_{15..0}) + GPR[base]$
        $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
        $pAddr \leftarrow pAddr_{PSIZE-1..2} \,||\, (pAddr_{1..0}\ xor\ ReverseEndian^{2})$
        $mem \leftarrow LoadMemory\ (uncached, BYTE, pAddr, vAddr, DATA)$
        $byte \leftarrow vAddr_{1..0}\ xor\ BigEndianCPU^{2}$
64  T:  $vAddr \leftarrow ((offset_{15})^{48} \,||\, offset_{15..0}) + GPR[base]$
        $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
        $pAddr \leftarrow pAddr_{PSIZE-1..3} \,||\, (pAddr_{2..0}\ xor\ ReverseEndian^{3})$
        $mem \leftarrow LoadMemory\ (uncached, BYTE, pAddr, vAddr, DATA)$
        $byte \leftarrow vAddr_{2..0}\ xor\ BigEndianCPU^{3}$
```

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

LD Load Doubleword LD

31 26	25 21	20 16	15 0
LD 110111	base	rt	offset
6	5	5	16

Format:

LD rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the 64-bit doubleword at the memory location specified by the effective address are loaded into general register *rt*.

If any of the three least-significant bits of the effective address are non-zero, an address error exception occurs.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

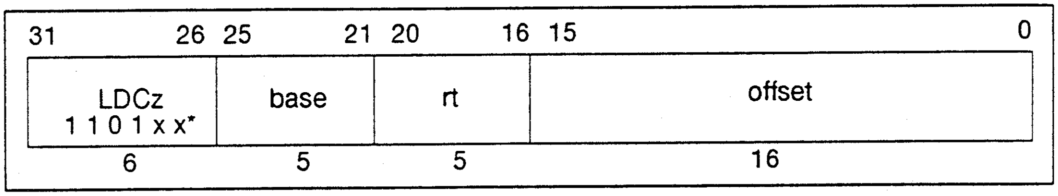
Operation:

```
64   T:   vAddr ← $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15..0})$ + GPR[base]
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
          GPR[rt] ← mem
```

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception
Reserved instruction exception (R4000 in 32-bit user mode
R4000 in 32-bit supervisor mode)

LDCz Load Doubleword To Coprocessor LDCz



Format:

LDCz rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The processor reads a doubleword from the addressed memory location and makes the data available to coprocessor unit *z*. The manner in which each coprocessor uses the data is defined by the individual coprocessor specifications.

If any of the three least-significant bits of the effective address are non-zero, an address error exception takes place.

This instruction is not valid for use with CP0.

This instruction is undefined when the least-significant bit of the *rt-field* is non-zero.

Execution of the instruction referencing coprocessor 3 causes a reserved instruction exception, not a coprocessor unusable exception.

*See the table "Opcode Bit Encoding" on next page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

LDCz — Load Doubleword To Coprocessor (continued) — LDCz

Operation:

```
32    T:   vAddr ← $((\text{offset}_{15})^{16} \,||\, \text{offset}_{15..0}) + \text{GPR}[\text{base}]$
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
           COPzLD (rt, mem)

64    T:   vAddr ← $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15..0}) + \text{GPR}[\text{base}]$
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
           COPzLD (rt, mem)
```

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception
Coprocessor unusable exception
Reserved instruction exception (coprocessor 3)

Opcode Bit Encoding:

LDCz

	Bit # 31	30	29	28	27	26	0
LDC1	1	1	0	1	0	1	
LDC2	1	1	0	1	1	0	
LDC3	1	1	0	1	1	1	

Opcode Coprocessor Unit Number

LDL Load Doubleword Left LDL

31 26	25 21	20 16	15 0
LDL 011010	base	rt	offset
6	5	5	16

Format:

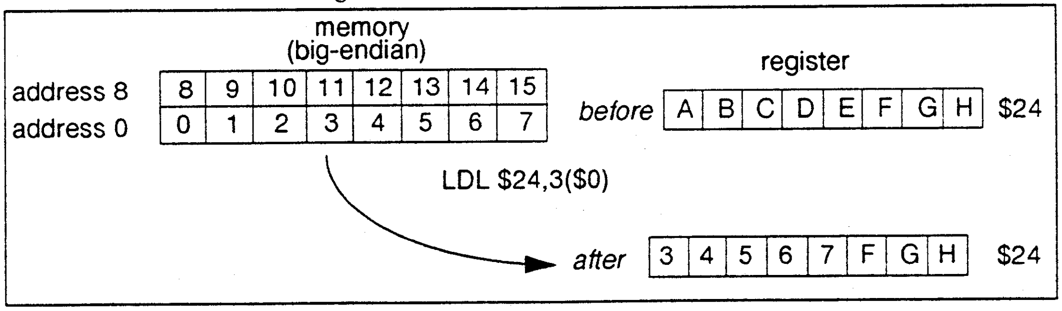
LDL rt, offset(base)

Description:

This instruction can be used in combination with the LDR instruction to load a register with eight consecutive bytes from memory, when the bytes cross a boundary between two doublewords. LDL loads the left portion of the register from the appropriate part of the high-order doubleword; LDR loads the right portion of the register from the appropriate part of the low-order doubleword.

The LDL instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which can specify an arbitrary byte. It reads bytes only from the doubleword in memory which contains the specified starting byte. From one to eight bytes will be loaded, depending on the starting byte specified.

Conceptually, it starts at the specified byte in memory and loads that byte into the high-order (left-most) byte of the register; then it proceeds toward the low-order byte of the doubleword in memory and the low-order byte of the register, loading bytes from memory into the register until it reaches the low-order byte of the doubleword in memory. The least-significant (right-most) byte(s) of the register will not be changed.



LDL Load Doubleword Left (continued) LDL

The contents of general register *rt* are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register *rt* and a following LDL (or LDR) instruction which also specifies register *rt*.

No address exceptions due to alignment are possible.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
64    T:   vAddr ← $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15..0}) + \text{GPR[base]}$
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← $\text{pAddr}_{\text{PSIZE}-1..3} \,||\, (\text{pAddr}_{2..0}\ \textit{xor}\ \text{ReverseEndian}^3)$
           if BigEndianMem = 0 then
                 pAddr ← $\text{pAddr}_{\text{PSIZE}-1..3} \,||\, 0^3$
           endif
           byte ¨ $\text{vAddr}_{2..0}\ \textit{xor}\ \text{BigEndianCPU}^3$
           mem ← LoadMemory (uncached, byte, pAddr, vAddr, DATA)
           GPR[rt] ← $\text{mem}_{7+8*\text{byte}..0} \,||\, \text{GPR[rt]}_{63-8*\text{byte}..0}$
```

LDL — Load Doubleword Left (continued) — LDL

Given a doubleword in a register and a doubleword in memory, the operation of LDL is as follows:

LDL

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

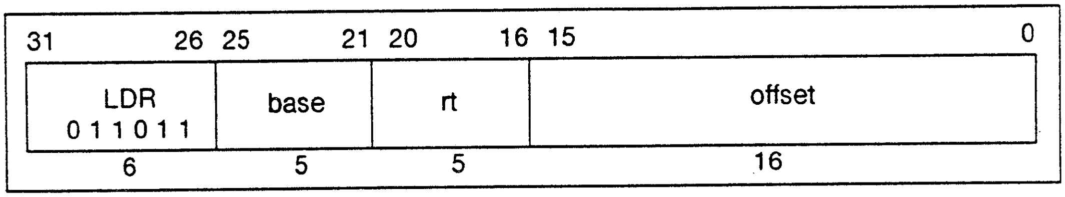
vAddr$_{2..0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	destination	type	offset		destination	type	offset	
			LEM	BEM			LEM	BEM
0	P B C D E F G H	0	0	7	I J K L M N O P	7	0	0
1	O P C D E F G H	1	0	6	J K L M N O P H	6	0	1
2	N O P D E F G H	2	0	5	K L M N O P G H	5	0	2
3	M N O P E F G P	3	0	4	L M N O P F G H	4	0	3
4	L M N O P F G H	4	0	3	M N O P E F G H	3	0	4
5	K L M N O P G H	5	0	2	N O P D E F G H	2	0	5
6	J K L M N O P H	6	0	1	O P C D E F G H	1	0	6
7	I J K L M N O P	7	0	0	P B C D E F G H	0	0	7

LEM	BigEndianMem = 0
BEM	BigEndianMem = 1
Type	AccessType (see Figure 2-2) sent to memory
Offset	pAddr$_{2..0}$ sent to memory

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception
Reserved instruction exception (R4000 in 32-bit mode)

LDR Load Doubleword Right LDR



Format:

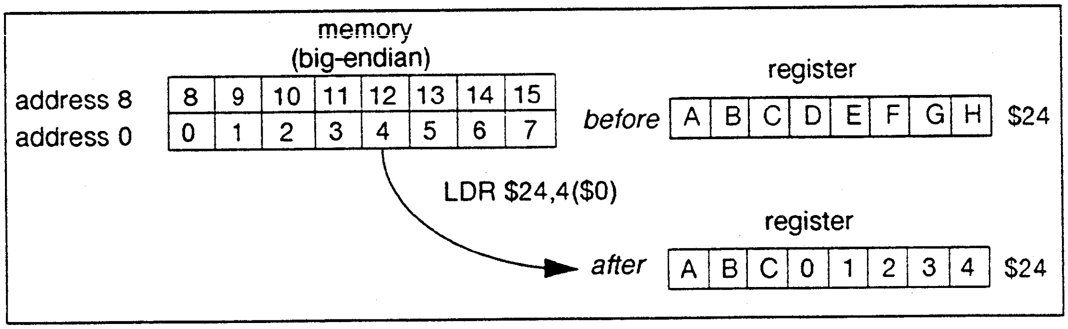
LDR rt, offset(base)

Description:

This instruction can be used in combination with the LDL instruction to load a register with eight consecutive bytes from memory, when the bytes cross a boundary between two doublewords. LDR loads the right portion of the register from the appropriate part of the low-order doubleword; LDL loads the left portion of the register from the appropriate part of the high-order doubleword.

The LDR instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which can specify an arbitrary byte. It reads bytes only from the doubleword in memory which contains the specified starting byte. From one to eight bytes will be loaded, depending on the starting byte specified.

Conceptually, it starts at the specified byte in memory and loads that byte into the low-order (right-most) byte of the register; then it proceeds toward the high-order byte of the doubleword in memory and the high-order byte of the register, loading bytes from memory into the register until it reaches the high-order byte of the doubleword in memory. The most significant (left-most) byte(s) of the register will not be changed.



LDR Load Doubleword Right (continued) LDR

The contents of general register *rt* are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register *rt* and a following LDR (or LDL) instruction which also specifies register *rt*.

No address exceptions due to alignment are possible.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
64    T:    vAddr ← $((\text{offset}_{15})^{48} \| \text{offset}_{15..0})$ + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← $\text{pAddr}_{\text{PSIZE}-1..3} \| (\text{pAddr}_{2..0}\ \textit{xor}\ \text{ReverseEndian}^{3})$
            if BigEndianMem = 1 then
                  pAddr ← $\text{pAddr}_{31..3} \| 0^{3}$
            endif
            byte ← $\text{vAddr}_{2..0}\ \textit{xor}\ \text{BigEndianCPU}^{3}$
            mem ← LoadMemory (uncached, byte, pAddr, vAddr, DATA)
            GPR[rt] ← $\text{mem}_{63..64-8*\text{byte}..0} \| \text{GPR[rt]}_{63-8*\text{byte}..0}$
```

LDR Load Doubleword Right (continued) LDR

Given a doubleword in a register and a doubleword in memory, the operation of LDR is as follows:

LDR

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

$vAddr_{2..0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	destination	type	offset		destination	type	offset	
			LEM	BEM			LEM	BEM
0	I J K L M N O P	7	0	0	A B C D E F G I	0	7	0
1	A I J K L M N O	6	1	0	A B C D E F I J	1	6	0
2	A B I J K L M N	5	2	0	A B C D E I J K	2	5	0
3	A B C I J K L M	4	3	0	A B C D I J K L	3	4	0
4	A B C D I J K L	3	4	0	A B C I J K L M	4	3	0
5	A B C D E I J K	2	5	0	A B I J K L M N	5	2	0
6	A B C D E F I J	1	6	0	A I J K L M N O	6	1	0
7	A B C D E F G I	0	7	0	I J K L M N O P	7	0	0

LEM	BigEndianMem = 0
BEM	BigEndianMem = 1
Type	AccessType (see Figure 2-2) sent to memory
Offset	$pAddr_{2..0}$ sent to memory

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception
Reserved instruction exception (R4000 in 32-bit mode)

LH Load Halfword LH

31 26	25 21	20 16	15 0
LH 100001	base	rt	offset
6	5	5	16

Format:

LH rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the halfword at the memory location specified by the effective address are sign-extended and loaded into general register *rt*.

If the least-significant bit of the effective address is non-zero, an address error exception occurs.

Operation:

```
32  T:  $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
        $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
        $pAddr \leftarrow pAddr_{PSIZE-1..2} \| (pAddr_{1..0}\ \mathit{xor}\ (ReverseEndian \| 0))$
        $mem \leftarrow LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)$
        $byte \leftarrow vAddr_{1..0}\ \mathit{xor}\ (BigEndianCPU \| 0)$
        $GPR[rt] \leftarrow (mem_{15+8*byte})^{16} \| mem_{15+8*byte..8*byte}$
64  T:  $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
        $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
        $pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0}\ xor\ (ReverseEndian \| 0))$
        $mem \leftarrow LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)$
        $byte \leftarrow vAddr_{2..0}\ \mathit{xor}\ (BigEndianCPU^{2} \| 0)$
        $GPR[rt] \leftarrow (mem_{15+8*byte})^{16} \| mem_{15+8*byte..8*byte}$
```

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

LHU Load Halfword Unsigned LHU

31 26	25 21	20 16	15 0
LHU 1 0 0 1 0 1	base	rt	offset
6	5	5	16

Format:

LHU rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the halfword at the memory location specified by the effective address are zero-extended and loaded into general register *rt*.

If the least-significant bit of the effective address is non-zero, an address error exception occurs.

Operation:

```
32  T:  $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
        $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
        $pAddr \leftarrow pAddr_{PSIZE-1..2} \| ( pAdd_{r1..0}$ xor $(ReverseEndian \| 0))$
        $mem \leftarrow LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)$
        $byte \leftarrow vAddr_{1..0}$ xor $(BigEndianCPU \| 0)$
        $GPR[rt] \leftarrow 0^{16} \| mem_{15+8*byte..8*byte}$

64  T:  $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
        $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
        $pAddr \leftarrow pAddr_{PSIZE-1..3} \| ( pAddr_{2..0}$ xor $(ReverseEndian^2 \| 0))$
        $mem \leftarrow LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)$
        $byte \leftarrow vAddr_{2..0}$ xor $(BigEndianCPU^2 \| 0)$
        $GPR[rt] \leftarrow 0^{48} \| mem_{15+8*byte..8*byte}$
```

Exceptions:

TLB refill exception

TLB invalid exception

Bus Error exception

Address error exception

LL Load Linked LL

31 26	25 21	20 16	15 0
LL 110000	base	rt	offset
6	5	5	16

Format:

LL rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the word at the memory location specified by the effective address are loaded into general register *rt*. In 64-bit mode, the loaded word is sign-extended.

This instruction implicitly performs a SYNC operation; all loads and stores to shared memory fetched prior to the LL must access memory before the LL, and loads and stores to shared memory fetched subsequent to the LL must access memory after the LL.

The processor begins checking the accessed word for modification by other processors and devices.

Load Linked and Store Conditional can be used to atomically update memory locations:

```
L1:
    LL      T1, (T0)
    ADD     T2, T1, 1
    SC      T2, (T0)
    BEQ     T2, 0, L1
    NOP
```

This atomically increments the word addressed by T0. Changing the ADD to an OR changes this to an atomic bit set.

LL Load Linked (continued) LL

The operation of LL is undefined if the addressed location is uncached and, for synchronization between multiple processors, the operation of LL is undefined if the addressed location is noncoherent. Exceptions also cause SC to fail, so persistent exceptions must be avoided.

This instruction is available in User mode, and it is not necessary for CP0 to be enabled.

If either of the two least-significant bits of the effective address is nonzero, an address error exception takes place.

Operation:

```
32   T:   $vAddr \leftarrow ((offset_{15})^{16} \,||\, offset_{15..0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
          $mem \leftarrow LoadMemory\ (uncached, WORD, pAddr, vAddr, DATA)$
          $GPR[rt] \leftarrow mem$
          $LLbit \leftarrow 1$
          SyncOperation()
64   T:   $vAddr \leftarrow ((offset_{15})^{48} \,||\, offset_{15..0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
          $pAddr \leftarrow PAddr_{PSIZE-1..3} \,||\, (pAddr_{2..0}\ \mathit{xor}\ (ReverseEndian \,||\, 0^2)$
          $mem \leftarrow LoadMemory\ (uncached, WORD, pAddr, vAddr, DATA)$
          $byte \leftarrow vAddr_{2..0}\ \mathit{xor}\ (BigEndianCPU \,||\, 0^2)$
          $GPR[rt] \leftarrow (mem_{31+8*byte})^{32} \,||\, mem_{31+8*byte..8*byte}$
          $LLbit \leftarrow 1$
          SyncOperation()
```

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

LLD Load Linked Doubleword LLD

31 26	25 21	20 16	15 0
LLD 110100	base	rt	offset
6	5	5	16

Format:

LLD rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the doubleword at the memory location specified by the effective address are loaded into general register *rt*.

This instruction implicitly performs a SYNC operation; all loads and stores to shared memory fetched prior to the LLD must access memory before the LLD, and loads and stores to shared memory fetched subsequent to the LLD must access memory after the LLD.

The processor begins checking the accessed doubleword for modification by other processors and devices.

Load Linked Doubleword and Store Conditional Doubleword can be used to atomically update memory locations:

```
L1:
    LLD     T1, (T0)
    ADD     T2, T1, 1
    SCD     T2, (T0)
    BEQ     T2, 0, L1
    NOP
```

This atomically increments the word addressed by T0. Changing the ADD to an OR changes this to an atomic bit set.

LLD Load Linked Doubleword (continued) LLD

The operation of LLD is undefined if the addressed location is uncached and, for synchronization between multiple processors, the operation of LLD is undefined if the addressed location is noncoherent. Exceptions also cause SCD to fail, so persistent exceptions must be avoided.

This instruction is available in User mode, and it is not necessary for CP0 to be enabled.

If any of the three least-significant bits of the effective address are nonzero, an address error exception takes place.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
64   T:   vAddr ← $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15..0})$ + GPR[base]
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
          GPR[rt]  ← mem
          LLbit  ← 1
          SyncOperation()
```

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception
Reserved instruction exception (R4000 in 32-bit mode)

LUI Load Upper Immediate LUI

31 26	25 21	20 16	15 0
LUI 001111	0 00000	rt	immediate
6	5	5	16

Format:

LUI rt, immediate

Description:

The 16-bit *immediate* is shifted left 16 bits and concatenated to 16 bits of zeros. The result is placed into general register *rt*. In 64-bit mode, the loaded word is sign-extended.

Operation:

```
32   T:   $GPR[rt] \leftarrow immediate \; || \; 0^{16}$

64   T:   $GPR[rt] \leftarrow (immediate_{15})^{32} \; || \; immediate \; || \; 0^{16}$
```

Exceptions:

None

LW Load Word LW

31 26	25 21	20 16	15 0
LW 1 0 0 0 1 1	base	rt	offset
6	5	5	16

Format:

LW rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the word at the memory location specified by the effective address are loaded into general register *rt*. In 64-bit mode, the loaded word is sign-extended.

If either of the two least-significant bits of the effective address is non-zero, an address error exception occurs.

Operation:

```
32   T:   vAddr ← $((\text{offset}_{15})^{16} \,||\, \text{offset}_{15..0})$ + GPR[base]
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
          GPR[rt] ← mem
64   T:   vAddr ← $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15..0})$ + GPR[base]
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          pAddr ← $\text{pAddr}_{\text{PSIZE-1..3}} \,||\, (\text{pAddr}_{2..0}$ xor (ReverseEndian || $0^2$))
          mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
          byte ← $\text{vAddr}_{2..0}$ xor (BigEndianCPU || $0^2$)
          GPR[rt] ← $(\text{mem}_{31+8*\text{byte}})^{32} \,||\, \text{mem}_{31+8*\text{byte}..8*\text{byte}}$
```

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

LWCz Load Word To Coprocessor LWCz

31 26	25 21	20 16	15 0
LWCz 1 1 0 0 x x*	base	rt	offset
6	5	5	16

Format:

LWCz rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The processor reads a word from the addressed memory location, and makes the data available to coprocessor unit *z*. The manner in which each coprocessor uses the data is defined by the individual coprocessor specifications.

If either of the two least-significant bits of the effective address is non-zero, an address error exception occurs.

This instruction is not valid for use with CP0.

*See the table "Opcode Bit Encoding" on next page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

LWCz Load Word To Coprocessor (continued) LWCz

Operation:

```
32  T:  $vAddr \leftarrow ((offset_{15})^{16} \,||\, offset_{15..0}) + GPR[base]$
        $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
        $byte \leftarrow vAddr_{1..0}$
        $mem \leftarrow LoadMemory\ (uncached, DOUBLEWORD, pAddr, vAddr, DATA)$
        $COPzLW\ (rt, mem)$
64  T:  $vAddr \leftarrow ((offset_{15})^{48} \,||\, offset_{15..0}) + GPR[base\}$
        $(pAddr, uncached) \leftarrow AddressTranslation(vAddr, DATA)$
        $pAddr \leftarrow pAddr_{PSIZE-1..3} \,||\, (pAddr_{2..0}\ xor\ (ReverseEndian \,||\, 0^2)$
        $mem \leftarrow LoadMemory\ (uncached, WORD, pAddr, vAddr, DATA)$
        $byte \leftarrow vAddr_{2..0}\ xor\ (BigEndianCPU \,||\, 0^2)$
        $COPzLW\ (byte, rt, mem)$
```

Exceptions:

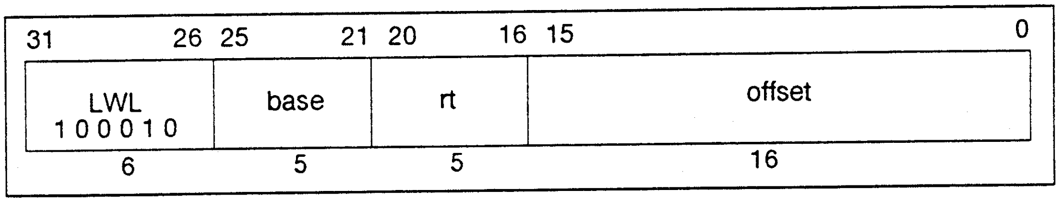
TLB refill exception
TLB invalid exception
Bus error exception
Address error exception
Coprocessor unusable exception

Opcode Bit Encoding:

LWCz	Bit # 31	30	29	28	27	26	0
LWC1	1	1	0	0	0	1	
	Bit # 31	30	29	28	27	26	0
LWC2	1	1	0	0	1	0	
	Bit # 31	30	29	28	27	26	0
LWC3	1	1	0	0	1	1	

Opcode Coprocessor Unit Number

LWL Load Word Left LWL



Format:

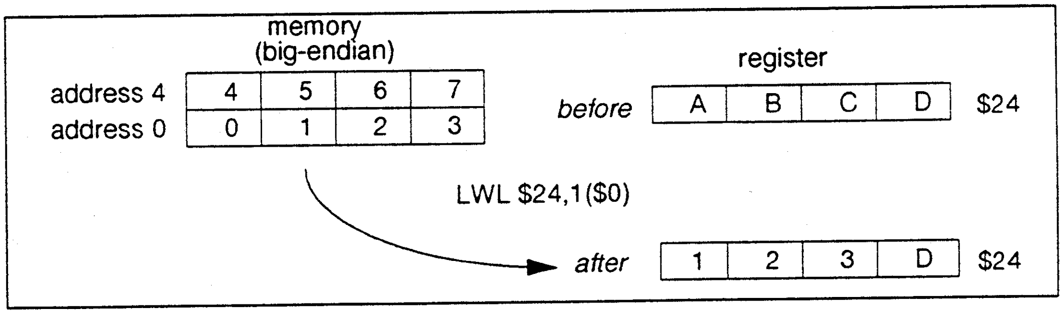
LWL rt, offset(base)

Description:

This instruction can be used in combination with the LWR instruction to load a register with four consecutive bytes from memory, when the bytes cross a boundary between two words. LWL loads the left portion of the register from the appropriate part of the high-order word; LWR loads the right portion of the register from the appropriate part of the low-order word.

The LWL instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which can specify an arbitrary byte. It reads bytes only from the word in memory which contains the specified starting byte. From one to four bytes will be loaded, depending on the starting byte specified. In 64-bit mode, the loaded word is sign-extended.

Conceptually, it starts at the specified byte in memory and loads that byte into the high-order (left-most) byte of the register; then it proceeds toward the low-order byte of the word in memory and the low-order byte of the register, loading bytes from memory into the register until it reaches the low-order byte of the word in memory. The least-significant (right-most) byte(s) of the register will not be changed.



LWL Load Word Left (continued) LWL

The contents of general register *rt* are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register *rt* and a following LWL (or LWR) instruction which also specifies register *rt*.

No address exceptions due to alignment are possible.

Operation:

```
32    T:    $vAddr \leftarrow ((offset_{15})^{16} \,||\, offset_{15..0}) + GPR[base]$
            $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
            $pAddr \leftarrow pAddr_{PSIZE-1..2} \,||\, (pAddr_{1..0}\ xor\ ReverseEndian^{2})$
            if $BigEndianMem = 0$ then
                  $pAddr \leftarrow pAddr_{PSIZE-31..2} \,||\, 0^{2}$
            endif
            $byte \leftarrow vAddr_{1..0}\ xor\ BigEndianCPU^{2}$
            $mem \leftarrow LoadMemory\ (uncached, byte, pAddr, vAddr, DATA)$
            $GPR[rt] \leftarrow mem_{7+8*byte..0} \,||\, GPR[rt]_{23-8*byte..0}$
64    T:    $vAddr \leftarrow ((offset_{15})^{48} \,||\, offset_{15..0}) + GPR[base]$
            $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
            $pAddr \leftarrow pAddr_{PSIZE-1..3} \,||\, (pAddr_{2..0}\ xor\ ReverseEndian^{3})$
            if $BigEndianMem = 0$ then
                  $pAddr \leftarrow pAddr_{PSIZE-1..3} \,||\, 0^{3}$
            endif
            $byte \leftarrow vAddr_{1..0}\ xor\ BigEndianCPU^{2}$
            $word \leftarrow vAddr_{2}\ xor\ BigEndianCPU$
            $mem \leftarrow LoadMemory\ (uncached, 0 \,||\, byte, pAddr, vAddr, DATA)$
            $temp \leftarrow mem_{31+32*word-8*byte..32*word} \,||\, GPR[rt]_{23-8*byte..0}$
            $GPR[rt] \leftarrow (temp_{31})^{32} \,||\, temp$
```

LWL Load Word Left (continued) LWL

Given a doubleword in a register and a doubleword in memory, the operation of LWL is as follows:

LWL

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

vAddr$_{2..0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	destination	type	offset		destination	type	offset	
			LEM	BEM			LEM	BEM
0	S S S S P F G H	0	0	7	S S S S I J K L	3	4	0
1	S S S S O P G H	1	0	6	S S S S J K L H	2	4	1
2	S S S S N O P H	2	0	5	S S S S K L G H	1	4	2
3	S S S S M N O P	3	0	4	S S S S L F G H	0	4	3
4	S S S S L F G H	0	4	3	S S S S M N O P	3	0	4
5	S S S S K L G H	1	4	2	S S S S N O P H	2	0	5
6	S S S S J K L H	2	4	1	S S S S O P G H	1	0	6
7	S S S S I J K L	3	4	0	S S S S P F G H	0	0	7

LEM	BigEndianMem = 0
BEM	BigEndianMem = 1
Type	AccessType (see Figure 2-2) sent to memory
Offset	pAddr$_{2..0}$ sent to memory
S	sign-extend of destination$_{31}$

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

LWR Load Word Right LWR

31 26	25 21	20 16	15 0
LWR 1 0 0 1 1 0	base	rt	offset
6	5	5	16

Format:

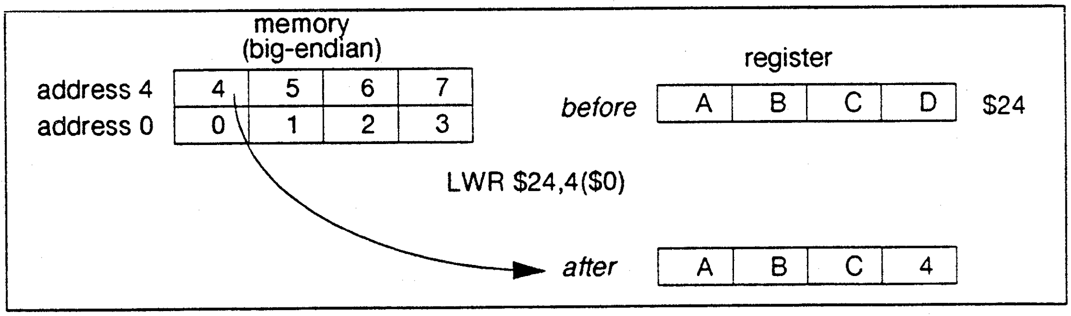
LWR rt, offset(base)

Description:

This instruction can be used in combination with the LWL instruction to load a register with four consecutive bytes from memory, when the bytes cross a boundary between two words. LWR loads the right portion of the register from the appropriate part of the low-order word; LWL loads the left portion of the register from the appropriate part of the high-order word.

The LWR instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which can specify an arbitrary byte. It reads bytes only from the word in memory which contains the specified starting byte. From one to four bytes will be loaded, depending on the starting byte specified. In 64-bit mode, the loaded word is sign-extended.

Conceptually, it starts at the specified byte in memory and loads that byte into the low-order (right-most) byte of the register; then it proceeds toward the high-order byte of the word in memory and the high-order byte of the register, loading bytes from memory into the register until it reaches the high-order byte of the word in memory. The most significant (left-most) byte(s) of the register will not be changed.



The contents of general register *rt* are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register *rt* and a following LWR (or LWL) instruction which also specifies register *rt*.

No address exceptions due to alignment are possible.

Operation:

```
32    T:  $vAddr \leftarrow ((offset_{15})^{16} \,\|\, offset_{15..0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1..2} \,\|\, (pAddr_{1..0}\ \mathit{xor}\ ReverseEndian^{2})$
          if $BigEndianMem = 0$ then
                $pAddr \leftarrow pAddr_{PSIZE-31..2} \,\|\, 0^{2}$
          endif
          $byte \leftarrow vAddr_{1..0}\ \mathit{xor}\ BigEndianCPU^{2}$
          $mem \leftarrow LoadMemory\ (uncached, byte, pAddr, vAddr, DATA)$
          $GPR[rt] \leftarrow mem_{31..32\text{-}8*byte..0} \,\|\, GPR[rt]_{31-8*byte..0}$
64    T:  $vAddr \leftarrow ((offset_{15})^{48} \,\|\, offset_{15..0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1..3} \,\|\, (pAddr_{2..0}\ \mathit{xor}\ ReverseEndian^{3})$
          if $BigEndianMem = 1$ then
                $pAddr \leftarrow pAddr_{PSIZE-31..3} \,\|\, 0^{3}$
          endif
          $byte \leftarrow vAddr_{1..0}\ \mathit{xor}\ BigEndianCPU^{2}$
          $word \leftarrow vAddr_{2}$ xor BigEndianCPU
          $mem \leftarrow LoadMemory\ (uncached, 0 \,\|\, byte, pAddr, vAddr, DATA)$
          $temp \leftarrow GPR[rt]_{31..32\text{-}8*byte..0} \,\|\, mem_{31+32*word\text{-}32*word+8*byte}$
          $GPR[rt] \leftarrow (temp_{31})^{32} \,\|\, temp$
```

LWR Load Word Right (continued) LWR

Given a word in a register and a word in memory, the operation of LWR is as follows:

LWR

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

vAddr$_{2..0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	destination	type	offset		destination	type	offset	
			LEM	BEM			LEM	BEM
0	S S S S M N O P	0	0	4	S S S S E F G I	0	7	0
1	S S S S E M N O	1	1	4	S S S S E F I J	1	6	0
2	S S S S E F M N	2	2	4	S S S S E I J K	2	5	0
3	S S S S E F G M	3	3	4	S S S S I J K L	3	4	0
4	S S S S I J K L	0	4	0	S S S S E F G M	0	3	4
5	S S S S E I J K	1	5	0	S S S S E F M N	1	2	4
6	S S S S E F I J	2	6	0	S S S S E M N O	2	1	4
7	S S S S E F G I	3	7	0	S S S S M N O P	3	0	4

LEM	BigEndianMem = 0
BEM	BigEndianMem = 1
Type	AccessType (see Figure 2-2) sent to memory
Offset	pAddr2..0 sent to memory
S	sign-extend of destination$_{31}$

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

LWU Load Word Unsigned LWU

31 26	25 21	20 16	15 0
LWU 1 0 1 1 1 1	base	rt	offset
6	5	5	16

Format:

LWU rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the word at the memory location specified by the effective address are loaded into general register *rt*. The loaded word is zero-extended.

If either of the two least-significant bits of the effective address is non-zero, an address error exception occurs.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
64    T:    $\mathrm{vAddr} \leftarrow ((\mathrm{offset}_{15})^{48} \,||\, \mathrm{offset}_{15..0}) + \mathrm{GPR[base]}$
            $(\mathrm{pAddr, uncached}) \leftarrow \mathrm{AddressTranslation\ (vAddr, DATA)}$
            $\mathrm{pAddr} \leftarrow \mathrm{pAddr}_{\mathrm{PSIZE-1..3}} \,||\, (\mathrm{pAddr}_{2..0}\ \mathit{xor}\ (\mathrm{ReverseEndian} \,||\, 0^2)$
            $\mathrm{mem} \leftarrow \mathrm{LoadMemory\ (uncached, WORD, pAddr, vAddr, DATA)}$
            $\mathrm{byte} \leftarrow \mathrm{vAddr}_{2..0}\ \mathit{xor}\ (\mathrm{BigEndianCPU} \,||\, 0^2)$
            $\mathrm{GPR[rt]} \leftarrow 0^{32} \,||\, \mathrm{mem}_{31+8*\mathrm{byte}..8*\mathrm{byte}}$
```

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception
Reserved instruction exception (R4000 in 32-bit mode)

MFC0 Move From System Control Coprocessor MFC0

31 26	25 21	20 16	15 11	10 0
COP0 0 1 0 0 0 0	MF 0 0 0 0 0	rt	rd	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format:

MFC0 rt, rd

Description:

The contents of coprocessor register *rd* of the CP0 are loaded into general register *rt*.

Operation:

```
32    T:     data ← CPR[0,rd]
      T+1:   GPR[rt] ← data

64    T:     data ← CPR[0,rd]
      T+1:   GPR[rt] ← $(data_{31})^{32}$ || $data_{31..0}$
```

Exceptions:

Coprocessor unusable exception

MFCz Move From Coprocessor MFCz

31 26	25 21	20 16	15 11	10 0
COPz 0 1 0 0 x x*	MF 0 0 0 0 0	rt	rd	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format:

MFCz rt, rd

Description:

The contents of coprocessor register *rd* of coprocessor z are loaded into general register *rt*.

Execution of the instruction referencing coprocessor 3 causes a reserved instruction exception, not a coprocessor unusable exception.

Operation:

```
32   T:     data ← CPR[z,rd]
     T+1:   GPR[rt] ← data
64   T:     if $rd_0 = 0$
                      data ← $\mathrm{CPR}[z,rd_{4..1} \,||\, 0]_{31..0}$
            else
                      data ← $\mathrm{CPR}[z,rd_{4..1} \,||\, 0]_{63..32}$
            endif
```

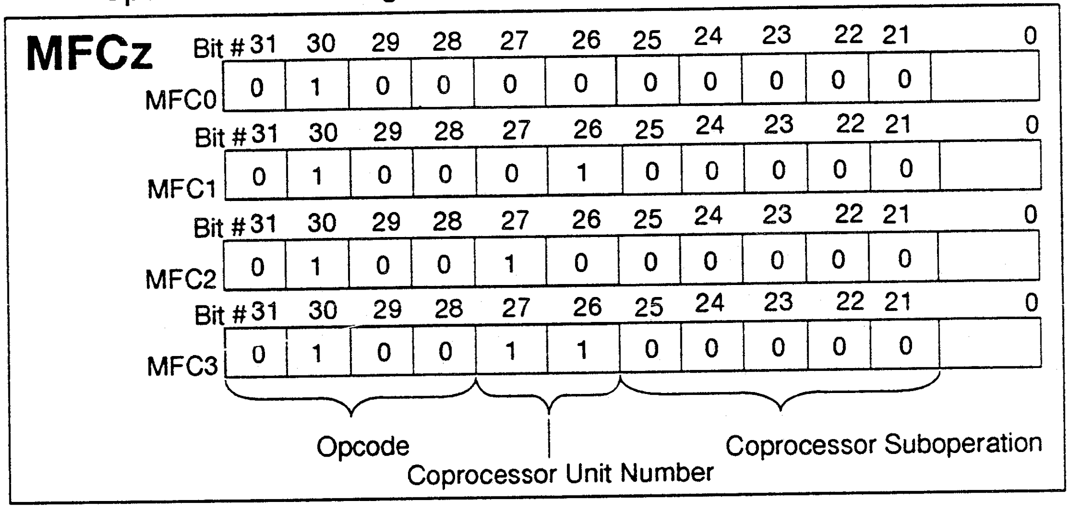
Exceptions:

Coprocessor unusable exception
Reserved instruction exception (coprocessor 3)

*See the table "Opcode Bit Encoding" on next page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

MFCz Move From Coprocessor (continued) MFCz

Opcode Bit Encoding:



MFCz

	Bit # 31	30	29	28	27	26	25	24	23	22	21	0
MFC0	0	1	0	0	0	0	0	0	0	0	0	
MFC1	0	1	0	0	0	1	0	0	0	0	0	
MFC2	0	1	0	0	1	0	0	0	0	0	0	
MFC3	0	1	0	0	1	1	0	0	0	0	0	

MFHI Move From HI MFHI

31 26	25 16	15 11	10 6	5 0
SPECIAL 000000	0 00 0000 0000	rd	0 00000	MFHI 010000
6	10	5	5	6

Format:

MFHI rd

Description:

The contents of special register *HI* are loaded into general register *rd*.

To ensure proper operation in the event of interruptions, the two instructions which follow a MFHI instruction may not be any of the instructions which modify the *HI* register: MULT, MULTU, DIV, DIVU, MTHI, DMULT, DMULTU, DDIV, DDIVU.

Operation:

```
32, 64     T:     GPR[rd] ← HI
```

Exceptions:

None

MFLO Move From Lo MFLO

31 26	25 16	15 11	10 6	5 0
SPECIAL 000000	0 00 0000 0000	rd	0 00000	MFLO 010010
6	10	5	5	6

Format:

MFLO rd

Description:

The contents of special register *LO* are loaded into general register *rd*.

To ensure proper operation in the event of interruptions, the two instructions which follow a MFLO instruction may not be any of the instructions which modify the *LO* register: MULT, MULTU, DIV, DIVU, MTLO, DMULT, DMULTU, DDIV, DDIVU.

Operation:

```
        32, 64      T:      GPR[rd] ← LO
```

Exceptions:

None

MTC0 Move To System Control Coprocessor MTC0

31 26	25 21	20 16	15 11	10 0
COP0 010000	MT 00100	rt	rd	0 000 0000 0000
6	5	5	5	11

Format:

MTC0 rt, rd

Description:

The contents of general register *rt* are loaded into coprocessor register *rd* of the CP0.

Because the state of the virtual address translation system may be altered by this instruction, the operation of load, store instructions and TLB operations immediately prior to and after this instruction are undefined.

Operation:

```
32, 64   T:      data ← GPR[rt]
         T+1:    CPR[0,rd] ← data
```

Exceptions:

Coprocessor unusable exception

MTCz Move To Coprocessor MTCz

31 26	25 21	20 16	15 11	10 0
COPz 0 1 0 0 x x*	MT 0 0 1 0 0	rt	rd	0 000 0000 0000
6	5	5	5	11

Format:

MTCz rt, rd

Description:

The contents of general register *rt* are loaded into coprocessor register *rd* of coprocessor z. Execution of the instruction referencing coprocessor 3 causes a reserved instruction exception, not a coprocessor unusable exception.

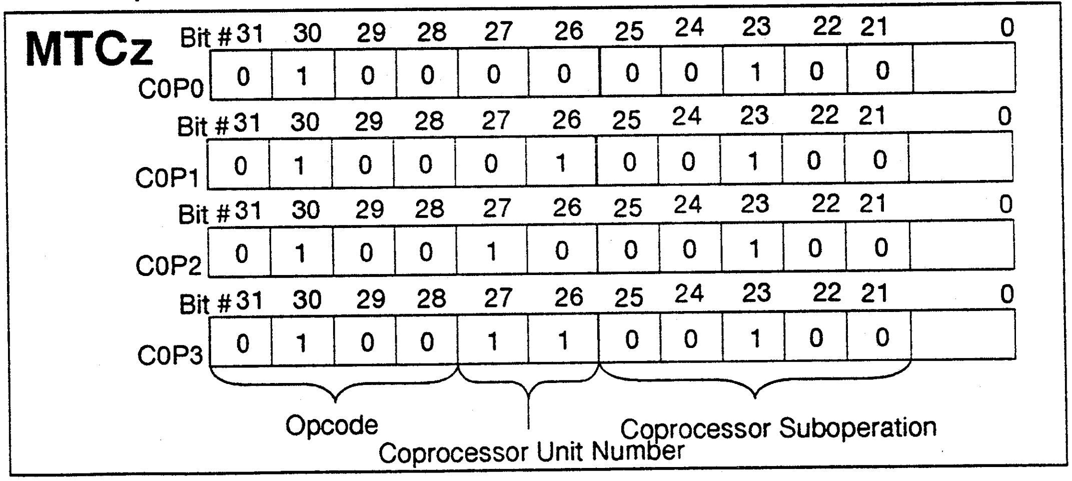
Operation:

```
32    T:      data ← GPR[rt]
      T+1:    CPR[z,rd] ← data
64    T:      data ← GPR[rt]31..0
      T+1:    if rd0 = 0
                 CPR[z,rd4..1 || 0] ← CPR[z, rd4..1 || 0]63..32 || data
              else
                     data ← CPR[z,rd4..1 || 0]63..32
              endif
```

Exceptions:

Coprocessor unusable exception
Reserved instruction exception (coprocessor 3)

*Opcode Bit Encoding:



	Bit # 31	30	29	28	27	26	25	24	23	22	21	0
COP0	0	1	0	0	0	0	0	0	1	0	0	
COP1	0	1	0	0	0	1	0	0	1	0	0	
COP2	0	1	0	0	1	0	0	0	1	0	0	
COP3	0	1	0	0	1	1	0	0	1	0	0	

MTHI Move To HI MTHI

31 26	25 21	20 6	5 0
SPECIAL 000000	rs	0 000 000000000000	MTHI 010001
6	5	15	6

Format:

MTHI rs

Description:

The contents of general register *rs* are loaded into special register *HI*.

If a MTHI operation is executed following a MULT, MULTU, DIV, or DIVU instruction, but before any MFLO, MFHI, MTLO, or MTHI instructions, the contents of special register *LO* are undefined.

Operation:

```
32,64  T–2:  HI ← undefined
       T–1:  HI ← undefined
       T:    HI ← GPR[rs]
```

Exceptions:

None

MTLO Move To LO MTLO

31 26	25 21	20 6	5 0
SPECIAL 000000	rs	0 000000000000000	MTLO 010011
6	5	15	6

Format:

MTLO rs

Description:

The contents of general register *rs* are loaded into special register *LO*

If a MTLO operation is executed following a MULT, MULTU, DIV, or DIVU instruction, but before any MFLO, MFHI, MTLO, or MTHI instructions, the contents of special register *HI* are undefined.

Operation:

```
32,64  T–2:  LO ← undefined
       T–1:  LO ← undefined
       T:    LO ← GPR[rs]
```

Exceptions:

None

MULT Multiply MULT

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 00 0000 0000	MULT 011000
6	5	5	10	6

Format:

MULT rs, rt

Description:

The contents of general registers *rs* and *rt* are multiplied, treating both operands as 32-bit 2's-complement values. No integer overflow exception occurs under any circumstances. In 64-bit mode, the operands must be valid 32-bit, sign-extended values.

When the operation completes, the low-order word of the double result is loaded into special register *LO*, and the high-order word of the double result is loaded into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results of these instructions are undefined. Correct operation requires separating reads of *HI* or *LO* from writes by a minimum of two other instructions.

MULT — Multiply (continued) — MULT

Operation:

```
32   T–2:   LO ← undefined
            HI ← undefined
     T–1:   LO ← undefined
            HI ← undefined
     T:     t ← GPR[rs] * GPR[rt]
            LO ← $t_{31..0}$
            H I← $t_{63..32}$

64   T–2:   LO ← undefined
            HI ← undefined
     T–1:   LO ← undefined
            HI ← undefined
     T:     t ← $GPR[rs]_{31..0}$ * $GPR[rt]_{31..0}$
            LO ← $(t_{31})^{32}$ || $t_{31..0}$
            HI ← $(t_{63})^{32}$ || $t_{63..32}$
```

Exceptions:

None

MULTU Multiply Unsigned MULTU

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 00 0000 0000	MULTU 011001
6	5	5	10	6

Format:

MULTU rs, rt

Description:

The contents of general register *rs* and the contents of general register *rt* are multiplied, treating both operands as unsigned values. No overflow exception occurs under any circumstances. In 64-bit mode, the operands must be valid 32-bit, sign-extended values.

When the operation completes, the low-order word of the double result is loaded into special register *LO*, and the high-order word of the double result is loaded into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results of these instructions are undefined. Correct operation requires separating reads of *HI* or *LO* from writes by a minimum of two instructions.

MULTU Multiply Unsigned (continued) MULTU

Operation:

```
32   T-2:   LO ← undefined
            HI ← undefined
     T-1:   LO ← undefined
            HI ← undefined
     T:     t ← (0 || GPR[rs]) * (0 || GPR[rt])
            LO ← $t_{31..0}$
            HI ← $t_{63..32}$

64   T-2:   LO ← undefined
            HI  ← undefined
     T-1:   LO ← undefined
            HI  ← undefined
     T:     t   ← (o || $GPR[rs]_{31..0}$) * (0 || $GPR[rt]_{31..0}$)
            LO  ← $(t_{31})^{32}$ || $t_{31..0}$
            HI  ← $(t_{63})^{32}$ || $t_{63..32}$
```

Exceptions:

None

NOR Nor NOR

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	NOR 100111
6	5	5	5	5	6

Format:

NOR rd, rs, rt

Description:

The contents of general register *rs* are combined with the contents of general register *rt* in a bit-wise logical NOR operation. The result is placed into general register *rd*.

Operation:

```
32, 64     T:    GPR[rd] ← GPR[rs] nor GPR[rt]
```

Exceptions:

None

OR Or OR

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	OR 100101
6	5	5	5	5	6

Format:

OR rd, rs, rt

Description:

The contents of general register *rs* are combined with the contents of general register *rt* in a bit-wise logical OR operation. The result is placed into general register *rd*.

Operation:

32, 64 T: GPR[rd] ← GPR[rs] *or* GPR[rt]

Exceptions:

None

ORI Or Immediate ORI

31 26	25 21	20 16	15 0
ORI 0 0 1 1 0 1	rs	rt	immediate
6	5	5	16

Format:

ORI rt, rs, immediate

Description:

The 16-bit *immediate* is zero-extended and combined with the contents of general register *rs* in a bit-wise logical OR operation. The result is placed into general register *rt*.

Operation:

```
32   T:   $GPR[rt] \leftarrow GPR[rs]_{31..16} \,||\, (immediate\ \mathit{or}\ GPR[rs]_{15..0})$
64   T:   $GPR[rt] \leftarrow GPR[rs]_{63..16} \,||\, (immediate\ \mathit{or}\ GPR[rs]_{15..0})$
```

Exceptions:

None

SB Store Byte SB

31 26	25 21	20 16	15 0
SB 1 0 1 0 0 0	base	rt	offset
6	5	5	16

Format:

SB rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The least-significant byte of register *rt* is stored at the effective address.

Operation:

```
32  T:  $vAddr \leftarrow ((offset_{15})^{16} \,||\, offset_{15..0}) + GPR[base]$
        $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
        $pAddr \leftarrow pAddr_{PSIZE-1..2} \,||\, (pAddr_{1..0}\ xor\ ReverseEndian^{2})$
        $byte \leftarrow vAddr_{1..0}\ xor\ BigEndianCPU^{2}$
        $data \leftarrow GPR[rt]_{31-8*byte..0} \,||\, 0^{8*byte}$
        StoreMemory (uncached, BYTE, data, pAddr, vAddr, DATA)
64  T:  $vAddr \leftarrow ((offset_{15})^{48} \,||\, offset_{15..0}) + GPR[base]$
        $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
        $pAddr \leftarrow pAddr_{PSIZE-1..3} \,||\, (pAddr_{2..0}\ xor\ ReverseEndian^{3})$
        $byte \leftarrow vAddr_{2..0}\ xor\ BigEndianCPU^{3}$
        $data \leftarrow GPR[rt]_{63-8*byte..0} \,||\, 0^{8*byte}$
        StoreMemory (uncached, BYTE, data, pAddr, vAddr, DATA)
```

Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception

SC Store Conditional SC

31 26	25 21	20 16	15 0
SC 111000	base	rt	offset
6	5	5	16

Format:

SC rt, offset(base)

Description:

The 16-bit offset is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of general register *rt* are conditionally stored at the memory location specified by the effective address.

This instruction implicitly performs a SYNC operation; loads and stores to shared memory fetched prior to the SC must access memory before the SC; loads and stores to shared memory fetched subsequent to the SC must access memory after the SC.

If any other processor or device has modified the physical address since the time of the previous Load Linked instruction, or if an ERET instruction occurs between the Load Linked instruction and this store instruction, the store fails and is inhibited from taking place.

The success or failure of the store operation (as defined above) is indicated by the contents of general register *rt* after execution of the instruction. A successful store sets the contents of general register *rt* to 1; an unsuccessful store sets it to 0.

The operation of Store Conditional is undefined when the address is different from the address used in the last Load Linked.

This instruction is available in User mode; it is not necessary for CP0 to be enabled.

If either of the two least-significant bits of the effective address is nonzero, an address error exception takes place.

SC Store Conditional (continued) SC

If this instruction should both fail and take an exception, the exception takes precedence.

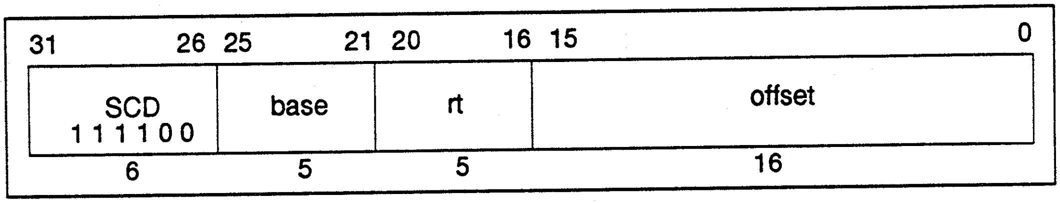
Operation:

```
32   T:   $vAddr \leftarrow ((offset_{15})^{16} \,||\, offset_{15..0}) + GPR[base]$
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          data ← GPR[rt]
          if LLbit then
               StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
          endif
          $GPR[rt] \leftarrow 0^{31} \,||\, LLbit$
          SyncOperation()

64   T:   $vAddr \leftarrow ((offset_{15})^{48} \,||\, offset_{15..0}) + GPR[base]$
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          $pAddr \leftarrow pAddr_{PSIZE-1..3} \,||\, ( pAddr_{2..0}\ xor\ (ReverseEndian \,||\, 0^{2}))$
          $data \leftarrow GPR[rt]_{63-8*byte..0} \,||\, 0^{8*byte}$
          if LLbit then
               StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
          endif
          $GPR[rt] \leftarrow 0^{63} \,||\, LLbit$
          SyncOperation()
```

Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception

SCD Store Conditional Doubleword SCD



Format:

SCD rt, offset(base)

Description:

The 16-bit offset is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of general register *rt* are conditionally stored at the memory location specified by the effective address.

This instruction implicitly performs a SYNC operation; loads and stores to shared memory fetched prior to the SCD must access memory before the SCD; loads and stores to shared memory fetched subsequent to the SCD must access memory after the SCD.

If any other processor or device has modified the physical address since the time of the previous Load Linked Doubleword instruction, or if an ERET instruction occurs between the Load Linked Doubleword instruction and this store instruction, the store fails and is inhibited from taking place.

The success or failure of the store operation (as defined above) is indicated by the contents of general register *rt* after execution of the instruction. A successful store sets the contents of general register *rt* to 1; an unsuccessful store sets it to 0.

The operation of Store Conditional Doubleword is undefined when the address is different from the address used in the last Load Linked Doubleword.

This instruction is available in User mode; it is not necessary for CP0 to be enabled.

If either of the three least-significant bits of the effective address is non-zero, an address error exception takes place.

SCD Store Conditional Doubleword (continued) SCD

If this instruction should both fail and take an exception, the exception takes precedence.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
64    T:    vAddr ← $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15..0})$ + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            data ← GPR[rt]
            if LLbit then
                StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
            endif
            GPR[rt] ← $0^{63}$ || LLbit
            SyncOperation()
```

Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception
Reserved instruction exception (R4000 in 32-bit mode)

SD　Store Doubleword　SD

31 … 26	25 … 21	20 … 16	15 … 0
SD 111111	base	rt	offset
6	5	5	16

Format:

SD rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of general register *rt* are stored at the memory location specified by the effective address.

If either of the three least-significant bits of the effective address are non-zero, an address error exception occurs.

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
64    T:    vAddr ← $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15..0}) + \text{GPR[base]}$
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            data ← GPR[rt]
            StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
```

Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception
Reserved instruction exception (R4000 in 32-bit user mode
R4000 in 32-bit supervisor mode)

SDCz Store Doubleword From Coprocessor SDCz

31 26	25 21	20 16	15 0
SDCz 1 1 1 1 x x*	base	rt	offset
6	5	5	16

Format:

SDCz rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. Coprocessor unit z sources a doubleword, which the processor writes to the addressed memory location. The data to be stored is defined by individual coprocessor specifications.

If any of the three least-significant bits of the effective address are non-zero, an address error exception takes place.

This instruction is not valid for use with CP0.

This instruction is undefined when the least-significant bit of the *rt-field* is non-zero.

*See the table, "Opcode Bit Encoding" on next page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

SDCz Store Doubleword From Coprocessor (continued) SDCz

Operation:

```
32   T:   $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $data \leftarrow COPzSD(rt),$
          StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)

64   T:   $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $data \leftarrow COPzSD(rt),$
          StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
```

Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception
Coprocessor unusable exception

Opcode Bit Encoding:

SDCz

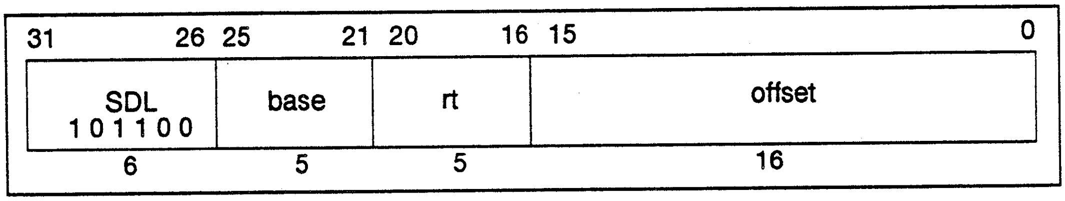
Bit #	31	30	29	28	27	26	0
SDC1	1	1	1	1	0	1	

Bit #	31	30	29	28	27	26	0
SDC2	1	1	1	1	1	0	

Bit #	31	30	29	28	27	26	0
SDC3	1	1	1	1	1	1	

SD opcode (bits 31–28) Coprocessor Unit Number (bits 27–26)

SDL Store Doubleword Left SDL



Format:

SDL rt, offset(base)

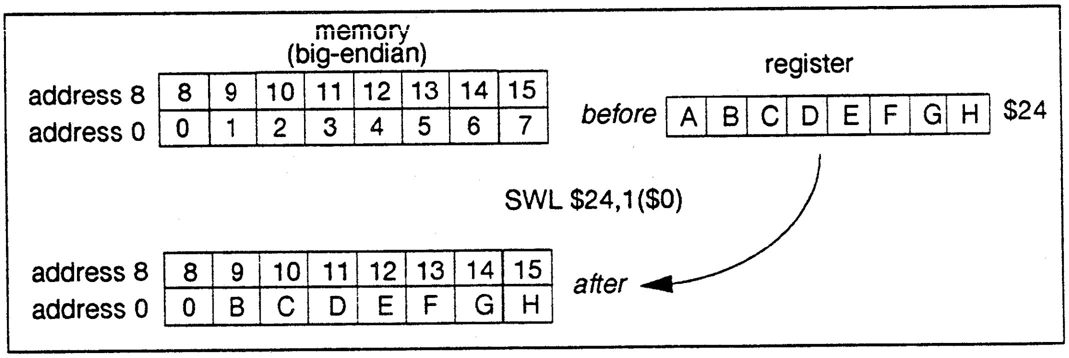
Description:

This instruction can be used with the SDR instruction to store the contents of a register into eight consecutive bytes of memory, when the bytes cross a boundary between two doublewords. SDL stores the left portion of the register into the appropriate part of the high-order doubleword of memory; SDR stores the right portion of the register into the appropriate part of the low-order doubleword.

The SDL instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which may specify an arbitrary byte. It alters only the word in memory which contains that byte. From one to four bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the most-significant byte of the register and copies it to the specified byte in memory; then it proceeds toward the low-order byte of the register and the low-order byte of the word in memory, copying bytes from register to memory until it reaches the low-order byte of the word in memory.

No address exceptions due to alignment are possible.



SDL Store Doubleword Left (continued) SDL

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
64    T:   $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
           $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
           $pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0}\ xor\ ReverseEndian^{3})$
           If $BigEndianMem = 0$ then
                 $pAddr \leftarrow pAddr_{31..3} \| 0^{3}$
           endif
           $byte \leftarrow vAddr_{2..0}\ xor\ BigEndianCPU^{3}$
           $data \leftarrow 0^{56-8*byte} \| GPR[rt]_{63..56-8*byte}$
           Storememory (uncached, byte, data, pAddr, vAddr, DATA)
```

SDL Store Doubleword Left (continued) SDL

Given a doubleword in a register and a doubleword in memory, the operation of SWL is as follows:

LWL

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

$vAddr_{2..0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	destination	type	offset		destination	type	offset	
			LEM	BEM			LEM	BEM
0	I J K L M N O A	0	0	7	A B C D E F G H	7	0	0
1	I J K L M N A B	1	0	6	I A B C D E F G	6	0	1
2	I J K L M A B C	2	0	5	I J A B C D E F	5	0	2
3	I J K L A B C D	3	0	4	I J K A B C D E	4	0	3
4	I J K A B C D E	4	0	3	I J K L A B C D	3	0	4
5	I J A B C D E F	5	0	2	I J K L M A B C	2	0	5
6	I A B C D E F G	6	0	1	I J K L M N A B	1	0	6
7	A B C D E F G H	7	0	0	I J K L M N O A	0	0	7

LEM	BigEndianMem = 0
BEM	BigEndianMem = 1
Type	AccessType (see Figure 2-2) sent to memory
Offset	$pAddr_{2..0}$ sent to memory

Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception
Reserved instruction exception (R4000 in 32-bit mode)

SDR Store Doubleword Right SDR

31 26	25 21	20 16	15 0
SDR 101101	base	rt	offset
6	5	5	16

Format:

SDR rt, offset(base)

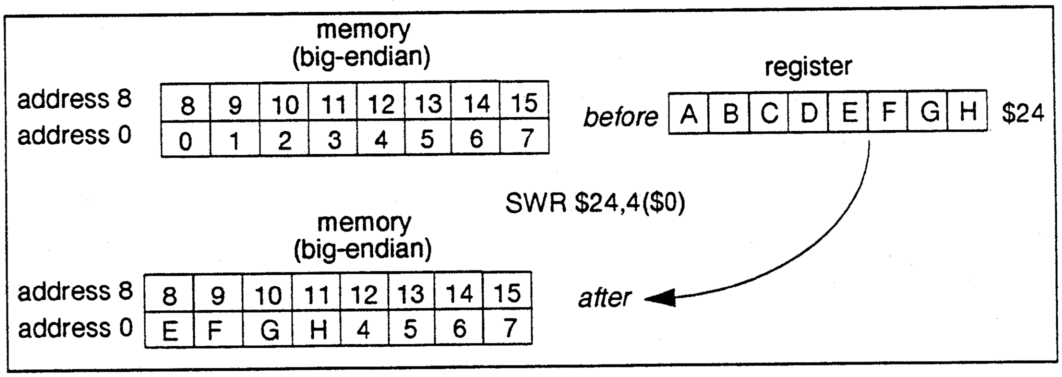
Description:

This instruction can be used with the SDL instruction to store the contents of a register into eight consecutive bytes of memory, when the bytes cross a boundary between two doublewords. SDR stores the right portion of the register into the appropriate part of the low-order doubleword; SDL stores the left portion of the register into the appropriate part of the low-order doubleword of memory.

The SDR instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which may specify an arbitrary byte. It alters only the word in memory which contains that byte. From one to eight bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the least-significant (rightmost) byte of the register and copies it to the specified byte in memory; then it proceeds toward the high-order byte of the register and the high-order byte of the word in memory, copying bytes from register to memory until it reaches the high-order byte of the word in memory.

No address exceptions due to alignment are possible.



SDR Store Doubleword Right (continued) SDR

This operation is only defined for the R4000 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
64    T:   vAddr ← $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15..0}) + \text{GPR[base]}$
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← $\text{pAddr}_{\text{PSIZE}-1..3} \,||\, (\text{pAddr}_{2..0}$ xor $\text{ReverseEndian}^3)$
           If BigEndianMem = 0 then
                 pAddr ← $\text{pAddr}_{\text{PSIZE}-31..3} \,||\, 0^3$
           endif
           byte ← $\text{vAddr}_{1..0}$ xor $\text{BigEndianCPU}^3$
           data ← $\text{GPR[rt]}_{63-8*\text{byte}} \,||\, 0^{8*\text{byte}}$
           StoreMemory (uncached, DOUBLEWORD-byte, data, pAddr, vAddr,
```

SDR Store Doubleword Right (continued) SDR

Given a doubleword in a register and a doubleword in memory, the operation of SDR is as follows:

SDR

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

vAddr$_{2..0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	destination	type	offset		destination	type	offset	
			LEM	BEM			LEM	BEM
0	A B C D E F G H	7	0	0	H J K L M N O P	0	7	0
1	B C D E F G H P	6	1	0	G H K L M N O P	1	6	0
2	C D E F G H O P	5	2	0	F G H L M N O P	2	5	0
3	D E F G H N O P	4	3	0	E F G H M N O P	3	4	0
4	E F G H M N O P	3	4	0	D E F G H N O P	4	3	0
5	F G H L M N O P	2	5	0	C D E F G H O P	5	2	0
6	G H K L M N O P	1	6	0	B C D E F G H P	6	1	0
7	H J K L M N O P	0	7	0	A B C D E F G H	7	0	0

LEM	BigEndianMem = 0
BEM	BigEndianMem = 1
Type	AccessType (see Figure 2-2) sent to memory
Offset	pAddr$_{2..0}$ sent to memory

Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception
Reserved instruction exception (R4000 in 32-bit mode)

SH Store Halfword SH

31 26	25 21	20 16	15 0
SH 101001	base	rt	offset
6	5	5	16

Format:

SH rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form an unsigned effective address. The least-significant halfword of register *rt* is stored at the effective address. If the least-significant bit of the effective address is non-zero, an address error exception occurs.

Operation:

```
32   T:   $vAddr \leftarrow ((offset_{15})^{16} \,||\, offset_{15..0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1..2} \,||\, (pAddr_{1..0}\ xor\ (ReverseEndian \,||\, 0))$
          $byte \leftarrow vAddr_{1..0}\ \ xor\ (BigEndianCPU \,||\, 0)$
          $data \leftarrow GPR[rt]_{31-8*byte..0} \,||\, 0^{8*byte}$
          StoreMemory (uncached, HALFWORD, data, pAddr, vAddr, DATA)

64   T:   $vAddr \leftarrow ((offset_{15})^{48} \,||\, offset_{15..0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1..3} \,||\, (pAddr_{2..0}\ xor\ (ReverseEndian^2 \,||\, 0))$
          $byte \leftarrow vAddr_{2..0}\ \ xor\ (BigEndianCPU^2 \,||\, 0)$
          $data \leftarrow GPR[rt]_{63-8*byte..0} \,||\, 0^{8*byte}$
          StoreMemory (uncached, HALFWORD, data, pAddr, vAddr, DATA)
```

Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception

SLL Shift Left Logical SLL

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	sa	SLL 000000
6	5	5	5	5	6

Format:

SLL rd, rt, sa

Description:

The contents of general register *rt* are shifted left by *sa* bits, inserting zeros into the low-order bits. The result is placed in register *rd*. In 64-bit mode, the operand must be a valid sign-extended, 32-bit value.

Operation:

```
32   T:   $GPR[rd] \leftarrow GPR[rt]_{31-sa..0} \,||\, 0^{sa}$

64   T:   $s \leftarrow 0 \,||\, sa$
          $temp \leftarrow GPR[rt]_{31-s..0} \,||\, 0^{s}$
          $GPR[rd] \leftarrow (temp_{31})^{32} \,||\, temp$
```

Exceptions:

None

SLLV Shift Left Logical Variable SLLV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	0 00000	SLLV 000100
6	5	5	5	5	6

Format:

SLLV rd, rt, rs

Description:

The contents of general register *rt* are shifted left by the number of bits specified by the low-order five bits contained as contents of general register *rs*, inserting zeros into the low-order bits. The result is placed in register *rd*. In 64-bit mode, the operand must be a valid sign-extended, 32-bit value.

Operation:

```
32   T:   $s \leftarrow GP[rs]_{4..0}$
          $GPR[rd] \leftarrow GPR[rt]_{(31-s)..0} \| 0^s$

64   T:   $s \leftarrow 0 \| GP[rs]_{4..0}$
          $temp \leftarrow GPR[rt]_{(31-s)..0} \| 0^s$
          $GPR[rd] \leftarrow (temp_{31})^{32} \| temp$
```

Exceptions:

None

SLT Set On Less Than SLT

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SLT 101010
6	5	5	5	5	6

Format:

SLT rd, rs, rt

Description:

The contents of general register *rt* are subtracted from the contents of general register *rs*. Considering both quantities as signed integers, if the contents of general register *rs* are less than the contents of general register *rt*, the result is set to one, otherwise the result is set to zero. The result is placed into general register *rd*.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflows.

Operation:

```
32    T:    if GPR[rs] < GPR[rt] then
                      GPR[rd] ← $0^{31}$ || 1
            else
                      GPR[rd] ← $0^{32}$
            endif

64    T:    if GPR[rs] < GPR[rt] then
                      GPR[rd] ← $0^{63}$ || 1
            else
                      GPR[rd] ← $0^{64}$
            endif
```

Exceptions:

None

SLTI Set On Less Than Immediate SLTI

31 26	25 21	20 16	15 0
SLTI 0 0 1 0 1 0	rs	rt	immediate
6	5	5	16

Format:

SLTI rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and subtracted from the contents of general register *rs*. Considering both quantities as signed integers, if *rs* is less than the sign-extended immediate, the result is set to one, otherwise the result is set to zero. The result is placed into general register *rt*.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflows.

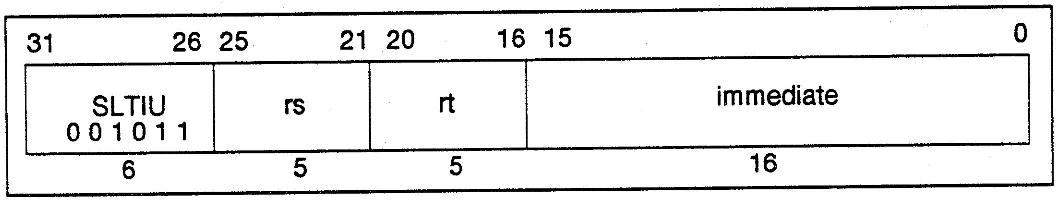
Operation:

```
32   T:   if GPR[rs] < $(\text{immediate}_{15})^{16} \| \text{immediate}_{15..0}$ then
                    GPR[rd] ← $0^{31} \| 1$
          else
                    GPR[rd] ← $0^{32}$
          endif

64   T:   if GPR[rs] < $(\text{immediate}_{15})^{48} \| \text{immediate}_{15..0}$ then
                    GPR[rd] ← $0^{63} \| 1$
          else
                    GPR[rd] ← $0^{64}$
          endif
```

Exceptions:

None

SLTIU Set On Less Than Immediate Unsigned SLTIU



Format:

SLTIU rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and subtracted from the contents of general register *rs*. Considering both quantities as unsigned integers, if rs is less than the sign-extended immediate, the result is set to one, otherwise the result is set to zero. The result is placed into general register rt.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflows.

Operation:

```
32   T:   if (0 || GPR[rs]) < $(\text{immediate}_{15})^{16}$ || $\text{immediate}_{15..0}$ then
                    GPR[rd] ← $0^{31}$ || 1
          else
                    GPR[rd] ← $0^{32}$
          endif

64   T:   if (0 || GPR[rs]) < $(\text{immediate}_{15})^{48}$ || $\text{immediate}_{15..0}$ then
                    GPR[rd] ← $0^{63}$ || 1
          else
                    GPR[rd] ← $0^{64}$
          endif
```

Exceptions:

None

SLTU Set On Less Than Unsigned SLTU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SLTU 101011
6	5	5	5	5	6

Format:

SLTU rd, rs, rt

Description:

The contents of general register *rt* are subtracted from the contents of general register *rs*. Considering both quantities as unsigned integers, if the contents of general register *rs* are less than the contents of general register *rt*, the result is set to one, otherwise the result is set to zero. The result is placed into general register *rd*.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflows.

Operation:

```
32    T:    if (0 || GPR[rs]) < 0 || GPR[rt] then
                 GPR[rd] ← $0^{31}$ || 1
            else
                 GPR[rd] ← $0^{32}$
            endif
64    T:    if (0 || GPR[rs]) < 0 || GPR[rt] then
                 GPR[rd] ← $0^{63}$ || 1
            else
                 GPR[rd] ← $0^{64}$
            endif
```

Exceptions:

None

SRA Shift Right Arithmetic SRA

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	SRA 000011
6	5	5	5	5	6

Format:

SRA rd, rt, sa

Description:

The contents of general register *rt* are shifted right by *sa* bits, sign-extending the high-order bits. The result is placed in register *rd*. In 64-bit mode, the operand must be a valid sign-extended, 32-bit value.

Operation:

```
32   T:   $GPR[rd] \leftarrow (GPR[rt]_{31})^{sa} \,||\, GPR[rt]_{31..sa}$

64   T:   $s \leftarrow 0 \,||\, sa$
          $temp \leftarrow (GPR[rt]_{31})^{s} \,||\, GPR[rt]_{31..s}$
          $GPR[rd] \leftarrow (temp_{31})^{32} \,||\, temp$
```

Exceptions:

None

SRAV Shift Right Arithmetic Variable SRAV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SRAV 000111
6	5	5	5	5	6

Format:

SRAV rd, rt, rs

Description:

The contents of general register *rt* are shifted right by the number of bits specified by the low-order five bits of general register *rs*, sign-extending the high-order bits. The result is placed in register *rd*. In 64-bit mode, the operand must be a valid sign-extended, 32-bit value.

Operation:

```
32   T:   $s \leftarrow GPR[rs]_{4..0}$
          $GPR[rd] \leftarrow (GPR[rt]_{31})^{s} \,||\, GPR[rt]_{31..s}$
64   T:   $s \leftarrow GPR[rs]_{4..0}$
          $temp \leftarrow (GPR[rt]_{31})^{s} \,||\, GPR[rt]_{31..s}$
          $GPR[rd] \leftarrow (temp_{31})^{32} \,||\, temp$
```

Exceptions:

None

SRL Shift Right Logical SRL

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	SRL 000010
6	5	5	5	5	6

Format:

SRL rd, rt, sa

Description:

The contents of general register *rt* are shifted right by *sa* bits, inserting zeros into the high-order bits. The result is placed in register *rd*. In 64-bit mode, the operand must be a valid sign-extended, 32-bit value.

Operation:

```
32    T:    $GPR[rd] \leftarrow 0^{sa} || GPR[rt]_{31..sa}$

64    T:    $s \leftarrow 0 || sa$
            $temp \leftarrow 0^{s} || GPR[rt]_{31..s}$
            $GPR[rd] \leftarrow (temp_{31})^{32} || temp$
```

Exceptions:

None

SRLV Shift Right Logical Variable SRLV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SRLV 000110
6	5	5	5	5	6

Format:

SRLV rd, rt, rs

Description:

The contents of general register *rt* are shifted right by the number of bits specified by the low-order five bits of general register *rs,* inserting zeros into the high-order bits. The result is placed in register *rd*. In 64-bit mode, the operand must be a valid sign-extended, 32-bit value.

Operation:

```
32   T:   $s \leftarrow GPR[rs]_{4..0}$
          $GPR[rd] \leftarrow 0^s \,||\, GPR[rt]_{31..s}$

64   T:   $s \leftarrow GPR[rs]_{4..0}$
          $temp \leftarrow 0^s \,||\, GPR[rt]_{31..s}$
          $GPR[rd] \leftarrow (temp_{31})^{32} \,||\, temp$
```

Exceptions:

None

SUB Subtract SUB

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SUB 100010
6	5	5	5	5	6

Format:

SUB rd, rs, rt

Description:

The contents of general register *rt* are subtracted from the contents of general register *rs* to form a result. The result is placed into general register *rd*. In 64-bit mode, the operands must be valid sign-extended, 32-bit values.

The only difference between this instruction and the SUBU instruction is that SUBU never traps on overflow.

An integer overflow exception takes place if the carries out of bits 30 and 31 differ (2's-complement overflow). The destination register *rd* is not modified when an integer overflow exception occurs.

Operation:

```
32    T:    $GPR[rd] \leftarrow GPR[rs] - GPR[rt]$

64    T:    $temp \leftarrow GPR[rs] - GPR[rt]$
            $GPR[rd] \leftarrow (temp_{31})^{32} \,||\, temp_{31..0}$
```

Exceptions:

Integer overflow exception

SUBU Subtract Unsigned SUBU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SUBU 100011
6	5	5	5	5	6

Format:

SUBU rd, rs, rt

Description:

The contents of general register *rt* are subtracted from the contents of general register *rs* to form a result. The result is placed into general register *rd*. In 64-bit mode, the operands must be valid sign-extended, 32-bit values.

The only difference between this instruction and the SUB instruction is that SUBU never traps on overflow. No integer overflow exception occurs under any circumstances.

Operation:

```
32    T:    $GPR[rd] \leftarrow GPR[rs] - GPR[rt]$

64    T:    $temp \leftarrow GPR[rs] - GPR[rt]$
            $GPR[rd] \leftarrow (temp_{31})^{32} \,||\, temp_{31..0}$
```

Exceptions:

None

SW Store Word SW

31 26	25 21	20 16	15 0
SW 1 0 1 0 1 1	base	rt	offset
6	5	5	16

Format:

SW rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of general register *rt* are stored at the memory location specified by the effective address.

If either of the two least-significant bits of the effective address are non-zero, an address error exception occurs.

Operation:

```
32   T:   $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $data \leftarrow GPR[rt]$
          StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
64   T:   $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0}\ xor\ (ReverseEndian \| 0^2)$
          $byte \leftarrow vAddr_{2..0}\ xor\ (BigEndianCPU \| 0^2)$
          $data \leftarrow GPR[rt]_{63-8*byte} \| 0^{8*byte}$
          StoreMemory (uncache, WORD, data, pAddr, vAddr DATA)
```

Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception

SWCz Store Word From Coprocessor SWCz

31 26	25 21	20 16	15 0
SWCz 1 1 1 0 x x*	base	rt	offset
6	5	5	16

Format:

SWCz rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. Coprocessor unit z sources a word, which the processor writes to the addressed memory location. The data to be stored is defined by individual coprocessor specifications. This instruction is not valid for use with CP0. If either of the two least-significant bits of the effective address is non-zero, an address error exception occurs.

Execution of the instruction referencing coprocessor 3 causes a reserved instruction exception, not a coprocessor unusable exception.

Operation:

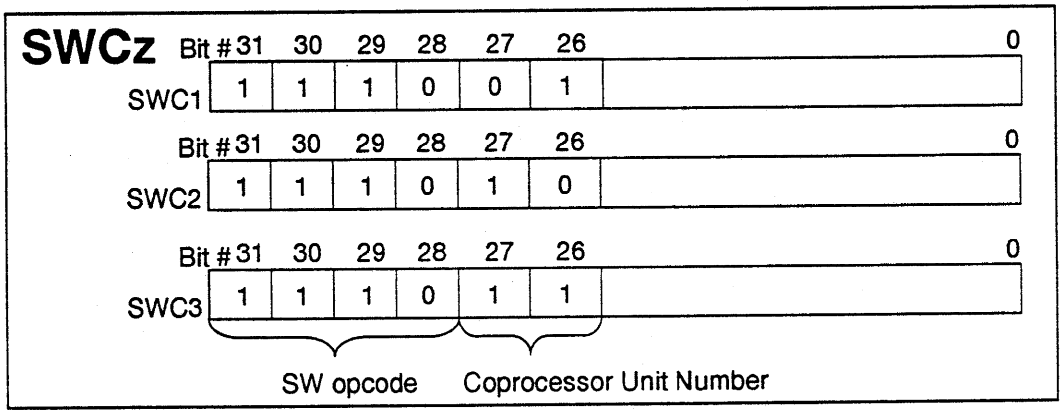
```
32  T:  vAddr ← $((\text{offset}_{15})^{16} \,||\, \text{offset}_{15..0})$ + GPR[base]
        (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
        byte ← $\text{vAddr}_{1..0}$
        data ← COPzSW (byte, rt)
        StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)

64  T:  vAddr ← $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15..0})$ + GPR[base]
        (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
        pAddr ← $\text{pAddr}_{\text{PSIZE}-1..3} \,||\, (\text{pAddr}_{2..0}$ *xor* (ReverseEndian $|| \; 0^2$))
        byte ← $\text{vAddr}_{2..0}$ *xor* (BigEndianCPU $|| \; 0^2$)
        data ← COPzSW (byte,rt)
        StoreMemory (uncache, WORD, data, pAddr, vAddr DATA)
```

*See the table "Opcode Bit Encoding" on next page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

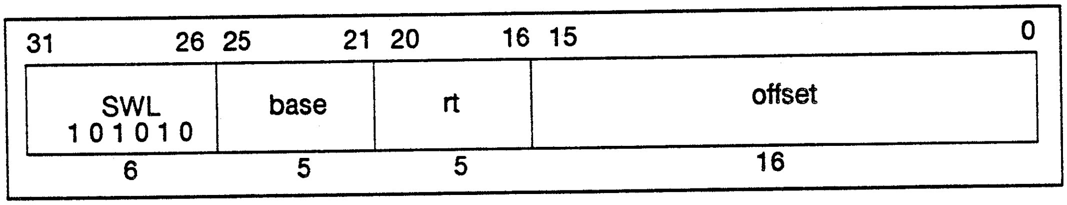
Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception
Coprocessor unusable exception
Reserved instruction exception (coprocessor 3)

Opcode Bit Encoding:



SWL Store Word Left SWL



Format:

SWL rt, offset(base)

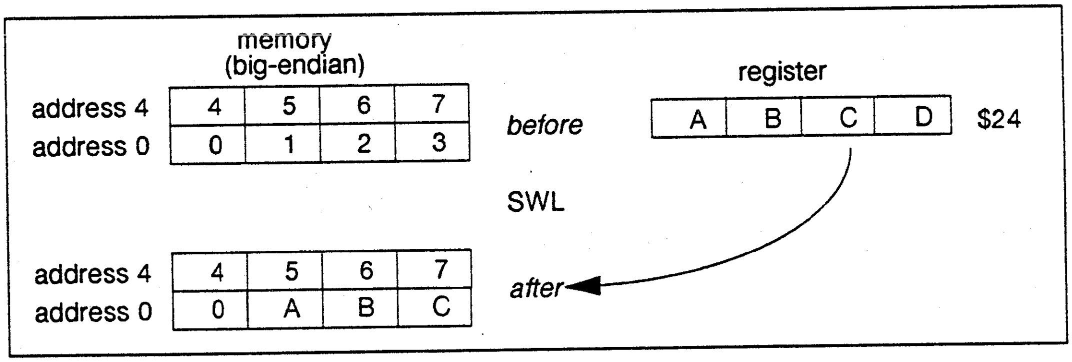
Description:

This instruction can be used with the SWR instruction to store the contents of a register into four consecutive bytes of memory, when the bytes cross a boundary between two words. SWL stores the left portion of the register into the appropriate part of the high-order word of memory; SWR stores the right portion of the register into the appropriate part of the low-order word.

The SWL instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which may specify an arbitrary byte. It alters only the word in memory which contains that byte. From one to four bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the most-significant byte of the register and copies it to the specified byte in memory; then it proceeds toward the low-order byte of the register and the low-order byte of the word in memory, copying bytes from register to memory until it reaches the low-order byte of the word in memory.

No address exceptions due to alignment are possible.



SWL Store Word Left (Continued) SWL

Operation:

```
32   T:   $vAddr \leftarrow ((offset_{15})^{16} \,||\, offset_{15..0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1..2} \,||\, (pAddr_{1..0}\ xor\ ReverseEndian^2)$
          If BigEndianMem = 0 then
               $pAddr \leftarrow pAddr_{PSIZE-1..2} \,||\, 0^2$
          endif
          $byte \leftarrow vAddr_{1..0}\ xor\ BigEndianCPU^2$
          $data \leftarrow 0^{24-8*byte} \,||\, GPR[rt]_{31..24-8*byte}$
          Storememory (uncached, byte, data, pAddr, vAddr, DATA)
64   T:   $vAddr \leftarrow ((offset_{15})^{48} \,||\, offset_{15..0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1..3} \,||\, (pAddr_{2..0}\ xor\ ReverseEndian^2)$
          If BigEndianMem = 0 then
               $pAddr \leftarrow pAddr_{31..2} \,||\, 0^2$
          endif
          $byte \leftarrow vAddr_{1..0}\ xor\ BigEndianCPU^2$
          if $(vAddr_2\ xor\ BigEndianCPU) = 0$ then
               $data \leftarrow 0^{32} \,||\, 0^{24-8*byte} \,||\, GPR[rt]_{31..24-8*byte}$
          else
               $data \leftarrow 0^{24-8*byte} \,||\, GPR[rt]_{31..24-8*byte} \,||\, 0^{32}$
          endif
          StoreMemory(uncached, byte, data, pAddr, vAddr, DATA)
```

SWL Store Word Left (Continued) SWL

Given a doubleword in a register and a doubleword in memory, the operation of SWL is as follows:

SWL

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

vAddr$_{2..0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	destination	type	offset		destination	type	offset	
			LEM	BEM			LEM	BEM
0	I J K L M N O E	0	0	7	E F G H M N O P	3	4	0
1	I J K L M N E F	1	0	6	I E F G M N O P	2	4	1
2	I J K L M E F G	2	0	5	I J E F M N O P	1	4	2
3	I J K L E F G H	3	0	4	I J K E M N O P	0	4	3
4	I J K E M N O P	0	4	3	I J K L E F G H	3	0	4
5	I J E F M N O P	1	4	2	I J K L M E F G	2	0	5
6	I E F G M N O P	2	4	1	I J K L M N E F	1	0	6
7	E F G H M N O P	3	4	0	I J K L M N O E	0	0	7

LEM	BigEndianMem = 0
BEM	BigEndianMem = 1
Type	AccessType (see Figure 2-2) sent to memory
Offset	pAddr$_{2..0}$ sent to memory

Exceptions:

TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception

SWR Store Word Right SWR

31 26	25 21	20 16	15 0
SWR 1 0 1 1 1 0	base	rt	offset
6	5	5	16

Format:

SWR rt, offset(base)

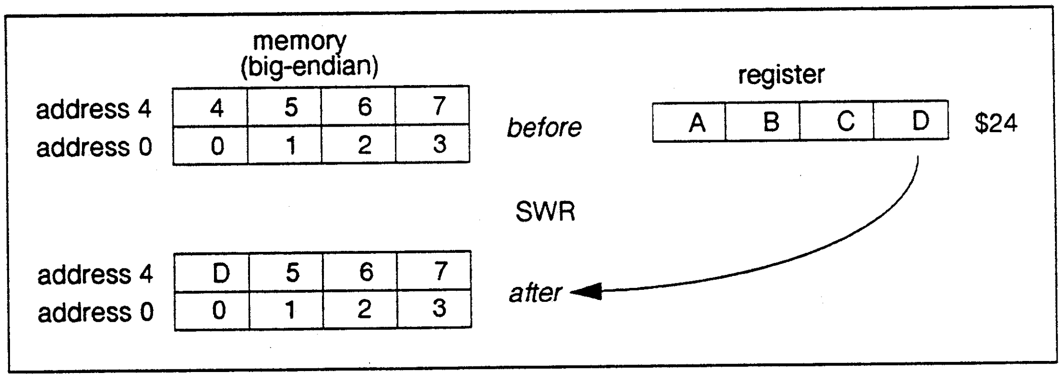
Description:

This instruction can be used with the SWL instruction to store the contents of a register into four consecutive bytes of memory, when the bytes cross a boundary between two words. SWR stores the right portion of the register into the appropriate part of the low-order word; SWL stores the left portion of the register into the appropriate part of the low-order word of memory.

The SWR instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which may specify an arbitrary byte. It alters only the word in memory which contains that byte. From one to four bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the least-significant (rightmost) byte of the register and copies it to the specified byte in memory; then it proceeds toward the high-order byte of the register and the high-order byte of the word in memory, copying bytes from register to memory until it reaches the high-order byte of the word in memory.

No address exceptions due to alignment are possible.



SWR   Store Word Right (Continued)   SWR

Operation:

```
32    T:   $vAddr \leftarrow ((offset_{15})^{16} \,||\, offset_{15..0}) + GPR[base]$
           $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
           $pAddr \leftarrow pAddr_{PSIZE-1..2} \,||\, (pAddr_{1..0}\ xor\ ReverseEndian^2)$
           If BigEndianMem = 0 then
                 $pAddr \leftarrow pAddr_{PSIZE-31..2} \,||\, 0^2$
           endif
           $byte \leftarrow vAddr_{1..0}\ xor\ BigEndianCPU^2$
           $data \leftarrow GPR[rt]_{31-8*byte} \,||\, 0^{8*byte}$
           Storememory (uncached, WORD-byte, data, pAddr, vAddr, DATA)
64    T:   $vAddr \leftarrow ((offset_{15})^{48} \,||\, offset_{15..0}) + GPR[base]$
           $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
           $pAddr \leftarrow pAddr_{PSIZE-1..3} \,||\, (pAddr_{2..0}\ xor\ ReverseEndian^3)$
           If BigEndianMem = 0 then
                 $pAddr \leftarrow pAddr_{31..2} \,||\, 0^2$
           endif
           $byte \leftarrow vAddr_{1..0}\ xor\ BigEndianCPU^2$
           if $(vAddr_2\ xor\ BigEndianCPU) = 0$ then
                 $data \leftarrow 0^{32} \,||\, GPR[rt]_{31-8*byte..0} \,||\, 0^{8*byte}$
           else
                 $data \leftarrow GPR[rt]_{31-8*byte..0} \,||\, 0^{8*byte} \,||\, 0^{32}$
           endif
           StoreMemory(uncached, WORD-byte, data ,pAddr, vAddr, DATA)
```

Given a doubleword in a register and a doubleword in memory, the operation of SWR is as follows:

SWR

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

	BigEndianCPU = 0				BigEndianCPU = 1			
$vAddr_{2..0}$	destination	type	offset		destination	type	offset	
			LEM	BEM			LEM	BEM
0	I J K L E F G H	3	0	4	H J K L M N O P	0	7	0
1	I J K L F G H P	2	1	4	G H K L M N O P	1	6	0
2	I J K L G H O P	1	2	4	F G H L M N O P	2	5	0
3	I J K L H N O P	0	3	4	E F G H M N O P	3	4	0
4	E F G H M N O P	3	4	0	I J K L H N O P	0	3	4
5	F G H L M N O P	2	5	0	I J K L G H O P	1	2	4
6	G H K L M N O P	1	6	0	I J K L F G H P	2	1	4
7	H J K L M N O P	0	7	0	I J K L E F G H	3	0	4

LEM	BigEndianMem = 0
BEM	BigEndianMem = 1
Type	AccessType (see Figure 2-2) sent to memory
Offset	$pAddr_{2..0}$ sent to memory

Exceptions:

TLB refill exception

TLB invalid exception

TLB modification exception

Bus error exception

Address error exception

SYNC Synchronize SYNC

31 … 26	25 … 6	5 … 0
SPECIAL 000000	0 0000 0000 0000 0000 0000	SYNC 001111
6	20	6

Format:

SYNC

Description:

The SYNC instruction ensures that any loads and stores fetched *prior to* the present instruction are completed before any loads or stores *after* this instruction are allowed to start. Use of the SYNC instruction to serialize certain memory references may be required in multiprocessor environment for proper synchronization.

For example:

Processor A	Processor B
`SW R1, DATA`	`1: LW R2, FLAG`
`LI R2, 1`	`BEQ R2, R0, 1B`
`SYNC`	`NOP`
`SW R2, FLAG`	`SYNC`
	`LW R1, DATA`

The SYNC in processor A prevents DATA being written after FLAG, which could cause processor B to read stale data. The SYNC in processor B prevents DATA from being read before FLAG, which could likewise result in reading stale data. For processors which only execute loads and stores in order, with respect to shared memory, this instruction is a NOP.

LL and SC instructions implicitly perform a SYNC.

This instruction is allowed in User mode.

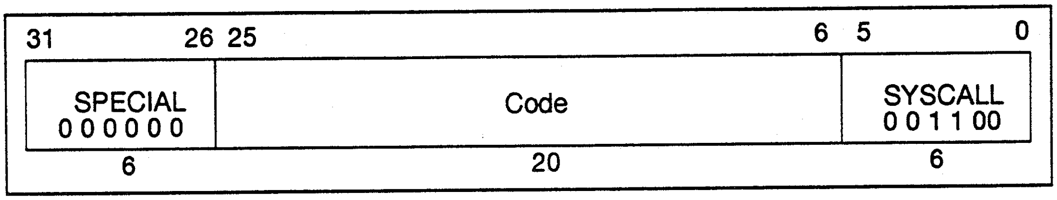
Operation:

```
32, 64    T:    SyncOperation()
```

Exceptions:

None

SYSCALL System Call SYSCALL



Format:

SYSCALL

Description:

A system call exception occurs, immediately and unconditionally transferring control to the exception handler.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
32, 64   T:    SystemCallException
```

Exceptions:

System Call exception

TEQ Trap If Equal TEQ

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	code	TEQ 1 1 0 1 0 0
6	5	5	10	6

Format:

TEQ rs, rt

Description:

The contents of general register *rt* are compared to general register *rs*. If the contents of general register *rs* are equal to the contents of general register *rt*, a trap exception occurs.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
32, 64    T:    if GPR[rs] = GPR[rt] then
                        TrapException
                endif
```

Exceptions:

Trap exception

TEQI — Trap If Equal Immediate — TEQI

31 … 26	25 … 21	20 … 16	15 … 0
REGIMM 000001	rs	TEQI 01100	immediate
6	5	5	16

Format:

TEQI rs, immediate

Description:

The 16-bit *immediate* is sign-extended and compared to the contents of general register *rs*. If the contents of general register *rs* are equal to the sign-extended *immediate*, a trap exception occurs.

Operation:

```
32   T:   if GPR[rs] = $(\text{immediate}_{15})^{16} \,||\, \text{immediate}_{15..0}$ then
               TrapException
          endif
64   T:   if GPR[rs] = $(\text{immediate}_{15})^{48} \,||\, \text{immediate}_{15..0}$ then
               TrapException
          endif
```

Exceptions:

Trap exception

TGE Trap If Greater Than Or Equal TGE

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	code	TGE 110000
6	5	5	10	6

Format:

TGE rs, rt

Description:

The contents of general register *rt* are compared to the contents of general register *rs*. Considering both quantities as signed integers, if the contents of general register *rs* are greater than or equal to the contents of general register *rt*, a trap exception occurs.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
32, 64   T:   if GPR[rs] ≥ GPR[rt] then
                   TrapException
              endif
```

Exceptions:

Trap exception

TGEI Trap If Greater Than Or Equal Immediate TGEI

31 26	25 21	20 16	15 0
REGIMM 000001	rs	TGEI 01000	immediate
6	5	5	16

Format:

TGEI rs, immediate

Description:

The 16-bit *immediate* is sign-extended and compared to the contents of general register *rs*. Considering both quantities as signed integers, if the contents of general register *rs* are greater than or equal to the sign-extended *immediate*, a trap exception occurs.

Operation:

```
32    T:    if GPR[rs] ≥ $(\text{immediate}_{15})^{16} \| \text{immediate}_{15..0}$ then
                  TrapException
            endif
64    T:    if GPR[rs] ≥ $(\text{immediate}_{15})^{48} \| \text{immediate}_{15..0}$ then
                  TrapException
            endif
```

Exceptions:

Trap exception

TGEIU — Trap If Greater Than Or Equal Immediate Unsigned — TGEIU

31 26	25 21	20 16	15 0
REGIMM 000001	rs	TGEIU 01001	immediate
6	5	5	16

Format:

TGEIU rs, immediate

Description:

The 16-bit *immediate* is sign-extended and compared to the contents of general register *rs*. Considering both quantities as unsigned integers, if the contents of general register *rs* are greater than or equal to the sign-extended *immediate,* a trap exception occurs.

Operation:

```
32    T:    if $(0 || GPR[rs]) \geq (0 || (immediate_{15})^{16} || immediate_{15..0})$ then
                  TrapException
            endif
64    T:    if $(0 || GPR[rs]) \geq (0 || (immediate_{15})^{48} || immediate_{15..0})$ then
                  TrapException
            endif
```

Exceptions:

Trap exception

TGEU Trap If Greater Than Or Equal Unsigned TGEU

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	code	TGEU 110001
6	5	5	10	6

Format:

TGEU rs, rt

Description:

The contents of general register *rt* are compared to the contents of general register *rs*. Considering both quantities as unsigned integers, if the contents of general register *rs* are greater than or equal to the contents of general register *rt*, a trap exception occurs.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
T:    if (0 || GPR[rs]) ≥ (0 || GPR[rt]) then
          TrapException
      endif
```

Exceptions:

Trap exception

TLBP Probe TLB For Matching Entry TLBP

31 26	25	24 6	5 0
COP0 0 1 0 0 0 0	CO 1	0 000 0000 0000 0000 0000	TLBP 0 0 1 0 0 0
6	1	19	6

Format:

TLBP

Description:

The *Index* register is loaded with the address of the TLB entry whose contents match the contents of the *EntryHi* register. If no TLB entry matches, the high-order bit of the *Index* register is set.

The architecture does not specify the operation of memory references associated with the instruction immediately after a TLBP instruction, nor is the operation specified if more than one TLB entry matches.

Operation:

```
32    T:    Index ← $1 || 0^{25} || 1^6$
            for i in 0..TLBEntries−1
                if (TLB[i]$_{95..77}$ = EntryHi$_{31..12}$) and (TLB[i]$_{76}$ or
                (TLB[i]$_{71..64}$ = EntryHi$_{7..0}$ )) then
                    Index ← $0^{26}$ || i$_{5..0}$
                endif
            endfor
64    T:    Index ← $1 || 0^{31}$
            for i in 0..TLBEntries−1
                if (TLB[i]$_{167..141}$ and not ($0^{15}$ || TLB[i]$_{216..205}$))
                 = EntryHi$_{39..13}$) and not ($0^{15}$ || TLB[i]$_{216..205}$)) and
                (TLB[i]$_{140}$ or (TLB[i]$_{135..128}$ = EntryHi$_{7..0}$ )) then
                    Index ← $0^{26}$ || i$_{5..0}$
                endif
            endfor
```

Exceptions:

Coprocessor unusable exception

TLBR Read Indexed TLB Entry TLBR

31 26	25	24 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBR 000001
6	1	19	6

Format:

TLBR

Description:

The *G* bit (controls ASID matching) read from the TLB is written into both *EntryLo0* and *EntryLo1*.

The *EntryHi* and *EntryLo* registers are loaded with the contents of the TLB entry pointed at by the contents of the TLB *Index* register. The operation is invalid (and the results are unspecified) if the contents of the TLB *Index* register are greater than the number of TLB entries in the processor.

Operation:

```
32    T:    PageMask ← TLB[Index$_{5..0}$]$_{127..96}$
            EntryHi ← TLB[Index$_{5..0}$]$_{95..64}$ and not TLB[Index$_{5..0}$]$_{127..96}$
            EntryLo1 ←TLB[Index$_{5..0}$]$_{63..32}$
            EntryLo0 ← TLB[Index$_{5..0}$]$_{31..0}$
64    T:    PageMask ← TLB[Index$_{5..0}$]$_{255..192}$
            EntryHi ← TLB[Index$_{5..0}$]$_{191..1284}$ and not TLB[Index$_{5..0}$]$_{255..192}$
            EntryLo1 ←TLB[Index$_{5..0}$]$_{127..65}$ || TLB[Index$_{5..0}$]$_{140}$
            EntryLo0 ← TLB[Index$_{5..0}$]$_{63..1}$ || TLB[Index$_{5..0}$]$_{140}$
```

Exceptions:

Coprocessor unusable exception

TLBWI Write Indexed TLB Entry TLBWI

31 26	25	24 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBWI 000010
6	1	19	6

Format:

TLBWI

Description:

The *G* bit of the TLB is written with the logical AND of the *G* bits in *EntryLo0* and *EntryLo1*.

The TLB entry pointed at by the contents of the TLB *Index* register is loaded with the contents of the *EntryHi* and *EntryLo* registers.

The operation is invalid (and the results are unspecified) if the contents of the TLB *Index* register are greater than the number of TLB entries in the processor.

Operation:

```
32, 64      T:    TLB[Index$_{5..0}$] ←
                     PageMask || (EntryHi and not PageMask) || EntryLo1 || EntryLo0
```

Exceptions:

Coprocessor unusable exception

TLBWR Write Random TLB Entry TLBWR

31 … 26	25	24 … 6	5 … 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	TLBWR 000110
6	1	19	6

Format:

TLBWR

Description:

The *G* bit of the TLB is written with the logical AND of the *G* bits in *EntryLo0* and *EntryLo1*.

The TLB entry pointed at by the contents of the TLB *Random* register is loaded with the contents of the *EntryHi* and *EntryLo* registers.

Operation:

```
32, 64      T:    TLB[$Random_{5..0}$] ←
                     PageMask || (EntryHi  and not PageMask) || EntryLo1 || EntryLo0
```

Exceptions:

Coprocessor unusable exception

TLT Trap If Less Than TLT

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	code	TLT 110010
6	5	5	10	6

Format:

TLT rs, rt

Description:

The contents of general register *rt* are compared to general register *rs*. Considering both quantities as signed integers, if the contents of general register *rs* are less than the contents of general register *rt*, a trap exception occurs.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
32, 64     T:    if GPR[rs] < GPR[rt] then
                       TrapException
                 endif
```

Exceptions:

Trap exception

TLTI Trap If Less Than Immediate TLTI

31 26	25 21	20 16	15 0
REGIMM 000001	rs	TLTI 01010	immediate
6	5	5	16

Format:

TLTI rs, immediate

Description:

The 16-bit *immediate* is sign-extended and compared to the contents of general register *rs*. Considering both quantities as signed integers, if the contents of general register *rs* are less than the sign-extended *immediate*, a trap exception occurs.

Operation:

```
32   T:   if GPR[rs] < $(\text{immediate}_{15})^{16} \| \text{immediate}_{15..0}$ then
              TrapException
          endif
64   T:   if GPR[rs] < $(\text{immediate}_{15})^{48} \| \text{immediate}_{15..0}$ then
              TrapException
          endif
```

Exceptions:

Trap exception

TLTIU Trap If Less Than Immediate Unsigned TLTIU

31 26	25 21	20 16	15 0
REGIMM 000001	rs	TLTIU 01011	immediate
6	5	5	16

Format:

TLTIU rs, immediate

Description:

The 16-bit *immediate* is sign-extended and compared to the contents of general register *rs*. Considering both quantities as signed integers, if the contents of general register *rs* are less than the sign-extended *immediate*, a trap exception occurs.

Operation:

```
32    T:    if (0 || GPR[rs]) <  (0 || (immediate$_{15}$)$^{16}$ || immediate$_{15..0}$) then
                  TrapException
            endif
32    T:    if (0 || GPR[rs]) <  (0 || (immediate$_{15}$)$^{48}$ || immediate$_{15..0}$) then
                  TrapException
            endif
```

Exceptions:

Trap exception

TLTU Trap If Less Than Unsigned TLTU

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	code	TLTU 110011
6	5	5	10	6

Format:

TLTU rs, rt

Description:

The contents of general register *rt* are compared to general register *rs*. Considering both quantities as unsigned integers, if the contents of general register *rs* are less than the contents of general register *rt*, a trap exception occurs.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
32, 64    T:    if (0 || GPR[rs]) < (0 || GPR[rt]) then
                     TrapException
                endif
```

Exceptions:

Trap exception

TNE Trap If Not Equal TNE

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	code	TNE 110110
6	5	5	10	6

Format:

TNE rs, rt

Description:

The contents of general register *rt* are compared to general register *rs*. If the contents of general register *rs* are not equal to the contents of general register *rt*, a trap exception occurs.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
32, 64     T:     if GPR[rs] ≠ GPR[rt] then
                        TrapException
                  endif
```

Exceptions:

Trap exception

TNEI Trap If Not Equal Immediate TNEI

31 26	25 21	20 16	15 0
REGIMM 000001	rs	TNEI 01110	immediate
6	5	5	16

Format:

TNEI rs, immediate

Description:

The 16-bit *immediate* is sign-extended and compared to the contents of general register *rs*. If the contents of general register *rs* are not equal to the sign-extended *immediate*, a trap exception occurs.

Operation:

```
32    T:    if $GPR[rs] \neq (immediate_{15})^{16} \| immediate_{15..0}$ then
                  TrapException
            endif
64    T:    if $GPR[rs] \neq (immediate_{15})^{48} \| immediate_{15..0}$ then
                  TrapException
            endif
```

Exceptions:

Trap exception

XOR Exclusive Or XOR

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	XOR 100110
6	5	5	5	5	6

Format:

XOR rd, rs, rt

Description:

The contents of general register *rs* are combined with the contents of general register *rt* in a bit-wise logical exclusive OR operation. The result is placed into general register *rd*.

Operation:

32, 64 T: GPR[rd] ← GPR[rs] *xor* GPR[rt]

Exceptions:

None

XORI Exclusive OR Immediate XORI

31 26	25 21	20 16	15 0
XORI 0 0 1 1 1 0	rs	rt	immediate
6	5	5	16

Format:

XORI rt, rs, immediate

Description:

The 16-bit *immediate* is zero-extended and combined with the contents of general register *rs* in a bit-wise logical exclusive OR operation. The result is placed into general register *rt*.

Operation:

```
32    T:    $GPR[rt] \leftarrow GPR[rs]\ \mathit{xor}\ (0^{16}\ ||\ immediate)$
64    T:    $GPR[rt] \leftarrow GPR[rs]\ \mathit{xor}\ (0^{48}\ ||\ immediate)$
```

Exceptions:

None

CPU Instruction Opcode Bit Encoding

The remainder of this Appendix presents the opcode bit encoding for the CPU instruction set (ISA and extensions), as implemented by the R4000. Figure A-2 lists the R4000 Opcode Bit Encoding.

Opcode

31..29	28..26 0	1	2	3	4	5	6	7
0	SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	COP0	COP1	COP2	*	BEQL	BNEL	BLEZL	BGTZL
3	DADDIε	DADDIUε	LDLε	LDRε	*	*	*	*
4	LB	LH	LWL	LW	LBU	LHU	LWR	LWUε
5	SB	SH	SWL	SW	SDLε	SDRε	SWR	CACHE δ
6	LL	LWC1	LWC2	*	LLDε	LDC1	LDC2	LDC3
7	SC	SWC1	SWC2	*	SCDε	SDC1	SDC2	SDC3

SPECIAL function

5..3	2..0 0	1	2	3	4	5	6	7
0	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
1	JR	JALR	*	*	SYSCALL	BREAK	*	SYNC
2	MFHI	MTHI	MFLO	MTLO	DSLLVε	*	DSRLVε	DSRAVε
3	MULT	MULTU	DIV	DIVU	DMULTε	DMULTUε	DDIVε	DDIVUε
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	*	*	SLT	SLTU	DADDε	DADDUε	DSUBε	DSUBUε
6	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	DSLLε	*	DSRLε	DSRAε	DSLL32ε	*	DSRL32ε	DSRA32ε

REGIMM rt

20..19	18..16 0	1	2	3	4	5	6	7
0	BLTZ	BGEZ	BLTZL	BGEZL	*	*	*	*
1	TGEI	TGEIU	TLTI	TLTIU	TEQI	*	TNEI	*
2	BLTZAL	BGEZAL	BLTZALL	BGEZALL	*	*	*	*
3	*	*	*	*	*	*	*	*

COPz rs

25, 24	23..21 0	1	2	3	4	5	6	7
0	MF	DMFε	CF	γ	MT	DMTε	CT	γ
1	BC	γ	γ	γ	γ	γ	γ	γ
2	CO							
3								

Figure A-2 R4000 Opcode Bit Encoding

COPz rt

20..19 \ 18..16	0	1	2	3	4	5	6	7
0	BCF	BCT	BCFL	BCTL	γ	γ	γ	γ
1	γ	γ	γ	γ	γ	γ	γ	γ
2	γ	γ	γ	γ	γ	γ	γ	γ
3	γ	γ	γ	γ	γ	γ	γ	γ

CP0 Function

5 .. 3 \ 2 .. 0	0	1	2	3	4	5	6	7
0	φ	TLBR	TLBWI	φ	φ	φ	TLBWR	φ
1	TLBP	φ	φ	φ	φ	φ	φ	φ
2	ξ	φ	φ	φ	φ	φ	φ	φ
3	ERET χ	φ	φ	φ	φ	φ	φ	φ
0	φ	φ	φ	φ	φ	φ	φ	φ
1	φ	φ	φ	φ	φ	φ	φ	φ
2	φ	φ	φ	φ	φ	φ	φ	φ
3	φ	φ	φ	φ	φ	φ	φ	φ

Figure A-2 R4000 Opcode Bit Encoding (cont.)

Key:

*	Operation codes marked with an asterisk cause reserved instruction exceptions in all current implementations and are reserved for future versions of the architecture.
γ	Operation codes marked with a gamma cause a reserved instruction exception. They are reserved for future versions of the architecture.
δ	Operation codes marked with a delta are valid only for R4000 processors with CP0 enabled, and cause a reserved instruction exception on other processors.
φ	Operation codes marked with a phi are invalid but do not cause reserved instruction exceptions in R4000 implementations.
ξ	Operation codes marked with a xi cause a reserved instruction exception on R4000 processors.
χ	Operation codes marked with a chi are valid only on R4000.
ε	Operation codes marked with epsilon are valid when the processor operating as a 64-bit processor. These instructions will cause a reserved instruction exception if 64-bit operation is not enabled.

FPU Instruction Set Details

B

This appendix provides a detailed description of the operation of each Floating-Point (FPU) instruction. The instructions are listed alphabetically. The exceptions that may occur due to the execution of each instruction are listed after the description of each instruction. The description of the immediate causes and the manner of handling exceptions is omitted from the instruction descriptions in this chapter. Refer to Chapter 8 for detailed descriptions of floating-point exceptions and handling.

Figure B-5 lists the entire bit encoding for the constant fields of the Floating-Point instruction set; the bit encoding for each instruction is included with that individual instruction.

Instruction Formats

There are three basic instruction format types:

- I-Type, or Immediate instructions, which include load and store operations,
- M-Type, or Move instructions, and
- R-Type, or Register instructions, which include the two- and three-register Floating-Point operations.

The instruction description subsections that follow show how the three basic instruction formats are used by:

- Load and store instructions,
- Move instructions, and
- Floating-Point Computational instructions.

A fourth instruction description subsection describes the special instruction format used by floating-point branch instructions.

Floating-point instructions are mapped onto the MIPS coprocessor instructions, defining coprocessor unit number one (CP1) as the floating-point unit.

Each operation is valid only for certain formats. Implementations may support some of these formats and operations only through emulation, but only need support combinations that are valid, which are marked with a V in Table B-1 below. Those combinations marked with a "R" are not currently specified by this architecture, causing an unimplemented instruction trap, to maintain compatibility with future architecture extensions.

Table B-1 Valid FPU Instruction Formats

Operation	Source Format			
	Single	Double	Word	Longword
ADD	V	V	R	R
SUB	V	V	R	R
MUL	V	V	R	R
DIV	V	V	R	R
SQRT	V	V	R	R
ABS	V	V	R	R
MOV	V	V		
NEG	V	V	R	R
TRUNC.L	V	V		
ROUND.L	V	V		
CEIL.L	V	V		
FLOOR.L	V	V		
TRUNC.W	V	V		
ROUND.W	V	V		
CEIL.W	V	V		
FLOOR.W	V	V		
CVT.S		V	V	V
CVT.D	V		V	V
CVT.W	V	V		
CVT.L	V	V		
C	V	V	R	R

The coprocessor branch on condition true/false instructions can be used to logically negate any predicate. Thus, the 32 possible conditions require only 16 distinct comparisons, as shown in Table B-2 below.

Table B-2 Logical Negation of Predicates by Condition True/False

Condition			Relations				Invalid operation exception if unordered
Mnemonic		Code	Greater Than	Less Than	Equal	Unordered	
True	False						
F	T	0	F	F	F	F	No
UN	OR	1	F	F	F	T	No
EQ	NEQ	2	F	F	T	F	No
UEQ	OGL	3	F	F	T	T	No
OLT	UGE	4	F	T	F	F	No
ULT	OGE	5	F	T	F	T	No
OLE	UGT	6	F	T	T	F	No
ULE	OGT	7	F	T	T	T	No
SF	ST	8	F	F	F	F	Yes
NGLE	GLE	9	F	F	F	T	Yes
SEQ	SNE	10	F	F	T	F	Yes
NGL	GL	11	F	F	T	T	Yes
LT	NLT	12	F	T	F	F	Yes
NGE	GE	13	F	T	F	T	Yes
LE	NLE	14	F	T	T	F	Yes
NGT	GT	15	F	T	T	T	Yes

Floating-Point Loads, Stores, and Moves

All movement of data between the floating-point coprocessor and memory is accomplished by coprocessor load and store operations, which reference the floating-point coprocessor's *General-Purpose* Registers. These operations are unformated; no format conversions are performed and, therefore, no floating-point exceptions occur due to these operations.

Data may also be directly moved between the floating-point coprocessor and the processor by move to coprocessor and move from coprocessor instructions. Like the floating-point load and store operations, move to/from operations perform no format conversions and never cause floating-point exceptions.

An additional pair of coprocessor registers are available, called *Floating-Point Control* registers for which the only data movement operations supported are moves to and from processor *General-Purpose* Registers.

Floating-Point Operations

The floating-point unit's operation set includes floating-point add, subtract, multiply, divide, square root, convert between fixed-point and floating-point format, convert between floating-point formats, and floating-point compare. These operations satisfy IEEE Standard 754's requirements for accuracy. Specifically, these operations obtain a result which is identical to performing the result with infinite precision and then rounding to the specified format, using the current rounding mode.

Instructions must specify the format of their operands. Except for conversion functions, mixed-format operations are not provided.

Instruction Notational Conventions

In this appendix, all variable subfields in an instruction format (such as *fs, ft, immediate,* and so on) are shown with lower-case names. The instruction name (such as ADD, SUB, and so on) is shown in upper-case.

For the sake of clarity, an alias is sometimes substituted for a variable subfield in the formats of specific instructions. For example, we use *rs = base* in the format for load and store instructions. Such an alias is always lower case, since it refers to a variable subfield.

In some instructions, however, the two instruction subfields *op* and *function* have constant 6-bit values. When reference is made to these instructions, upper-case mnemonics are used. In the floating-point instruction, for example, we use *op* = COP1 and *function* = FADD. In some cases, a single field has both fixed and variable subfields, so the name contains both upper and lower case characters. Actual bit encoding for mnemonics is shown in Figure B-5 at the end of this appendix, and are also included with each individual instruction.

In the instruction description examples that follow, the *Operation* section describes the operation performed by each instruction using a high-level language notation.

Instruction Notation Examples

Example #1:

$$\text{GPR[ft]} \quad \leftarrow \quad \text{immediate} \;||\; 0^{16}$$

Sixteen zero bits are concatenated with an immediate value (typically 16 bits), and the 32-bit string (with the lower 16 bits set to zero) is assigned to GPR register *ft.*

Example #2:

$$(\text{immediate}_{15})^{16} \;||\; \text{immediate}_{15..0}$$

Bit 15 (the sign bit) of an immediate value is extended for 16 bit positions, and the result is concatenated with bits 15 through 0 of the immediate value to form a 32-bit sign-extended value.

Load and Store Instructions

In the MIPS ISA, all load operations have a delay of at least one instruction. That is, the instruction immediately following a load cannot use the contents of the register that will be loaded with the data being fetched from storage.

In the R4000, the instruction immediately following a load may use the contents of the register loaded. In such cases, the hardware will interlock, requiring additional real cycles, so scheduling load delay slots is still desirable, although not absolutely required for functional code.

When the FR bit in the *Status* register equals zero, the *Floating-Point General Registers* (*FGR*) are 32-bits wide. When the FR bit in the *Status* register equals one, the *Floating-Point General Registers* (*FGR*) are 64-bits wide. The behavior of the load store insturctions in dependent on the width of the *FGR*s.

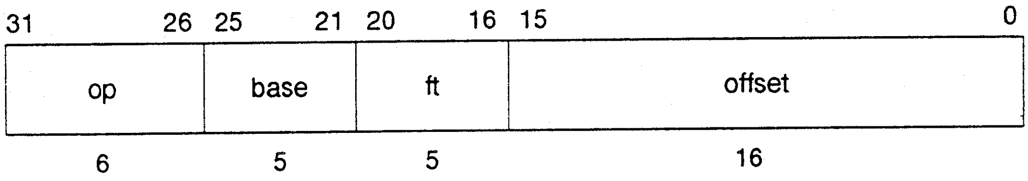
In the load/store operation descriptions, the functions listed in Table B-3 are used to summarize the handling of virtual addresses and physical memory.

Table B-3 Load/Store Common Functions

Function	Meaning
AddressTranslation	Uses the TLB to find the physical address given the virtual address. The function fails and an exception is taken if the required translation is not present in the TLB.
LoadMemory	Uses the cache and main memory to find the contents of the word containing the specified physical address. The low-order two bits of the address and the access type field indicates which of each of the four bytes within the data word need to be returned. If the cache is enabled for this access, the entire word is returned and loaded into the cache.
StoreMemory	Uses the cache, write buffer and main memory to store the word or part of word specified as data in the word containing the specified physical address. The low-order two bits of the address and the access type field indicates which of each of the four bytes within the data word should be stored.

Figure B-3 shows the I-Type instruction format used by load and store operations.

I-Type (Immediate)



where:

- op is a 6-bit operation code
- base is the 5-bit base register specifier
- ft is a 5-bit source (for stores) or destination (for loads) FPA register specifier
- offset is the 16-bit signed immediate offset

Figure B-3 Load and Store Instruction Format

All coprocessor loads and stores reference aligned word data items. Thus, for word loads and stores, the access type field is always WORD, and the low-order two bits of the address must always be zero.

For doubleword loads and stores, the access type field is always DOUBLEWORD, and the low-order three bits of the address must always be zero.

Regardless of byte-numbering order (endianness), the address specifies that byte which has the smallest byte-address of all of the bytes in the addressed field. For a Big-endian machine, this is the leftmost byte; for a Little-endian machine, this is the rightmost byte.

Computational Instructions

Computational instructions include all of the arithmetic floating-point operations performed by the FPU.

Figure B-4 shows the R-Type instruction format used for computational operations.

R-Type (Register)

31 26	25 21	20 16	15 11	10 6	5 0
COP1	fmt	ft	fs	fd	function
6	5	5	5	5	6

where:

COP1	is a 6-bit major operation code
fmt	is a 5-bit format specifier
fs	is a 5-bit source1 register
ft	is a 5-bit source2 register
fd	is a 5-bit destination register
function	is a 6-bit function field

Figure B-4 Computational Instruction Format

Each floating-point instruction can be applied to a number of operand formats. The operand format for an instruction is specified by the 4-bit *Format* field; decoding for this field is shown in Table B-1.

Table B-1 Format Field Decoding

Code	Mnemonic	Size	Format
16	S	single	Binary floating-point
17	D	double	Binary floating-point
18	Reserved		
19	Reserved		
20	W	single	Binary fixed-point
21	L	longword	64-bit binary fixed-point
22–31	–	–	Reserved

The *function* indicates which floating-point operation is to be performed. Table B-2 lists all floating-point instructions.

Table B-2 Floating-Point Instructions and Operations

Code (5..0)	Mnemonic	Operation
0	ADD	Add
1	SUB	Subtract
2	MUL	Multiply
3	DIV	Divide
4	SQRT	Square root
5	ABS	Absolute value
6	MOV	Move
7	NEG	Negate
8	ROUND.L	Convert to single fixed-point, rounded to nearest/even
9	TRUNC.L	Convert to single fixed-point, rounded toward zero
10	CEIL.L	Convert to single fixed-point, rounded to +∞
11	FLOOR.L	Convert to single fixed-point, rounded to -∞
12	ROUND.W	Convert to single fixed-point, rounded to nearest/even
13	TRUNC.W	Convert to single fixed-point, rounded toward zero
14	CEIL.W	Convert to single fixed-point, rounded to + ∞
15	FLOOR.W	Convert to single fixed-point, rounded to – ∞
16–31	–	Reserved
32	CVT.S	Convert to single floating-point
33	CVT.D	Convert to double floating-point
34	–	Reserved
35	–	Reserved
36	CVT.W	Convert to binary fixed-point
37	CVT.L	Convert to 64-bit binary fixed-point
38–47	–	Reserved
48–63	C	Floating-point compare

In the following pages, the notation FGR refers to the FPU's 32 General-Purpose Registers FGR0 through FGR31, and FPR refers to the FPU's Floating-Point Registers. When the FR bit in the *Status* register (SR$_{26}$) equals zero, only the even Floating-Point Registers are valid and the FPU's 32 General-Purpose Registers are 32-bits wide. When the FR bit in the *Status* register (SR$_{26}$) equals one, both odd and even Floating-Point Registers may be used and the FPU's 32 General-Purpose Registers are 64-bits wide.

The following routines are used in the description of the floating-point operations to get the value of an FPR or to change the value of an FGR:

```
32 Bit Mode

    value <-- ValueFPR(fpr, fmt)
        /* undefined for odd fpr */
        case fmt of
            S, W:
                value <-- FGR[fpr+0]
            D:
                /* undefined for fpr not even */
                value <-- FGR[fpr+1] || FGR[fpr+0]
        end

    StoreFPR(fpr, fmt, value):
        /* undefined for odd fpr */
        case fmt of
            S, W:
                FGR[fpr+1] <-- undefined
                FGR[fpr+0] <-- value
            D:
                FGR[fpr+1] <-- value$_{63..32}$
                FGR[fpr+0] <-- value$_{31..0}$
        end
```

```
64 Bit Mode
    value <-- ValueFPR(fpr, fmt)
        case fmt of
            S:
                value <-- $FGR[fpr]_{31..0}$
            D, L:
                value <-- FGR[fpr]
            W:
                value <-- FGR[fpr]
        end

    StoreFPR(fpr, fmt, value):
        case fmt of
            S, W:
                FGR[fpr] <-- $undefined^{32}$ || value
            D, L:
                FGR[fpr] <-- value
        end
```

ABS.fmt Floating-Point Absolute Value ABS.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	ABS 000101
6	5	5	5	5	6

Format:

ABS.fmt fd, fs

Description:

The contents of the FPU register specified by *fs* are interpreted in the specified format and the arithmetic absolute value is taken. The result is placed in the floating-point register specified by *fd*.

The absolute value operation is arithmetic; a NaN operand signals invalid operation.

This instruction is valid only for single- and double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

Operation:

```
T:     StoreFPR(fd, fmt, AbsoluteValue(ValueFPR(fs, fmt)))
```

Exceptions:

Coprocessor unusable exception
Coprocessor exception trap

Coprocessor Exceptions:

Unimplemented operation exception
Invalid operation exception

ADD.fmt Floating-Point Add ADD.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	ft	fs	fd	ADD 0 0 0 0 0 0
6	5	5	5	5	6

Format:

ADD.fmt fd, fs, ft

Description:

The contents of the FPU registers specified by *fs* and *ft* are interpreted in the specified format and arithmetically added. The result is rounded as if calculated to infinite precision and then rounded to the specified format (*fmt*), according to the current rounding mode. The result is placed in the floating-point register (*FPR*) specified by *fd*.

This instruction is valid only for single- and double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

Operation:

```
T:      StoreFPR (fd, fmt, ValueFPR(fs, fmt) + ValueFPR(ft, fmt))
```

Exceptions:

Coprocessor unusable exception
Floating-Point exception

Coprocessor Exceptions:

Unimplemented operation exception
Invalid operation exception
Inexact exception
Overflow exception
Underflow exception

BC1F Branch On FPA False (coprocessor 1) BC1F

31 26	25 21	20 16	15 0
COP1 0 1 0 0 0 1	BC 0 1 0 0 0	BCF 0 0 0 0 0	offset
6	5	5	16

Format:

BC1F offset

Description:

A 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. If the result of the last floating-point compare is false, the program branches to the target address, with a delay of one instruction.

Operation:

```
32      T–1:  condition ← not COC[1]
        T:    target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
        T+1:  if condition then
                   PC ← PC + target
              endif

64      T–1:  condition ← not COC[1]
        T:    target ← $(\text{offset}_{15})^{38}$ || offset || $0^2$
        T+1:  if condition then
                   PC ← PC + target
              endif
```

Exceptions:

Coprocessor unusable exception

BC1FL Branch On FPU False Likely (coprocessor 1) BC1FL

31 26	25 21	20 16	15 0
COP1 0 1 0 0 0 1	BC 0 1 0 0 0	BCFL 0 0 0 1 0	offset
6	5	5	16

Format:

BC1FL offset

Description:

A 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.

If the result of the last floating-point compare is false, the program branches to the target address, with a delay of one instruction. If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

Operation:

```
32    T–1:  condition ← not COC[1]
      T:    target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
      T+1:  if condition then
                PC ← PC + target
            else
                NullifyCurrentInstruction
            endif
64    T–1:  condition ← not COC[1]
      T:    target ← $(\text{offset}_{15})^{38}$ || offset || $0^2$
      T+1:  if condition then
                PC ← PC + target
      else
                NullifyCurrentInstruction
      endif
```

Exceptions:

Coprocessor unusable exception

BC1T Branch On FPU True (coprocessor 1) BC1T

31 26	25 21	20 16	15 0
COP1 0 1 0 0 0 1	BC 0 1 0 0 0	BCT 0 0 0 0 1	offset
6	5	5	16

Format:

BC1T offset

Description:

A 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. If the result of the last floating-point compare is true, the program branches to the target address, with a delay of one instruction.

Operation:

```
32      T–1:  condition ← COC[1]
        T:    target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
        T+1:  if condition then
                    PC ← PC + target
              endif

64      T–1:  condition ← COC[1]
        T:    target ← $(\text{offset}_{15})^{38}$ || offset || $0^2$
        T+1:  if condition then
                PC ← PC + target
              endif
```

Exceptions:

Coprocessor unusable exception

BC1TL Branch On FPU True Likely (coprocessor 1) BC1TL

31 26	25 21	20 16	15 0
COP1 0 1 0 0 0 1	BC 0 1 0 0 0	BCTL 0 0 0 1 1	offset
6	5	5	16

Format:

BC1TL offset

Description:

A 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.

If the result of the last floating-point compare is true, the program branches to the target address, with a delay of one instruction. If the conditional branch is *not* taken, the instruction in the branch delay slot is nullified.

Operation:

```
32      T–1:  condition ← COC[1]
        T:    target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
        T+1:  if condition then
                        PC ← PC + target
              else
                        NullifyCurrentInstruction
              endif
64      T–1:  condition ← COC[1]
        T:    target ← $(\text{offset}_{15})^{38}$ || offset || $0^2$
        T+1:  if condition then
                        PC ← PC + target
              else
                        NullifyCurrentInstruction
              endif
```

Exceptions:

Coprocessor unusable exception

C.cond.fmt Floating-Point Compare C.cond.fmt

31 26	25 21	20 16	15 11	10 6	5 4	3 0
COP1 010001	fmt	ft	fs	0 00000	FC*	cond*
6	5	5	5	5	2	4

Format:

C.cond.fmt fs, ft

Description:

The contents of the floating-point registers specified by *fs* and *ft* are interpreted in the specified *format* and arithmetically compared.

A result is determined based on the comparison and the conditions specified in the instruction. If one of the values is a Not a Number (NaN), and the high-order bit of the *condition* field is set, an invalid operation exception is taken. After a one-instruction delay, the condition is available for testing with branch on floating-point coprocessor condition instructions.

Comparisons are exact and can neither overflow nor underflow. Four mutually exclusive relations are possible results: less than, equal, greater than, and unordered. The last case arises when one or both of the operands are NaN; every NaN compares unordered with everything, including itself. Comparisons ignore the sign of zero, so +0 = –0.

This instruction is valid only for single- and double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

*See "FPU Instruction Opcode Bit Encoding" at the end of Appendix B.

C.cond.fmt Floating-Point Compare (continued) C.cond.fmt

Operation:

```
T:    if NaN(ValueFPR(fs, fmt)) or NaN(ValueFPR(ft, fmt)) then
          less ← false
          equal ← false
          unordered ← true
          if $cond_3$ then
              signal InvalidOperationException
          endif
      else
          less ← ValueFPR(fs, fmt) < ValueFPR(ft, fmt)
          equal ← ValueFPR(fs, fmt) = ValueFPR(ft, fmt)
          unordered ← false
      endif
      condition ← ($cond_2$ and less) or ($cond_1$ and equal) or
          ($cond_0$ and unordered)
      FCR[31]$_{23}$ ← condition
      COC[1] ← condition
```

Exceptions:

Coprocessor unusable
Floating-Point exception

Coprocessor Exceptions:

Unimplemented operation exception
Invalid operation exception

CEIL.L.fmt Floating-Point Ceiling to Long Fixed-Point Format CEIL.L.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	CEIL.L 001010
6	5	5	5	5	6

Format:

CEIL.L.fmt fd, fs

Description:

The contents of the floating-point register specified by *fs* are interpreted in the specified source format, *fmt*, and arithmetically converted to the single fixed-point format. The result is placed in the floating-point register specified by *fd*.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round to +∞ (2).

This instruction is valid only for conversion from single-, double-, extended or quad-precision floating-point formats. If extended or quad-precision format is specified, the operation is not defined if bit 0 of the source register specification is set, since the register number specifies an aligned coprocessor general register. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

When the source operand is an Infinity, NaN, or the correctly rounded integer result is outside of -2^{63} to $2^{63}-1$, the Invalid operation exception is raised. If the Invalid operation is not enabled then no exception is taken and $2^{63}-1$ is returned.

This instruction is not implemented on MIPS I or MIPS II processors, and will cause an unimplemented operation exception to occur.

CEIL.L.fmt Floating-Point Ceiling to Long Fixed-Point Format (continued) CEIL.L.fmt

Operation:

```
T:      StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Exceptions:

Coprocessor unusable exception
Coprocessor Interrupt (R2000, R3000, or R6000)
Floating-Point exception (R4000)

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

CEIL.W.fmt Floating-Point Ceiling to Single Fixed-Point Format CEIL.W.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	CEIL.W 0 0 1 1 1 0
6	5	5	5	5	6

Format:

CEIL.W.fmt fd, fs

Description:

The contents of the floating-point register specified by *fs* are interpreted in the specified source format, *fmt*, and arithmetically converted to the single fixed-point format. The result is placed in the floating-point register specified by *fd*.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round to +∞ (2).

This instruction is valid only for conversion from a single- or double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

When the source operand is an Infinity or NaN, or the correctly rounded integer result is outside of -2^{31} to $2^{31} - 1$, the Invalid operation exception is raised. If the Invalid operation is not enabled then no exception is taken and $2^{31} - 1$ is returned.

CEIL.W.fmt Floating-Point Ceiling to Single Fixed-Point Format (continued) CEIL.W.fmt

Operation:

```
T:      StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

Exceptions:

Coprocessor unusable exception
Floating-Point exception

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

CFC1 Move Control Word From FPU (coprocessor 1) CFC1

31 26	25 21	20 16	15 11	10 0
COP1 0 1 0 0 0 1	CF 0 0 0 1 0	rt	fs	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format:

CFC1 rt, fs

Description:

The contents of the FPU's control register *fs* are loaded into general register *rt*.

This operation is only defined when *fs* equals 0 or 31.

The contents of general register *rt* are undefined for time *T* of the instruction immediately following this load instruction.

Operation:

```
32      T:      temp ← FCR[fs]
        T+1 :   GPR[rt] ← temp

64      T:      temp ← FCR[fs]
        T+1 :   GPR[rt] ← $(temp_{31})^{32}$ || temp
```

Exceptions:

Coprocessor unusable exception

CTC1 — Move Control Word To FPU (coprocessor 1) — CTC1

31 26	25 21	20 16	15 11	10 0
COP1 0 1 0 0 0 1	CT 0 0 1 1 0	rt	fs	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format:

CTC1 rt, fs

Description:

The contents of general register *rt* are loaded into the FPU's control register *fs*. This operation is only defined when *fs* equals 0 or 31.

Writing to *Control Register 31*, the floating-point *Control/Status* register, causes an interrupt or exception if any cause bit and its corresponding enable bit are both set. The register will be written before the exception occurs. The contents of floating-point control register *fs* are undefined for time *T* of the instruction immediately following this load instruction.

Operation:

```
32    T:     temp ← GPR[rt]
      T+1:   FCR[fs] ← temp
             COC[1] ← $FCR[31]_{23}$

64    T:     temp ← $GPR[rt]_{31..0}$
      T+1:   FCR[fs] ← temp
             COC[1] ← $FCR[31]_{23}$
```

Exceptions:

Coprocessor unusable exception
Floating-Point exception

Coprocessor Exceptions:

Unimplemented operation exception
Invalid operation exception
Division by zero exception
Inexact exception
Overflow exception
Underflow exception

CVT.D.fmt Floating-Point Convert to Double Floating-Point Format CVT.D.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	CVT.D 100001
6	5	5	5	5	6

Format:

CVT.D.fmt fd, fs

Description:

The contents of the floating-point register specified by *fs* is interpreted in the specified source format, *fmt*, and arithmetically converted to the double binary floating-point format. The result is placed in the floating-point register specified by *fd*.

This instruction is valid only for conversions from single floating-point format, 32-bit or 64-bit fixed-point format.

If the single floating-point or single fixed-point format is specified, the operation is exact. The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

Operation:

```
T:     StoreFPR (fd, D, ConvertFmt(ValueFPR(fs, fmt), fmt, D))
```

Exceptions:

Coprocessor unusable exception
Floating-Point exception

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception
Underflow exception

CVT.L.fmt Floating-Point Convert to Long Fixed-Point Format CVT.L.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	CVT.L 100101
6	5	5	5	5	6

Format:

CVT.L.fmt fd, fs

Description:

The contents of the floating-point register specified by *fs* is interpreted in the specified source format, *fmt*, and arithmetically converted to the long fixed-point format. The result is placed in the floating-point register specified by *fd*.

This instruction is valid only for conversions from single-, double-, extended- or quard-precision floating-point formats. If extended- or quad-precision format is specified, the operation is not defined if bit 0 of the source register specification is set, since the register number specifies an aligned coprocessor general register.

When the source operand is an Infinity, NaN, or the correctly rounded integer result is outside of -2^{63} to $2^{63}-1$, the Invalid operation exception is raised. If the Invalid operation is not enabled then no exception is taken and $2^{63}-1$ is returned.

This instruction is not implemented on MIPS I or MIPS II processors, and will cause an unimplemented operation exception to occur.

CVT.L.fmt Floating-Point Convert to Long Fixed-Point Format CVT.L.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	CVT.L 100101
6	5	5	5	5	6

Operation:

```
T:    StoreFPR (fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Exceptions:

Coprocessor unusable exception
Coprocessor Interrupt (R2000, R3000, or R6000)
Floating-Point exception (R4000)

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

CVT.S.fmt Floating-Point Convert to Single Floating-Point Format CVT.S.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	CVT.S 1 0 0 0 0 0
6	5	5	5	5	6

Format:

CVT.S.fmt fd, fs

Description:

The contents of the floating-point register specified by *fs* are interpreted in the specified source format, *fmt*, and arithmetically converted to the single binary floating-point format. The result is placed in the floating-point register specified by *fd*. Rounding occurs according to the currently specified rounding mode.

This instruction is valid only for conversions from double floating-point format, or from 32-bit or 64-bit fixed-point format. The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

Operation:

```
T:      StoreFPR(fd, S, ConvertFmt(ValueFPR(fs, fmt), fmt, S))
```

Exceptions:

Coprocessor unusable exception
Floating-Point exception

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception
Underflow exception

CVT.W.fmt Floating-Point Convert to Fixed-Point Format CVT.W.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	CVT.W 100100
6	5	5	5	5	6

Format:

CVT.W.fmt fd, fs

Description:

The contents of the floating-point register specified by *fs* are interpreted in the specified source format, *fmt*, and arithmetically converted to the single fixed-point format. The result is placed in the floating-point register specified by *fd*.

This instruction is valid only for conversion from a single- or double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

When the source operand is an Infinity or NaN, or the correctly rounded integer result is outside of -2^{31} to $2^{31} - 1$, an Invalid operation exception is raised. If Invalid operation is not enabled, then no exception is taken and $2^{31} - 1$ is returned.

Operation:

```
T:    StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

Exceptions:

Coprocessor unusable exception
Floating-Point exception

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

DIV.fmt Floating-Point Divide DIV.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	ft	fs	fd	DIV 000011
6	5	5	5	5	6

Format:

DIV.fmt fd, fs, ft

Description:

The contents of the floating-point registers specified by *fs* and *ft* are interpreted in the specified *format* and arithmetically divided. The result is rounded as if calculated to infinite precision and then rounded to the specified format, according to the current rounding mode. The result is placed in the floating-point register specified by *fd*.

This instruction is valid for only single or double precision floating-point formats.

The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

Operation:

```
T:     StoreFPR (fd, fmt, $\frac{\text{ValueFPR(fs, fmt)}}{\text{ValueFPR(ft, fmt))}}$
```

Exceptions:

Coprocessor unusable exception
Floating-Point exception

Coprocessor Exceptions:

Unimplemented operation exception
Invalid operation exception
Division-by-zero exception
Inexact exception
Overflow exception
Underflow exception

DMFC1 Doubleword Move From Floating-Point Coprocessor DMFC1

31 26	25 21	20 16	15 11	10 0
COP1 0 1 0 0 0 1	DMF 0 0 0 0 1	rt	fs	0 0 0 0 0 0 0 0 0 0 00
6	5	5	5	11

Format:

DMFC1 rt, fs

Description:

The contents of register *fs* from the floating-point coprocessor is stored into processor register *rt*.

The contents of general register *rt* are undefined for time T of the instruction immediately following this load instruction.

The *FR* bit in the *Status* register specifies whether all 32 register of the R4000 are addressable. When FR is clear, this instruction is not defined when the least significant bit of *fs* is non-zero. When *FR* is set, *fs* may specify either odd or even registers.

Operation:

```
64      T:      if $SR_{26}$ = 1 then
                        data ← CPR[1,fs]
                else
                        data ← CPR[1,$fs_{4..1}$ || 0]
                endif

        T+1:    GPR[rt] ← data
```

Exceptions:

Coprocessor unusable exception

DMTC1 Doubleword Move To Floating-Point Coprocessor DMTC1

31 26	25 21	20 16	15 11	10 0
COP1 0 1 0 0 0 1	DMT 0 0 1 0 1	rt	fs	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format:

DMTC1 rt, fs

Description:

The contents of general register *rt* are loaded into coprocessor register*fs* of the CP1.

The contents of floating-point register *fs* are undefined for time *T* of the instruction immediately following this load instruction.

The *FR* bit in the *Status* register specifies whether all 32 register of the R4000 are addressable. When *FR* equals zero, this instruction is not defined when the least significant bit of *fs* is non-zero. When *FR* equals one, *fs* may specify either odd or even registers.

Operation:

```
64   T:    data ← GPR[rt]

     T+1:  if $SR_{26}$ = 1 then
                     CPR[1, fs] ← data
           else
                     CPR[1, $fs_{4..1}$ || 0] ← data
           endif
```

Exceptions:

Coprocessor unusable exception

FLOOR.L.fmt Floating-Point Floor to Long Fixed-Point Format FLOOR.L.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	FLOOR.L 001011
6	5	5	5	5	6

Format:

FLOOR.L.fmt fd, fs

Description:

The contents of the floating-point register specified by *fs* are interpreted in the specified source format, *fmt*, and arithmetically converted to the single fixed-point format. The result is placed in the floating-point register specified by *fd*.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round to $-\infty$ (3).

This instruction is valid only for conversion from single-, double-, extended or quad-precision floating-point formats. If extended or quad-precision format is specified, the operation is not defined if bit 0 of the source register specification is set, since the register number specifies an aligned coprocessor general register.

When the source operand is an Infinity, NaN, or the correctly rounded integer result is outside of -2^{63} to $2^{63}-1$, the Invalid operation exception is raised. If the Invalid operation is not enabled then no exception is taken and $2^{63}-1$ is returned.

This instruction is not implemented on MIPS I or MIPS II processors, and will cause an unimplemented operation exception to occur.

FLOOR.L.fmt Floating-Point Floor to Long Fixed-Point Format FLOOR.L.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	FLOOR.L 0 0 1 0 1 1
6	5	5	5	5	6

Operation:

```
T:    StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Exceptions:

Coprocessor unusable exception
Coprocessor Interrupt (R2000, R3000, or R6000)
Floating-Point exception (R4000)

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

FLOOR.W.fmt Floating-Point Floor to Single Fixed-Point Format FLOOR.W.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	FLOOR.W 001111
6	5	5	5	5	6

Format:

FLOOR.W.fmt fd, fs

Description:

The contents of the floating-point register specified by *fs* are interpreted in the specified source format, *fmt*, and arithmetically converted to the single fixed-point format. The result is placed in the floating-point register specified by *fd*.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round to $-\infty$ (RM = 3).

This instruction is valid only for conversion from a single- or double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

When the source operand is an Infinity or NaN, or the correctly rounded integer result is outside of -2^{31} to $2^{31} - 1$, an Invalid operation exception is raised. If Invalid operation is not enabled, then no exception is taken and $2^{31} - 1$ is returned.

FLOOR.W.fmt Floating-Point Floor to Single Fixed-Point Format FLOOR.W.fmt (continued)

Operation:

```
      T:     StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

Exceptions:

Coprocessor unusable exception
Floating-Point exception

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

LDC1 Load Doubleword to FPU (coprocessor 1) LDC1

31 26	25 21	20 16	15 0
LDC1 1 1 0 1 0 1	base	ft	offset
6	5	5	16

Format:

LDC1 ft, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form an unsigned effective address. In 32-bit mode, the contents of the doubleword at the memory location specified by the effective address is loaded into registers *ft* and *ft+1* of the floating-point coprocessor. This instruction is not valid, and is undefined, when the least significant bit of *ft* is non-zero. In 64-bit mode, the contents of the doubleword at the memory location specified by the effective address are loaded into the 64-bit register *ft* of the floating point coprocessor. The *FR* bit of the *Status* register (SR_{26}) specifies whether all 32 registers of the R4000 are addressable. When FR=0, this instruction is not defined when the least significant bit of *ft* is non-zero. When FR=1, *ft* may specify either odd or even registers.

If any of the three least-significant bits of the effective address are non-zero, an address error exception takes place.

LDC1 Load Doubleword to FPU (coprocessor 1) (continued) LDC1

Operation:

```
32    T:    vAddr ← $((\mathrm{offset}_{15})^{16} \| \mathrm{offset}_{15..0}) + \mathrm{GPR[base]}$
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            if BigEndianCPU = 1 then
                  CPR[1, ft+1] ← LoadMemory (uncached, WORD,
                                      pAddr+0, vAddr+0, DATA)
                  CPR[1, ft+0] ← LoadMemory (uncached, WORD,
                                      pAddr+4, vAddr+4, DATA)
            else
                  CPR[1, ft+0] ← LoadMemory (uncached, WORD,
                                      pAddr+0, vAddr+0, DATA)
                  CPR[1, ft+1] ← LoadMemory (uncached, WORD,
                                      pAddr+4, vAddr+4, DATA)
            endif

64    T:    vAddr ← $((\mathrm{offset}_{15})^{48} \| \mathrm{offset}_{15..0}) + \mathrm{GPR[base]}$
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            data ← LoadMemory(uncached, DOUBLEWORD, pAddr, vAddr, DATA)
            if $\mathrm{SR}_{26}$ = 1 then
                  CPR[1, ft] ← data
            else
                  CPR[1, $\mathrm{ft}_{4..1}$ || 0] ← data
            endif
```

Exceptions:

Coprocessor unusable
TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

LWC1 Load Word to FPU (coprocessor 1) LWC1

31 26	25 21	20 16	15 0
LWC1 110001	base	ft	offset
6	5	5	16

Format:

LWC1 ft, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form an unsigned effective address. The contents of the word at the memory location specified by the effective address is loaded into register *ft* of the floating-point coprocessor.

The *FR* bit of the *Status* register specifies whether all 64-bit *Floating-Point Registers* are addressable. If *FR* equals zero, LWC1 loads either the high or low half of the 16 even *Floating-Point Registers*. If *FR* equals one, LWC1 loads the low 32-bits of both even and odd *Floating-Point Registers*.

If either of the two least-significant bits of the effective address is nonzero, an address error exception occurs.

LWC1 Load Word to FPU (coprocessor 1) (continued) LWC1

Operation:

```
32    T:   vAddr ← $((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           CPR[1, ft] ← LoadMemory (uncached, WORD,
                                        pAddr, vAddr, DATA)

64    T:   vAddr ← $( (offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← $pAddr_{PSIZE-1..3} \| (pAddr_{2..0}$ xor (ReverseEndian $\| 0^2$))
           mem ← LoadMemory(uncached, WORD, pAddr, vAddr, DATA)
           byte ← $vAddr_{2..0}$ xor (BigEndianCPU $\| 0^2$)
           if $SR_{26} = 1$ then
               CPR[1, ft] ← $undefined^{32} \| mem_{31+8\#byte..8\#byte}$
           else if $ft_0$=0 then
               $CPR[1, ft_{4..1} \| 0]$ ← $CPR[1, ft_{4..1} \| 0]_{64..32} \| mem_{31+8\#byte..8\#byte}$
           else
               $CPR[1, ft_{4..1} \| 0]$ ← $mem_{31+8\#byte..8*byte} \| CPR[1, ft_{4..1} \| 0]_{31..0}$
           endif
```

Exceptions:

Coprocessor unusable
TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

MFC1 — Move From FPU (coprocessor 1) — MFC1

31 26	25 21	20 16	15 11	10 0
COP1 0 1 0 0 0 1	MF 0 0 0 0 0	rt	fs	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format:

MFC1 rt, fs

Description:

The contents of register *fs* from the floating-point coprocessor are stored into processor register *rt*.

The contents of register *rt* are undefined for time *T* of the instruction immediately following this load instruction.

The *FR* bit of the *Status* register specifies whether all 32 registers of the R4000 are addressable. If *FR* equals zero, MFC1 stores either the high or low half of the 16 even *Floating-Point Registers*. If *FR* equals one, MFC1 stores the low 32-bits of both even and odd *Floating-Point Registers*.

Operation:

```
32    T:      data ← CPR[1, fs];
      T+1:    GPR[rt] ← data

64    T:      if $fs_0$=0 then
                  data ← $\text{CPR}[1, fs_{4..1} \| 0]_{31..0}$
              else
                  data ← $\text{CPR}[1, fs_{4..1} \| 0]_{63..32}$
              endif
      T+1:    GPR[rt] ← $(data_{31})^{32}$ || data
```

Exceptions:

Coprocessor unusable exception

MOV.fmt Floating-Point Move MOV.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	MOV 000110
6	5	5	5	5	6

Format:

MOV.fmt fd, fs

Description:

The contents of the FPU register specified by *fs* are interpreted in the specified *format* and are copied into the FPU register specified by *fd*.

The move operation is non-arithmetic; no IEEE 754 exceptions occur as a result of the instruction.

This instruction is valid only for single- or double-precision floating-point formats.

The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

Operation:

```
T:     StoreFPR(fd, fmt, ValueFPR(fs, fmt))
```

Exceptions:

Coprocessor unusable exception
Floating-Point exception

Coprocessor Exceptions:

Unimplemented operation exception

MTC1 **Move To FPU (coprocessor 1)** MTC1

31 26	25 21	20 16	15 11	10 0
COP1 010001	MT 00100	rt	fs	0 000 0000 0000
6	5	5	5	11

Format:

MTC1 rt, fs

Description:

The contents of register *rt* are loaded into the FPU's general register at location *fs*.

The contents of floating-point register *fs* is undefined for time *T* of the instruction immediately following this load instruction.

The *FR* bit of the *Status* register specifies whether all 32 registers of the R4000 are addressable. If *FR* equals zero, MTC1 loads either the high or low half of the 16 even *Floating-Point Registers*. If *FR* equals one, MTC1 loads the low 32-bits of both even and odd *Floating-Point Registers*.

Operation:

```
32    T:    data ← GPR[rt]
      T+1:  CPR[1, fs] ← data

64    T:    data ← GPR[rt]$_{31..0}$
      T+1:  if SR$_{26}$ = 1 then
                 CPR[1, fs] ← undefined$^{32}$ || data
            else if fs$_0$=0 then
                 CPR[1, fs$_{4..1}$ || 0] ← CPR[1, fs$_{4..1}$ || 0]$_{63..32}$ || data
            else
                 CPR[1, fs$_{4..1}$ || 0] ← data || CPR[1, fs$_{4..1}$ || 0]$_{31..0}$
            endif
```

Exceptions:

Coprocessor unusable exception

MUL.fmt Floating-Point Multiply MUL.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	ft	fs	fd	MUL 0 0 0 0 1 0
6	5	5	5	5	6

Format:

MUL.fmt fd, fs, ft

Description:

The contents of the floating-point registers specified by *fs* and *ft* are interpreted in the specified *format* and arithmetically multiplied. The result is rounded as if calculated to infinite precision and then rounded to the specified *format*, according to the current rounding mode. The result is placed in the floating-point register specified by *fd*.

This instruction is valid only for single- or double-precision floating-point formats.

The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

Operation:

```
T:    StoreFPR (fd, fmt, ValueFPR(fs, fmt) * ValueFPR(ft, fmt))
```

Exceptions:

Coprocessor unusable exception
Floating-Point exception

Coprocessor Exceptions:

Unimplemented operation exception
Invalid operation exception
Inexact exception
Overflow exception
Underflow exception

NEG.fmt Floating-Point Negate NEG.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	NEG 000111
6	5	5	5	5	6

Format:

NEG.fmt fd, fs

Description:

The contents of the FPU register specified by *fs* are interpreted in the specified *format* and the arithmetic negation is taken (the polarity of the sign-bit is changed). The result is placed in the FPU register specified by *fd*.

The negate operation is arithmetic; an NaN operand signals invalid operation.

This instruction is valid only for single- or double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

Operation:

```
T:     StoreFPR(fd, fmt, AbsoluteValue(ValueFPR(fs, fmt)))
```

Exceptions:

Coprocessor unusable exception
Floating-Point exception

Coprocessor Exceptions:

Unimplemented operation exception
Invalid operation exception

ROUND.L.fmt Floating-Point Round to Long Fixed-Point Format ROUND.L.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	ROUND.L 001000
6	5	5	5	5	6

Format:

ROUND.L.fmt fd, fs

Description:

The contents of the floating-point register specified by *fs* are interpreted in the specified source format, *fmt*, and arithmetically converted to the long fixed-point format. The result is placed in the floating-point register specified by *fd*.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round to nearest/ even (0).

This instruction is valid only for conversion from single-, double-, extended or quad-precision floating-point formats. If extended or quad-precision format is specified, the operation is not defined if bit 0 of the source register specification is set, since the register number specifies an aligned coprocessor general register.

When the source operand is an Infinity , NaN, or the correctly rounded integer result is outside of -2^{63} to $2^{63}-1$, the Invalid operation exception is raised. If the Invalid operation is not enabled then no exception is taken and $2^{63}-1$ is returned.

This instruction is not implemented on MIPS I or MIPS II processors, and will cause an unimplemented operation exception to occur.

ROUND.L.fmt Floating-Point Round to Long Fixed-Point Format ROUND.L.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	ROUND.L 001000
6	5	5	5	5	6

Operation:

```
T:      StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Exceptions:

Coprocessor unusable exception
Coprocessor Interrupt (R2000, R3000, or R6000)
Floating-Point exception (R4000)

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

ROUND.W.fmt Floating-Point Round to Single Fixed-Point Format ROUND.W.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	ROUND.W 001100
6	5	5	5	5	6

Format:

ROUND.W.fmt fd, fs

Description:

The contents of the floating-point register specified by *fs* are interpreted in the specified source format, *fmt*, and arithmetically converted to the single fixed-point format. The result is placed in the floating-point register specified by *fd*.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round to nearest/even (RM = 0).

This instruction is valid only for conversion from a single- or double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

When the source operand is an Infinity or NaN, or the correctly rounded integer result is outside of -2^{31} to $2^{31} - 1$, an Invalid operation exception is raised. If Invalid operation is not enabled, then no exception is taken and $2^{31} - 1$ is returned.

ROUND.W.fmt Floating-Point Round to Single Fixed-Point Format (continued) ROUND.W.fmt

Operation:

```
T:      StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

Exceptions:

Coprocessor unusable exception
Floating-Point exception

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

SDC1 Store Doubleword from FPU (coprocessor 1) SDC1

31 26	25 21	20 16	15 0
SDC1 1 1 1 1 0 1	base	ft	offset
6	5	5	16

Format:

SDC1 ft, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form an unsigned effective address.

In 32-bit mode, the contents of registers *ft* and *ft+1* from the floating-point coprocessor are stored at the memory location specified by the effective address. This instruction is not valid, and is undefined, when the least significant bit of *ft* is non-zero.

In 64-bit mode, the 64-bit register *ft* is stored to the contents of the doubleword at the memory location specified by the effective address. The *FR* bit of the *Status* register (SR_{26}) specifies whether all 32 registers of the R4000 are addressable. When FR=0, this instruction is not defined if the least significant bit of *ft* is non-zero. If FR=1, *ft* may specify either odd or even registers.

If any of the three least-significant bits of the effective address are non-zero, an address error exception takes place.

SDC1 Store Doubleword from FPU (coprocessor 1) (continued) SDC1

Operation:

```
32     T:     vAddr ← $(\text{offset}_{15})^{16}$ || $\text{offset}_{15..0}$) + GPR[base]
              (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
              if BigEndianCPU = 1 then
                       StoreMemory (uncached, WORD, CPR[1, ft+1],
                                pAddr+0, vAddr+0, DATA)
                       StoreMemory (uncached, WORD, CPR[1, ft+0],
                                pAddr+4, vAddr+4, DATA)
              else
                       StoreMemory (uncached, WORD, CPR[1, ft+0],
                                pAddr+0, vAddr+0, DATA)
                       StoreMemory (uncached, WORD, CPR[1, ft+1],
                                pAddr+4, vAddr+4, DATA)
              endif

64     T:     vAddr ← $(\text{offset}_{15})^{16}$ || $\text{offset}_{15..0}$) + GPR[base]
              (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
              if SR26 = 1
                       data ← CPR[1, ft]
              else
                       data ← CPR[1, ft4..1 || 0)
              endif
              StoreMemory(uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
```

Exceptions:

Coprocessor unusable
TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception

SQRT.fmt Floating-Point Square Root SQRT.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	SQRT 0 0 0 1 0 0
6	5	5	5	5	6

Format:

SQRT.fmt fd, fs

Description:

The contents of the floating-point register specified by *fs* are interpreted in the specified *format* and the positive arithmetic square root is taken. The result is rounded as if calculated to infinite precision and then rounded to the specified *format*, according to the current rounding mode. If the value of *fs* corresponds to –0, the result will be –0. The result is placed in the floating-point register specified by *fd*.

This instruction is valid only for single- or double-precision floating-point formats.

The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

Operation:

```
T:    StoreFPR(fd, fmt, SquareRoot(ValueFPR(fs, fmt)))
```

Exceptions:

Coprocessor unusable exception
Floating-Point exception

Coprocessor Exceptions:

Unimplemented operation exception
Invalid operation exception
Inexact exception

SUB.fmt Floating-Point Subtract SUB.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	ft	fs	fd	SUB 0 0 0 0 0 1
6	5	5	5	5	6

Format:

SUB.fmt fd, fs, ft

Description:

The contents of the floating-point registers specified by *fs* and *ft* are interpreted in the specified *format* and arithmetically subtracted. The result is rounded as if calculated to infinite precision and then rounded to the specified *format*, according to the current rounding mode. The result is placed in the floating-point register specified by *fd*.

This instruction is valid only for single- or double-precision floating-point formats.

The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

Operation:

```
T:    StoreFPR (fd, fmt, ValueFPR(fs, fmt)  – ValueFPR (ft, fmt))
```

Exceptions:

Coprocessor unusable exception
Floating-Point exception

Coprocessor Exceptions:

Unimplemented operation exception
Invalid operation exception
Inexact exception
Overflow exception
Underflow exception

SWC1 Store Word from FPU (coprocessor 1) SWC1

31 26	25 21	20 16	15 0
SWC1 111001	base	ft	offset
6	5	5	16

Format:

SWC1 ft, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form an unsigned effective address. The contents of register *ft* from the floating-point coprocessor are stored at the memory location specified by the effective address.

The *FR* bit of the *Status* register specifies whether all 64-bit *Floating-Point Registers* are addressable. If *FR* equals zero, SWC1 stores either the high or low half of the 16 even *Floating-Point Registers.* If *FR* equals one, SWC1 stores the low 32-bits of both even and odd *Floating-Point Registers.*

If either of the two least-significant bits of the effective address are non-zero, an address error exception occurs.

SWC1 Store Word from FPU (coprocessor 1) (continued) SWC1

Operation:

```
32    T:   vAddr ← $((offset_{15})^{16} \,||\, offset_{15..0}) + GPR[base]$
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           data ← CPR[1, ft]
           StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)

64    T:   vAddr ← $((offset_{15})^{48} \,||\, offset_{15..0}) + GPR[base]$
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← $pAddr_{PSIZE-1..3} \,||\, (pAddr_{2..0}$ xor (ReverseEndian || $0^2$))
           byte ← $vAddr_{2..0}$ xor (BigEndianCPU || $0^2$)
           if $SR_{26}$ = 1 then
                 data ← $CPR[1, ft]_{63-8*byte..0} \,||\, 0^{8*byte}$
           else if $ft_0$=0 then
                 data  ← $CPR[1, ft_{4..1} \,||\, 0]_{63-8*byte..0} \,||\, 0^{8*byte}$
           else
                 data ← $0^{32-8*byte} \,||\, CPR[1, ft_{4..1} \,||\, 0]_{63..32-8*byte}$
           endif8*byte
           StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
```

Exceptions:

Coprocessor unusable
TLB refill exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception

TRUNC.L.fmt Floating-Point Truncate to Long Fixed-Point Format TRUNC.L.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	TRUNC.L 001001
6	5	5	5	5	6

Format:

TRUNC.L.fmt fd, fs

Description:

The contents of the floating-point register specified by *fs* are interpreted in the specified source format, *fmt*, and arithmetically converted to the single fixed-point format. The result is placed in the floating-point register specified by *fd*.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round toward zero (1).

This instruction is valid only for conversion from single-, double-, extended or quad-precision floating-point formats. If extended or quad-precision format is specified, the operation is not defined if bit 0 of the source register specification is set, since the register number specifies an aligned coprocessor general register.

When the source operand is an Infinity, NaN, or the correctly rounded integer result is outside of -2^{63} to $2^{63}-1$, the Invalid operation exception is raised. If the Invalid operation is not enabled then no exception is taken and $2^{63}-1$ is returned.

This instruction is not implemented on MIPS I or MIPS II processors, and will cause an unimplemented operation exception to occur.

TRUNC.L.fmt Floating-Point Truncate to Long Fixed-Point Format TRUNC.L.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	TRUNC.L 0 0 1 0 0 1
6	5	5	5	5	6

Operation:

```
T:      StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Exceptions:

Coprocessor unusable exception
Coprocessor Interrupt (R2000, R3000, or R6000)
Floating-Point exception (R4000)

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

TRUNC.W.fmt Floating-Point Truncate to Single Fixed-Point Format TRUNC.W.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	TRUNC.W 001101
6	5	5	5	5	6

Format:

TRUNC.W.fmt fd, fs

Description:

The contents of the FPU register specified by *fs* are interpreted in the specified source format *fmt* and arithmetically converted to the single fixed-point format. The result is placed in the FPU register specified by *fd*.

Regardless of the setting of the current rounding mode, the conversion is rounded as if the current rounding mode is round toward zero (RM = 1).

This instruction is valid only for conversion from a single- or double-precision floating-point formats. The operation is not defined if bit 0 of any register specification is set and the *FR* bit in the *Status* register equals zero, since the register numbers specify an even-odd pair of adjacent coprocessor general registers. When the *FR* bit in the *Status* register equals one, both even and odd register numbers are valid.

When the source operand is an Infinity or NaN, or the correctly rounded integer result is outside of -2^{31} to $2^{31}-1$, an Invalid operation exception is raised. If Invalid operation is not enabled, then no exception is taken and -2^{31} is returned.

TRUNC.W.fmt Floating-Point Truncate to Single Fixed-Point Format (continued) TRUNC.W.fmt

Operation:

```
T:     StoreFPR(fd, W, ConvertFmt(ValueFPR(fs, fmt), fmt, W))
```

Exceptions:

Coprocessor unusable exception
Floating-Point exception

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact exception
Overflow exception

FPU Instruction Opcode Bit Encoding

Opcode

31..29 \ 28..26	0	1	2	3	4	5	6	7
0								
1								
2		COP1						
3								
4								
5								
6		LWC1				LDC1		
7		SWC1				SDC1		

sub

25..24 \ 23..21	0	1	2	3	4	5	6	7
0	MF	DMFη	CF	γ	MT	DMTη	CT	γ
1	BC	γ	γ	γ	γ	γ	γ	γ
2	S	D	δ	δ	W	Lη	δ	δ
3	δ	δ	δ	δ	δ	δ	δ	δ

br

20..19 \ 18..16	0	1	2	3	4	5	6	7
0	BCF	BCT	BCFL	BCTL	γ	γ	γ	γ
1	γ	γ	γ	γ	γ	γ	γ	γ
2	γ	γ	γ	γ	γ	γ	γ	γ
3	γ	γ	γ	γ	γ	γ	γ	γ

Figure B-5 Bit Encoding for FPU Instructions

	function							
5..3	2..0 0	1	2	3	4	5	6	7
0	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1	ROUND.Lη	TRUNC.Lη	CEIL.Lη	FLOOR.Lη	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2	δ	δ	δ	δ	δ	δ	δ	δ
3	δ	δ	δ	δ	δ	δ	δ	δ
4	CVT.S	CVT.D	δ	δ	CVT.W	CVT.Lη	δ	δ
5	δ	δ	δ	δ	δ	δ	δ	δ
6	C.F	C.UN	C.EQ	C.UEQ	C.OLT	C.ULT	C.OLE	C.ULE
7	C.SF	C.NGLE	C.SEQ	C.NGL	C.LT	C.NGE	C.LE	C.NGT

Figure B-6 Bit Encoding for FPU Instructions (cont.)

Key:

γ	Operation codes marked with a gamma cause a reserved instruction exception. They are reserved for future versions of the architecture.
δ	Operation codes marked with a delta cause unimplemented operation exceptions in all current implementations and are reserved for future versions of the architecture.
η	Valid for 64-bit mode only.

SECDED Codes

C

Single Error Correcting/Double Error Detecting Codes

The ECC codes chosen for the processor's secondary cache data and secondary cache tag are single error correcting double error detecting codes that also detect three or four bit errors within a nibble. These codes were developed from codes proposed by M. Y. Hsiao in his paper, "A Class of Optimal Minimum Odd-weight-column SECDED Codes". The 64-bit data code is a modification of one of the 64-bit codes proposed by Hsiao to include the ability to detect three- and four-bit errors within a nibble. The 25-bit tag code was created using the patterns observed in the 64-bit data code.

The data code has the following properties:

1. It is a single error correcting, double error and three or four bit error within a nibble detecting code.

2. It provides 64 data bits protected by 8 check bits yielding 8 bit syndromes.

3. It is minimal in that each parity tree used to generate the syndrome has only 27 inputs, the minimum possible number.

4. It provides byte XOrs of the data bits as part of the XOr trees used to build the parity generators. This allows picking byte parity out of the XOr trees that generate or check the code.

5. Single bit errors are indicated by syndromes that contain exactly 3 ones or by syndromes that contain exactly 5 ones in which bits 0-3 or bits 4-7 of the syndrome are all one. This makes it possible to decode the syndrome to find which data bit is in error with 4-input NAND gates, provided a pre-decode AND of bits 0-3 and bits 4-7 of the syndrome is available.

For the check bits, a full 8-bit decode of the syndrome is required.

6. Double bit errors are indicated by syndromes that contain an even number of ones.

7. Three bit errors within a nibble are indicated by syndromes that contain 5 ones in which bits 0-3 of the syndrome and bits 4-7 of the syndrome are not all one.

8. Four bit errors within a nibble are indicated by syndromes that contain 4 ones. Because this is an even number of ones, four bit errors within a nibble look like double bit errors.

The tag code has the following properties:

1. It is a single error correcting, double error and three or four bit error within a nibble detecting code.

2. It provides 25 data bits protected by 7 check bits yielding 7-bit syndromes.

3. It provides byte XOrs of the data bits as part of the XOr trees used to build the parity generators. This allows picking byte parity out of the XOr trees that generate or check the code.

4. Single bit errors are indicated by syndromes that contain exactly 3 ones. This makes it possible to decode the syndrome to find which data bit is in error with 3 input NAND gates. For the check bits a full 7 bit decode of the syndrome is required.

5. Double bit errors are indicated by syndromes that contain an even number of ones.

6. Three bit errors within a nibble are indicated by syndromes that contain 5 ones or 7 ones.

7. Four bit errors within a nibble are indicated by syndromes that contain 4 ones or 6 ones. Because these are even numbers of ones, four bit errors within a nibble look like double bit errors.

The parity check matrices for the data ECC code and the tag ECC code specifying the distribution of data and check bits across nibbles are shown in Figure C-7 and Figure C-8. The Check Bits in Figure C-7 correspond to **SysADC(7:0)**, **SCDChk(15:8)**, or **SCDChk(7:0)**. The Data Bits in Figure C-7 correspond to **SysAD(63:0)**, **SCData(127:64)**, or **SCData(63:0)**. The Check Bits in Figure C-8 correspond to **SCTChk(6:0)** and the Data Bits in Figure C-8 correspond to **SCTag(24:0)**.

The Parity Check Matrices, in Figure C-7 and Figure C-8, are used to generate the ECC code for a fixed-width data word. The Parity Check Matrices can also be used to find the data bit that is in error. In Figure C-7, the data word is 64 bits and in Figure C-8, the data word is 25 bits.

ECC Check Code Generation

An 8-bit ECC check code is generated in the following manner. The state, either 1 or 0, of each bit of the ECC check code is determined by generating even parity for a selected group of data bits. The state of an even parity bit is a "1" if there is an odd number of "1's" in the data word and a "0" if the data word is all "0's" or if there is an even number of "1's" in the data word.

For each bit of the ECC check code (1 ECC code bit per row), the selected group of data bits consists of all of the data and check bits in which there is a "1" in the data bit or check bit column for that row. If check bits are used to generate the ECC check code bit, assume that it has a "0" state. The "." represents a "0" and means that this particular data or check bit is not used to generate this ECC code bit. For example, if Data(63:0) = 0x 0000 0000 0000 0000 then the ECC check code is 0000 0000; if Data(63:0) = 0x 0000 0000 0000 0001 then the ECC check code is 0001 0011.

Determining Single Data Bit Errors

The following procedure is used to determine which single data bit is in error. Assume a system transmitted a 64-bit doubleword and 8 bits of ECC. To verify proper transmission of the 64-bit doubleword and 8-bit ECC check code, the receiving system generates an 8-bit ECC check code from the received 64-bit doubleword. The receiving system then exclusive ORs the received check bits with the newly generated ECC check bits. The results of this exclusive OR is called the syndrome. If the syndrome is 0000 0000, it indicates that the received word and newly generated ECC check bits are the same as the transmitted word and check bits. If the syndrome corresponds to any of the syndromes in the Figure C-7, the data bit or check bit that corresponds to this syndrome is the bit in error. A quick way to determine if there is a match is to look at the number on ones in the syndrome. If the syndrome contains either three or five ones, the syndrome is in Figure C-7. If the syndrome contains a single one, the erroneous bit is an ECC check bit. If the syndrome is contained in the Figure C-7, the bit that is in error can be corrected by inverting the state of that bit.

The following examples show in what instances Parity Check Matrices are used:

Single Data bit error.

Single Check bit error.

Multiple Data bit errors (2 consecutive bits in a nibble)

Multiple Data bit errors (3 consecutive bits in a nibble)

Multiple Data bit errors (4 consecutive bits in a nibble)

Single Data Bit ECC Error

A single data bit ECC error can be detected and corrected as follows. Assume, the data doubleword Data(63:0) = 0x 0000 0000 0000 0000 with ECC check code 0000 0000 is transmitted and data doubleword Data(63:0) = 0x 0000 0000 0000 0001 with ECC check code 0000 0000 is received. The receiving system will regenerate the ECC for the received data. The ECC check code for Data(63:0) = 0x 0000 0000 0000 0001 is 0001 0011. The syndrome is generated by the exclusive OR of the received check bits, 0000 0000, and the regenerated check bits, 0001 0011. The resulting syndrome is 0001 0011. Since the syndrome has three 1s, it is contained in the parity check matrix. Searching the matrix (Figure C-7) shows that the syndrome, 0001 0011, corresponds to data bit 0. This indicates that the state of the received data bit 0 is incorrect. To correct the error, the system will invert the state of the received data bit 0.

Single Check Bit ECC Error

A single check bit ECC error can be detected and corrected as follows. Assume the data doubleword Data(63:0) = 0x 0000 0000 0000 0000 with ECC check code 0000 0000 is transmitted and data doubleword Data(63:0) = 0x 0000 0000 0000 0000 with ECC check code 0000 0001 is received. The receiving system regenerates the ECC for the received data. The ECC check code for Data(63:0) = 0x 0000 0000 0000 0000 is 0000 0000. The syndrome is generated by the exclusive OR of the received check bits, 0000 0001, and the regenerated check bits, 0000 0000. The resulting syndrome is 0000 0001. Since the syndrome has one 1, it is contained in the parity check matrix. Searching this matrix (Figure C-7) shows that the syndrome, 0000 0001, corresponds to check bit 0. This indicates that the state of the received check bit 0 is incorrect. To correct the error, the system invert sthe state of the received check bit 0.

Multiple Data Bit ECC Errors

A multiple (2 bits within a nibble) data bit ECC error can be detected as follows. Assume the data doubleword Data(63:0) = 0x 0000 0000 0000 0000 with ECC check code 0000 0000 is transmitted and data doubleword Data(63:0) = 0x 0000 0000 0000 0011 with ECC check code 0000 0000 is received. The receiving system regenerates the ECC for the received data. The ECC check code for Data(63:0) = 0x 0000 0000 0000 0011 is 0011 0000. The syndrome is generated by the exclusive OR of the received check bits, 0000 0000, and the regenerated check bits, 0011 0000. The resulting syndrome is 0011 0000. Since the syndrome has two 1s or an even number of 1s, it indicates that the a double bit error has been detected. The double bit error cannot be corrected.

A multiple (3 bits within a nibble) data bit ECC error can be detected as follows. Assume the data doubleword Data(63:0) = 0x 0000 0000 0000 0000 with ECC check code 0000 0000 is transmitted and data doubleword Data(63:0) = 0x 0000 0000 0000 0111 with ECC check code 0000 0000 is received. The receiving system regenerates the ECC for the received data. The ECC check code for Data(63:0) = 0x 0000 0000 0000 0111 is 0111 0011. The syndrome is generated by the exclusive OR of the received check bits, 0000 0000, and the regenerated check bits, 0111 0011. The resulting syndrome is 0111 0011. Since the resulting syndrome has five 1s and no four of the 1s are contained in check bits (7:4) or check bits (3:0), the user knows that 3 errors occurred within a nibble. The triple bit error within a nibble cannot be corrected.

A multiple (4 bits within a nibble) data bit ECC error can be detected as follows. Assume the data doubleword Data(63:0) = 0x 0000 0000 0000 0000 with ECC check code 0000 0000 is transmitted and data doubleword Data(63:0) = 0x 0000 0000 0000 1111 with ECC check code 0000 0000 is received. The receiving system regenerates the ECC for the received data. The ECC check code for Data(63:0) = 0x 0000 0000 0000 1111 is 1111 0000, The syndrome is generated by the exclusive OR of the received check bits, 0000 0000, and the regenerated check bits, 1111 0000. The resulting syndrome is 1111 0000. Since the resulting syndrome has four 1s or an even number of 1s, this error looks like a double bit error. The 4 bit errors within a nibble cannot be corrected.

25-Bit Parity Check Matrix

This same procedure works for the 25-bit parity check matrix shown in Figure C-8. The only difference is in the number of check bits used and decode of errors .

Figure C-7 Parity Check Matrix for the Data ECC Code

Check Bit			43		52											70		61	
Data Bit		6666 3210	55 98	5555 7654	55 32	5544 1098	4444 7654	4444 3210	3333 9876	3333 5432	3322 1098	2222 7654	2222 3210	1111 9876	1111 5432	11 10	9876	54	3210
MSB	27	1111	11..	11..	1...	1...	....	1111	1111	....	1...	1...	1...	1...	....	1.1.	.1..	1...	1...
	27	1111	1...	1...	1...	.1..	....	....	....	1111	.1..	.1..	.1..	.1..	1111	11..	11..	1.1.	.1..
ECC	27	....	1...	11..	1.1.	..1.	1111	1111	....	....	..1.	..1.	..1.	..1.	1111	1...	1...	11..	..1.
Code	27	....	1.1.	.1..	11..	...1	1111	....	1111	1111	...1	...1	...1	...1	....	1...	11..	1...	...1
	27	1...	.1.1	..11	.1..	....	1...	1...	1...	1...	1111	1111	....	1111	1...	11..	...1	.1..	....
Bits	27	.1..	11..	..1.	.1.1	1111	.1..	.1..	.1..	.1..	....	....	1111	1111	.1..	.1..	..11	.1..	....
	27	..1.	.1..	..11	11..	1111	..1.	..1.	..1.	..1.	1111	....	....	....	..1.	.1..	..1.	.1.1	1111
LSB	27	...1	.1..	...1	.1..	....	...1	...1	...1	...1	....	1111	1111	....	...1	.1.1	..11	11..	1111
Number of ones in syndrome*		3333	5511	3333	5511	3333	3333	3333	3333	3333	3333	3333	3333	3333	3333	5511	3333	5511	3333

* This row indicates the number of ones in the generated syndrome, for each data bit in error.

Check Bit		0	12	34	56				
Data Bit		222 432	22 10	11 98	11 76	1111 5432	11 1098	7654	3210
MSB	11	.1..	1...	1...	...1	1111	1...	1...	1...
	13	1...	.1..	.1..	..1.	1111	1111	....	.1..
ECC	10	..1.	1...	...1	1...	....	1111	.1..	..1.
Code	10	.1..	.1..	..1.	.1..	1...	.1..	1111	....
Bits	13	1...	...1	1...	1...	.1..	....	1111	1111
	11	..1.	..1.	.1..	.1..	..1.	..1.	..1.	1111
LSB	14	1111	11..	11..	11..	...1	...1	...1	...1
Number of ones in syndrome*		3331	3311	3311	3311	3333	3333	3333	3333

Figure C-8 Parity Check Matrix for the Tag ECC Code

* This row indicates the number of ones in the generated syndrome, for each data bit in error.

Sub-block Ordering

D

Sub-block ordering is an order for transmitting the data elements that form the block of data when the data element transmitted first is not the data element at the beginning of the block. Sub-block ordering causes the data elements of the block to be transmitted in an order that fills out sub-blocks of increasing size. For the R4000, the smallest data elements of a block transfer is a double word; therefore, the double word at the target address is transferred first, followed by the double word that fills out the quad word that contains the starting double word. Next, the quad word that fills out an octal word containing the starting quad word is transferred in the same order as the first quad word, followed by the octal word that fills out a hex word containing the starting octal word in the same order as the first octal word, and so on through sub-blocks of increasing size until the entire block has been transferred.

Perhaps an easier way to consider sub-block ordering is to look at a method for generating the addresses, within the block, of the double words to be transferred for sub-block ordering. A simple method for generating such addresses is to bit-wise XOR the starting double word address with the output of a binary counter that is counting the double words in the block starting at double word zero.

Table D-1 through Table D-3 illustrate the sequence of double words transferred using sub-block ordering for a thirty-two word block based on three different starting block addresses. For these illustrations, the double words in the block will be identified by their block addresses. The block address for each double word in a block is derived by numbering the double words in the block sequentially starting with zero.

The tables also include a binary count of the double words in the block to illustrate the XOr relationship between this count, the starting address, and the block addresses of the double words transferred.

Table D-1 Sequence of Double Words Transferred Using Sub-block Ordering- 0010

Cycle	Starting Block Address	Binary Count	Double Word Transferred
1	0010	0000	0010
2	0010	0001	0011
3	0010	0010	0000
4	0010	0011	0001
5	0010	0100	0110
6	0010	0101	0111
7	0010	0110	0100
8	0010	0111	0101
9	0010	1000	1010
10	0010	1001	1011
11	0010	1010	1000
12	0010	1011	1001
13	0010	1100	1110
14	0010	1101	1111
15	0010	1110	1100
16	0010	1111	1101

Table D-2 Sequence of Double Words Transferred Using Sub-block Ordering- 1011

Cycle	Starting Block Address	Binary Count	Double Word Transferred
1	1011	0000	1011
2	1011	0001	1010
3	1011	0010	1001
4	1011	0011	1000
5	1011	0100	1111
6	1011	0101	1110
7	1011	0110	1101
8	1011	0111	1100
9	1011	1000	0011
10	1011	1001	0010
11	1011	1010	0001
12	1011	1011	0000
13	1011	1100	0111
14	1011	1101	0110
15	1011	1110	0101
16	1011	1111	0100

Table D-3 Sequence of Double Words Transferred Using Sub-block Ordering- 0101

Cycle	Starting Block Address	Binary Count	Double Word Transferred
1	0101	0000	0101
2	0101	0001	0100
3	0101	0010	0111
4	0101	0011	0110
5	0101	0100	0001
6	0101	0101	0000
7	0101	0110	0011
8	0101	0111	0010
9	0101	1000	1101
10	0101	1001	1100
11	0101	1010	1111
12	0101	1011	1110
13	0101	1100	1001
14	0101	1101	1000
15	0101	1110	1011
16	0101	1111	1010

Output Buffer Δi/Δt Control Mechanism

E

The speed of the R4000 output drivers is controlled by a negative feedback loop that insures drive-off times are only as fast as necessary to meet the system requirement of single cycle transfers. This guarantees the minimum ground bounce due to the L*Δi/Δt of the switching buffers, consistent with the system timing requirements. Four bits are used to control each of the pull-up and pull-down delays. These bits are initially set to the values in the mode bits **InitN<3..0>** for pull-up and **InitP<3..0>** for pull-down.

Under normal conditions, the Δi/Δt control mechanism is expected to be constantly enabled so that it can compensate the output buffer delay for any changes in the temperature or power supply voltage. The **EnblDPLL** mode bit should be set for this mode of operation.

For situations where the jitter associated with the operation of the Δi/Δt control mechanism cannot be tolerated and where the variation in temperature and supply voltage after **ColdReset** is expected to be small, the Δi/Δt control mechanism can be instructed to lock only during **ColdReset** and thereafter retain its control values. The **EnblDPLLR** mode bit should be set and the **EnblDPLL** mode bit should be cleared for this mode of operation.

In addition, if both the **EnblDPLL** and **EnblDPLLR** mode bits are cleared, the speed of the output buffers can be set with the **InitP<3..0>** and **InitN<3..0>** mode bits.

The drive off delays can be set through the mode bits. Currently, delays of 0.5T, 0.75T, and T are supported, corresponding to the **Drv0_50**, **Drv0_75**, and **Drv1_ 00** mode bits, where T is the **MasterClock** period. For example, in **Drv0_75** mode, the entire signal transmission path including the clock-to-Q, output buffer drive time, board flight time, input buffer delay, and setup time will be traversed in 0.75 * the MasterClock period, plus or minus the jitter due to the Δi/Δt control mechanism.

All output drivers on the R4000, with the exception of the clock drivers, are controlled by the Δi/Δt control mechanism. The delay due to the output buffer drive time component of the **SCAddr<17..0>**, **SCOEB**, **SCWRB**, **SCDCSB**, and **SCTCSB** pins is approximately 66% of the delay of drivers of the other pins.

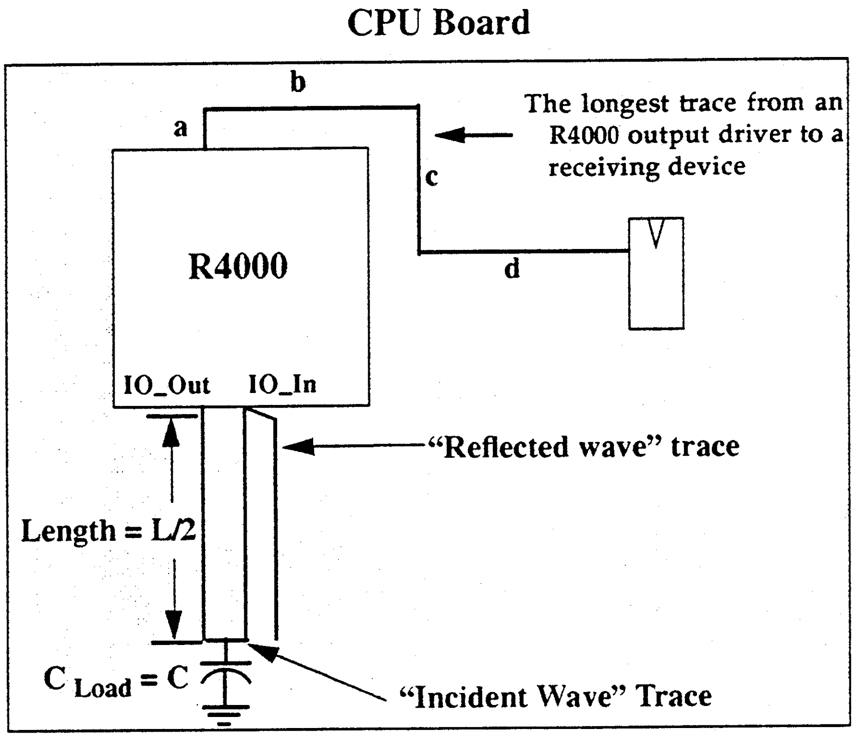
The R4000 determines the worst case propagation delay from an R4000 output driver to a receiving device by measuring the transmission line delay of the trace that connects the R4000 IO_Out and IO_In pins. This representative trace must have one and a half times the length and approximately the same capacitive loading as the worst case trace on any R4000 output.

The designer determines the trace characteristics by:

- measuring the longest path from an R4000 output driver to a receiving device: **L**
- calculating the maximum capacitive loading on any signal pin: **C**
- connecting an incident-wave trace of length L with a capacitive loading of C between the **IO_In** and **IO_Out** pins of the R4000
- connecting a reflected wave trace of length L/2 to the **IO_In** pin of the R4000.

An R4000 with appropriate traces connected to the **IO_In** and **IO_Out** pins is illustrated in Figure E-9.

Figure E-9 IO_In/IO_Out Board Trace



L = a + b + c + d

C = Total Capacitance Loading of the worst case trace

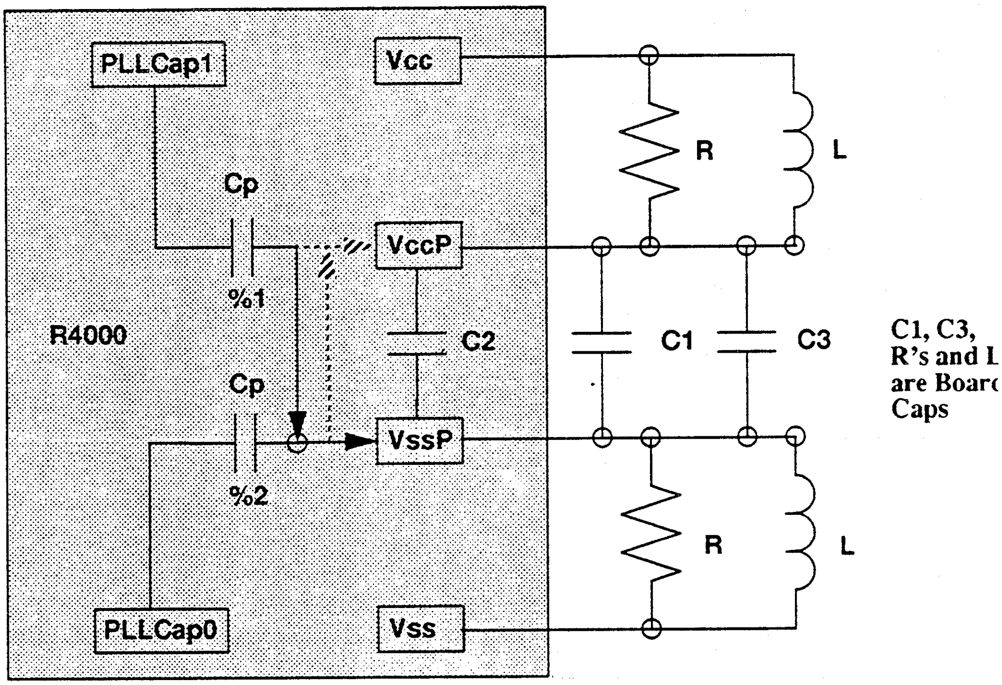
PLL Passive Components

F

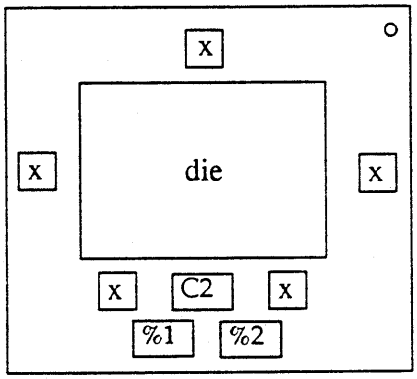
The Phase Locked Loop circuit requires several passive components for proper operation. These passive components are connected to **PLLCap0**, **PLLCap1**, **VccP**, and **VssP**, as illustrated in Figure F-10. In addition, the capacitors for **PLLCap0** (Cp) and **PLLCap1** (Cp) can be connected to either **VssP** (as shown), **VccP**, or one to **VssP** and one to

VccP. Note that C2 and both Cp capacitors are incorporated into both the 179PGA and 447PGA package designs as surface-mounted chip capacitors.

Figure F-10 PLL Passive Components

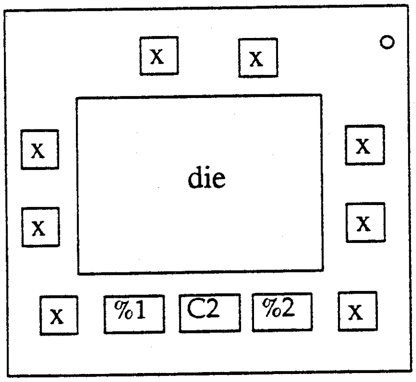


A top view of the 179-pin package with caps looks like this:



x: Vss-Vcc Bypass Caps
C2: VssP-VccP Bypass Caps
%1, %2: PLL Caps

A top view of the 447-pin package with chip capacitors looks like this:



x: Vss-Vcc Bypass Caps
C2: VssP-VccP Bypass Caps
%1, %2: PLL Caps

It is essential to isolate the analog power and ground for the PLL circuit (**VccP/VssP**) from the regular power and ground (**Vcc/Vss**). Initial evaluations have yielded good results with the following values:

R=5 ohms C1=1nF C2=82nF

C3=10uF Cp=470pF

Since the optimum values for the filter components depend upon the application and the system noise environment, these values should be considered as starting points for further experimentation within the application specific context. In addition, the chokes (inductors: L) can be considered for use as an alternative to the resistors (R) for power supply filtering.

R4000 Coprocessor 0 Hazards

G

R4000 Coprocessor 0 hazards are listed in Table G-4. The following notes apply:

α **Status.EXL** and **Status.ERL** are permanently cleared in stage 8, but the effect of clearing them is visible to instruction fetch starting in stage 4.

β Only one instruction needs to separate Index Load Tag and MFC0 Tag, even though the above would imply three instructions.

- The instruction following a MTC0 must not be a MFC0.
- The five instructions following a MTC0 to Status that changes KSU and sets EXL or ERL may be executed in the new mode, and not kernel mode. This can be avoided by setting EXL first, leaving KSU set to kernel, and then later changing KSU.
- There must be two non-load and non-CACHE instructions between a store and a CACHE instruction directed to the same primary cache line as the store.

Table G-4 R4000 Coprocessor 0 Hazards

Operation	Source		Destination	
	Name	Stage	Name	Stage
MTC0	gpr rt	3	cpr rd	7
MFC0	cpr rd	4	gpr rt	7
TLBR	Index, TLB	5-7	PageMask, EntryHi, EntryLo0, EntryLo1	8
TLBWI TLBWR	Index or Random	5-7	TLB	8
	PageMask, EntryHi,			
	EntryLo0, EntryLo1			
TLBP	PageMask, EntryHi	3-6	Index	7
ERET	EPC or ErrorEPC, Status, TLB	4	Status.EXL, Status.ERL	4-8α
			LLbit	7
CACHE Index Load Tag			TagLo, TagHi, ECC	8β
CACHE Index Store Tag	TagLo, TagHi, ECC	7		
CACHE Hit ops			Status.CH	8
Instruction fetch	EntryHi.ASID	0		
	Status.KSU, Status.EXL,			
	Status.ERL, Status.RE,			
	Config.K0C, Config.IB			
	Config.SB	3		
	TLB	2		
Instruction fetch exception			EPC, Status	8
			Cause, BadVAddr, Context	3
Coprocessor usable test	Status.CU, Status.KSU Status.EXL, Status.ERL	2		
Interrupt	Cause.IP, Status.IM Status.IE, Status.EXL Status.ERL	3		

Operation	Source		Destination	
	Name	Stage	Name	Stage
Load/Store	EntryHi.ASID Status.KSU, Status.EXL, Status.ERL, Status.RE, Config.K0C, Config.DB TLB	4		
	Config.SB	7		
	WatchHi, WatchLo	4-5		
Load/Store exception			EPC, Status, Cause	8
			BadVAddr, Context	
TLB shutdown			Status.TS	7

Additional errata which affect uniprocessor designs and may affect multiprocessor configurations are listed in the MIPS R4000PC, R4000SC Errata, Silicon Revision 2.2 and 3.0.

1. A store instruction whose tags match a primary cache line, completes into the primary cache at the beginning of the certain stalls. Then, during the stall, an intervention comes in that invalidates or changes the tags of the store to shared. When the R4000 runs again, the store looks at its tags and signals a cache miss although its store has actually completed into primary cache. During the cache miss of the store, incorrect data is sent out under the update protocol.

Workaround: There is no workaround for this problem when using the update protocol. Invalidate protocol should be used in place of updates.

2. Replace this errata with the text in errata 5.

3. External requests may cause an incorrect state change in the secondary cache when the R4000 is in a data or instruction cache miss on a non-coherent page.

Workaround: Do not use the cacheable, noncoherent TLB page attribute in a multiprocessing system.

4. When the CU bit in the Config register is set to one, the R4000 should issue updates on the Store Conditional (sc) and Store Conditional Doubleword (scd) instructions instead of the protocol specified by the TLB page attribute. The R4000 ignores this bit and issues invalidates.

Workaround: Update protocol should not be used for the store conditional instructions. Previous errata prevent the use of the update protocol for TLB pages.

5. Store-load interlock breaks strong ordering. This situation can occur under the following conditions:

    1. A store-load interlock is created when a load immediately following a store tries to access the same cache line. This load can be either to the same address or to an address which aliases to the same cache line location.
    2. The store misses in the primary.
    3. The stall sequence for the processor contains two stages.
    4. First, the secondary is accessed to see if the line is present. After getting the data, which may not be the correct data because the tags do not match, the processor executes a restart sequence which is when the secondary tags are compared.
    5. Second, there is a secondary cache miss after the restart, a stall for a cache state change or the processor must issue an invalidate or update request.

The problem arises if an intervention request is signaled to the processor between the two stages of the stall caused by the primary cache miss. To handle the store/load interlock correctly, the primary cache is written during the first stage of the stall and during the second stage of the stall.

This may create two problems. First, an intervention request that comes in after the first stage of the stall but before the second stage, will be satisfied before the store completes in the second stage of the stall. The intervention response will contain the new data. Later, the processor issues an update or invalidate which violates strong ordering.

Second, if the store was a store conditional, the intervention will receive the new data although the store conditional may never complete. In this case the intervening processor was returned incorrect data.

Workaround: Whenever a store to a sharable line is followed immediately by a load to a sharable line and the store and load have the same primary cache address, i.e. the 12 LSBs of the store address and load address match, insert a noop (or a non-memory instruction) between the store and load. A less fine-grained fix would be to just detect the store to sharable line followed by a load to sharable line without considering the address match and insert the noop or non-memory instruction.

New code sequence for general solution:

```
STORE instruction<SHARABLE line>
NOOP
LOAD  instruction<SHARABLE line>
```

6. With the system interface configured with ECC checking, the response to a Snoop request contains primary cache parity on the SysADC bus rather than ECC.

Workaround: Since a snoop request does not return data, the information present on the SysAD and SysADC busses should be ignored even if incorrect data is present.

7. If a store hit in the primary cache causes a state change stall and an external request invalidates the target of the store, the store will miss in and generate a read request for the target line. In this case, the check pins, SysADC, will not be driven during the read request.

Workaround: During a read request, ignore the SysADC pins.

8. Do not split the command and data cycles of an update cycle. Under certain conditions, the R4000 will prematurely take ownership of the system interface bus before the data cycle is issued to the R4000 if there are any cycles between the update command and data cycles.

Workaround: Do not split the command and data cycles of an update cycle.

9. Under the conditions listed below, the EB bit in the CacheErr register is incorrectly set.

1) A store targets a shared line in the primary cache
2) The tag in this line has a parity error
3) Under this condition, the processor will stall due to a data cache miss and the CacheErr register is set.
4) As the processor comes out of the data cache miss and before it vectors to the CacheErr exception vector, there is an instruction cache miss and a pending external request which targets the same line with the parity error.

Under these conditions, the EB bit will get set although there was no parity error. The EB bit implies that both a data and instruction parity error have occurred. In this case, there was only a data parity error.

Workaround: The EB bit is meaningful only if the ER bit indicates instruction error.

10. R4000PC, R4000SC: If a secondary cache line that is being replaced matches the address currently stored for a load linked, the cache line retained bit may not always be set in the read command issued for a cache refill..

Workaround: Software must guarantee that the load link address is not replaced by the instruction block that contains the load link instruction.

Note: Change bars in the left column indicate corrections or changes from the last revision of the errata.

1. R4000PC, R4000SC: Master/checker mode is not available.

Workaround: The mode bits to enable Master/Checker operation should not be set to one. These mode bits are the MCMode and DataMaster modebits. Behavior of the R4000 is undefined when Master/Checker mode is enabled.

2. R4000PC, R4000SC: Status output pins do not function.

3. R4000PC, R4000SC: Reduced power mode is not available in the current revision of the R4000. Setting the RP bit in the Status register has no effect on the operation of the R4000.

4. R4000PC, R4000SC (Note: Processor revision 3.0 does not contain this errata): An instruction sequence which contains a load which causes a data cache miss and a jump, where the jump instruction is that last instruction in the page and the delay slot of the jump is not currently mapped, causes the exception vector to be overwritten by the jump address. The R4000 will use the jump address as the exception vector.

Example:

```
lw              <----   data cache miss
noop            <----   one or two Noops
jr              <----   last instruction in the page (jump or branch instruction)
--------------  <----   page boundary
noop
```

Workaround: Jump and branch instructions should never be in the last location of a page.

5. R4000SC: When the primary instruction cache is configured with an 8-word line size, the virtual coherency exception (VCE) does not function correctly.

Workaround: The primary instruction cache should only be configured with a 4-word line size.

6. R4000PC, R4000SC: The following conditions cause the R4000 to operate incorrectly:

1. An exception is taken from user code
2. On the eret instruction of the exception handler, a CacheErr exception is taken.

The R4000 takes the CacheErr exception correctly but returns to user code instead of the ERET in the exception handler. This will then cause an Address Error exception.

Workaround: Use the following code sequence as the last three instructions of the exception handler:

```
eret
noop
eret
```

The CacheErr handler must add 4 to the ErrorEPC to return to the noop.

7. R4000PC (Note: Processor revision 3.0 does not contain this errata): When an external request

is placed between the read and write of a data cache replacement of a dirty cache line and an uncached store is stalled in the WB pipeline stage, the R4000 will send out the command code for an uncached store during the block write of the writeback.

Workaround: Interrupts should be signaled through the Int0 through Int5 pins on the R4000PC package. An external null request should not be signaled between the write and read of the replacement of a dirty cache line. In the Revision 2.2 R4000, there are four cycles between the write and read of a dirty cache line replacement.

8. R4000PC, R4000SC: In supervisor mode, with the SX bit (64-bit mode enabled for supervisor mode) in the Status register set to one, the R4000 will generate an Address Error Exception in the "csseg" region: 0xFFFF FFFF C000 0000 to 0xFFFF FFFF DFFF FFFF.

9. R4000PC: If a writeback is delayed by an external request and the processor executes a cache operation which generates a writeback with the retained bit set on the system command bus, the retained bit will also be set for the first writeback.

Workaround: Since the retained bit is intended for multiprocessing systems, R4000PC systems should ignore this bit on the system command bus.

10. R4000PC, R4000SC: In kernel mode with the KX bit (64-bit mode enabled for kernel mode) in the Status register set to one, the R4000 fails to generate an Address Error Exception if a load or store is attempted in the region: 0xC000 0FFF F000 0000 to 0xFFFF FFFF 8000. Attempting access to this region should cause an Address Error exception.

11. R4000PC, R4000SC: In the case:

```
lw          rA, (rn)
noop                    (or any non-conflicting instruction)
lw          rn, (rA)    (where the address in rA causes a TLB refill)
-------------------->   end of page
page not mapped
```

where rn and RA are general purpose registers r0 through r31

This code sequence causes the second load instruction to slip due to a load use interlock. When the R4000 crosses the page boundary after the lw, it vectors to 0x8000 0000 and later causes an instruction cache miss. After the instruction cache miss is complete the LW causes another TLB refill. This should vector to 0x8000 0000 but instead goes to 0x8000 0180.

Workaround: In the general exception handler, the CAUSE register must be checked to see if a TLB miss is indicated.

12. R4000SC: The following conditions cause the virtual address in the BadVA register to be corrupted by a TLB Probe instruction (TLBP)

The problem occurs if:
1. An exception is generated
2. A TLBProbe is required to handle the exception
3. While handling the exception, a VCED or VCEI occurs, the BadVA may be incorrect.

Workaround: The exception handler should jump to uncached space to handle the TLBProbe.

13. R4000SC: The cacheop, Create_DEX_SC (secondary cache operation), when executed on a

non-coherent line, will change the state of the secondary line to Clean Exclusive instead of Dirty Exclusive.

Workaround: This situation should not create any problem in normal operation since a subsequent store will change the line to Dirty Exclusive.

14. This errata is an update to errata 4. The incorrect behavior does not occur with two Noops between the load and jump. There is a qualifying condition on the load and jump instructions and several similar cases are listed which cause the same error.

R4000PC, R4000SC: An instruction sequence which contains a load which causes a data cache miss and a jump, where the jump instruction is that last instruction in the page and the delay slot of the jump is not currently mapped, causes the exception vector to be overwritten by the jump address. The R4000 will use the jump address as the exception vector. In the first case, the target of the load instruction and source register for the jump instruction must be the same register. In all other cases, the condition is independent of the registers used.

Example:

```
        lw          rA,(rn)        <----data cache miss
        noop                       <---- one Noop
        jr          rA             <---- jump or branch instruction as the last instruction in the page
        ----------------------     <---- page boundary
        PAGE NOT MAPPED

        lw                         <----data cache miss
        div                        <---- signed, unsigned and doubleword integer divide
        beq                        <---- branch instruction as the last instruction in the page
        ----------------------     <---- page boundary
        PAGE NOT MAPPED

        sw                         <----data cache miss
        div                        <---- signed, unsigned and doubleword integer divide
        beq                        <---- branch instruction as the last instruction in the page
        ----------------------     <---- page boundary
        PAGE NOT MAPPED

        cacheop                    <----data cache miss
        div                        <---- signed, unsigned and doubleword integer divide
        beq                        <---- branch instruction as the last instruction in the page
        ----------------------     <---- page boundary
        PAGE NOT MAPPED
```

Workaround: Jump and branch instructions should never be in the last location of a page.

15. R4000PC, R4000SC: This errata was deleted. The problem does not occur on the R4000.

16. R4000PC, R4000SC: Please refer to errata 28 for an update to this errata description.

The following code sequence causes the R4000 to incorrectly execute the Double Shift Right Arithmetic 32 (dsra32) instruction. If the dsra32 instruction is executed during an integer multiply, the dsra32 will only shift by the amount in specified in the instruction rather than the amount plus 32 bits.

```
instruction 1:          mult rs,rt          integer multiply
```

instruction 2-12: dsra32 rd,rt,rs doubleword shift right arithmetic + 32

Workaround: A dsra32 instruction placed after an integer multiply should not be one of the 11 instructions after the multiply instruction.

17. R4000SC: This problem occurs when the system interface of the R4000SC is configured with ECC checking. During the writeback of a dirty cache line, the first double word from the secondary cache which follows a double word from the primary cache is driven with an incorrect ECC code on the SysADC bus. The SysADC bus retains the same code for the secondary cache double word as was driven for the primary cache double word.

Workaround: Use parity checking on the system interface bus (SysAD) rather than ECC. Alternatively, mask all checking errors by setting the DE bit of the Diagnostic Status Field in the Status register to one.

18. R4000SC: Under some conditions, stores which address the same primary cache line (lower 12 bits are the same), will prevent the R4000 from responding to an external request until all the secondary cache accesses are complete. The conditions under which this occur are the following:

1. Back to back stores which address the same cache line where the tags match.
2. Back to back stores which address a different address but map to the same cache line (lower 12 bits of the address are identical). In this case, the stores will result in the replacement of the line. If the accesses to the secondary cache map to the same location, the problem does not occur because the secondary cache interface is idle during the writeback. In this case, the external request will be accepted during the writeback.
3. Successive stores, not necessarily immediately following one another, which map to the same cache line under the conditions listed in 1 and 2 above but where the length of the loop is short enough that the secondary cache bus is never idle. The number of instructions in the loop where the problem can be observed is dependent upon the secondary cache timing parameters programmed by the boot time mode bits.

19. R4000PC, R4000SC: When there is a store followed by a load to the same address and the Watch register contains the address for the load, the Watch exception for the load is not taken.

Workaround: A "noop" placed between the store and the load will enble the Watch exception to be taken correctly.

22. R4000PC, R4000SC: When returning from 32-bit kernel mode to 64-bit user mode, the R4000 interprets the address of the first instruction as a 32-bit mode address. This may cause an incorrect mapping of the address depending upon the virtual address.

Workaround: When operating the R4000 in 64-bit user mode, only use 64-bit kernel mode. (KX=1 in the CP Status register)

23. R4000PC, R4000SC: The 64-bit instruction, daddi, fails to take an overflow exception.

Workaround: There is no workaround for this problem.

24. R4000PC, R4000SC: The R4000 does not take a Reserved Instruction Exception on the "rfe" instruction. The R4000 executes a "noop" and kills the following instruction.

Workaround: There is no workaround for this problem.

25. R4000PC, R4000SC: The "dmtc0" and "dmfc0" instructions do not cause a Reserved Instruc-

tion Exception in Supervisor mode. These instructions will complete successfully.

Workaround: There is no workaround for this problem.

26. R4000SC: Sequential ordering must be used with the R4000SC. The option to turn off sequential ordering does not function.

27. R4000PC, R4000SC: A TLB refill exception occurs on an instruction fetch with a value in the CP0 register BadVAddr that does not match CP0 register EPC.

The specific case found involved the sequence:

```
sw
nop
jal
cvt.s.w
mul.d
```

The store takes a data cache miss. In the restart sequence at the end of the data cache miss, the pipeline is backed up and the floating point scheduler causes a pipeline slip to occur. In this particular case, the slip that is generated as the data cache stall completes, causes the data address instead of the instruction address to be sent to the TLB. This results in an undefined address being given to the TLB for translation. This undefined address may cause a TLB refill exception to be taken if the undefined address is not present in the TLB.

Looking at the general case that could cause this problem to occur, the sequence of instructions required is:

1) an instruction causes a data cache stall (load, store or cacheop)
2) any instruction that will run without causing a slip or stall to occur
3) jump or taken branch that does not have its destination mapped in the tlb
4) pipeline activity that causes a slip in the restart sequnce of the data cache stall

Workaround:

The operating system (OS) will normally terminate a process that tries to access an address outside the expected range for the process. When the OS detects such an address in BadVAddr, it should check EPC to make sure it is within a valid range for the process. If EPC is not within the valid range, the OS should execute an "eret" instruction. The refill instruction will be re-taken and BadVAddr will contain the correct value..

If the OS is unable to determine the valid address range for the process, the value in EPC should be used to look for a load or store instruction. If EPC does not point to a load or store, the OS should execute an "eret". The "eret" will then cause another TLB refill exception, which will have a valid BadVAddr. If EPC points to a load or store, the OS must then interpret the instruction to generate the address for the data. If this address matches the address in BadVAddr, the process tried to access data outside the process address space. Otherwise the OS should execute an "eret" causing a TLB refill exception where the value in BadVAddr will be valid.

28. R4000PC, R4000SC: The text from errata 16 should be replaced by the following description.

All extended shifts (shift by n+32) and variable shifts (32 and 64-bit versions) may produce incorrect results under the following conditions:

1. An integer multiply is currently executing
2. These types of shift instructions are executed immediately following an integer divide instruction.

Workaround:

1. Make sure no integer multiply is running wihen these instruction are executed. If this cannot be predicted at compile time, then insert a "mfhi" to R0 instruction immediately after the integer multiply instruction. This will cause the integer multiply to complete before the shift is executed.

2. Separate integer divide and these two classes of shift instructions by another instruction or a noop.

29. R4000SC: Errata 26 is incorrect. The R4000 always uses sequential ordering regardless of the state of the mode bit which specifies subblock or sequential ordering.

30. R4000PC, R4000SC: When an nmi or softreset exception is taken, both exl and erl bits are set to one in the Status register. The R4000 should perserve the previous state of exl.

31. R4000SC: The SCAPar pins on the secondary cache bus are driven incorrectly. The pins are driven with even parity for the secondary cache address, SCWrB, SCDCSB and SCTCSB pins.

Workaround: These bits are not stored in the tag and are only used in systems which externally compare the address and parity bits.

32. R4000PC, R4000SC: Under the following conditions, the CP0 register, BadVAddr, can be corrupted.

1) There is a data cache miss which causes a jal (jump and link) to be stalled in the DS pipeline stage
2) A floating point dependency causes a slip during the restart after the data for the data cache miss is returned.

Workaround:

33. R4000PC, R4000SC: With the following code sequence and conditions, the R4000 with use the general exception vector, 0x80000180, instead of the the TLB Exception vector, 0x800000000.

```
lw              - takes a watch exception
lw              - takes a watch exception
j               - TLB exception
---------------------> Page boundary
```

Workaround: The general exception handler needs to handle refill exceptions directly from user code rather than only within the refill exception handler.

34. R4000PC, R4000SC: The R4000 incorrectly allows xkseg to access up to 0xc000 0100 0000

0000. This region should only extend to 0xc000 00ff 8000 0000.

Workaround:

35. R4000PC, R4000SC: Split secondary cache mode can cause invalid cache error exceptions.

Workaround: Use unified secondary cache mode.

36. R4000PC, R4000SC: The first instruction of the ECC handler cannot write to a general purpose register.

Workaround: The first instruction of the ECC exception handler should be a Noop.

37. R4000PC, R4000SC: The PIDx field of the CP0 register CacheErr, gets the wrong value on Data and Instruction parity errors. ECC errors are will put the correct values in this register, however.

Workaround: Under this failure condition, all possible values of PIDx in the primary cache must be checked for parity errors. In the R4000 with 8K primary caches, there are only two values for the primary cache index that must be checked.

38. R4000PC, R4000SC: A parity error on the primary cache dirty data bit, W, will not be detected.

Workaround: There is no workaround for this condition.

39. Under the following conditions, the TLB attributes for a page being refill into the microTLB may be associated with a page mapped in the microTLB.

1) The address of a jump instruction is to the last instruction in a page.
2) The next instruction after the jump target is not mapped in the microTLB.
3) The jump is stalled in the DS pipeline stage.

When the page targeted by the jump is refilled into the microTLB, the coherency bits associated with that page may be incorrect.

Workaround: This problem occurs if the TLB attributes are different for the two pages. Under these conditions, if the TLB attributes are the same, this problem will not occur.