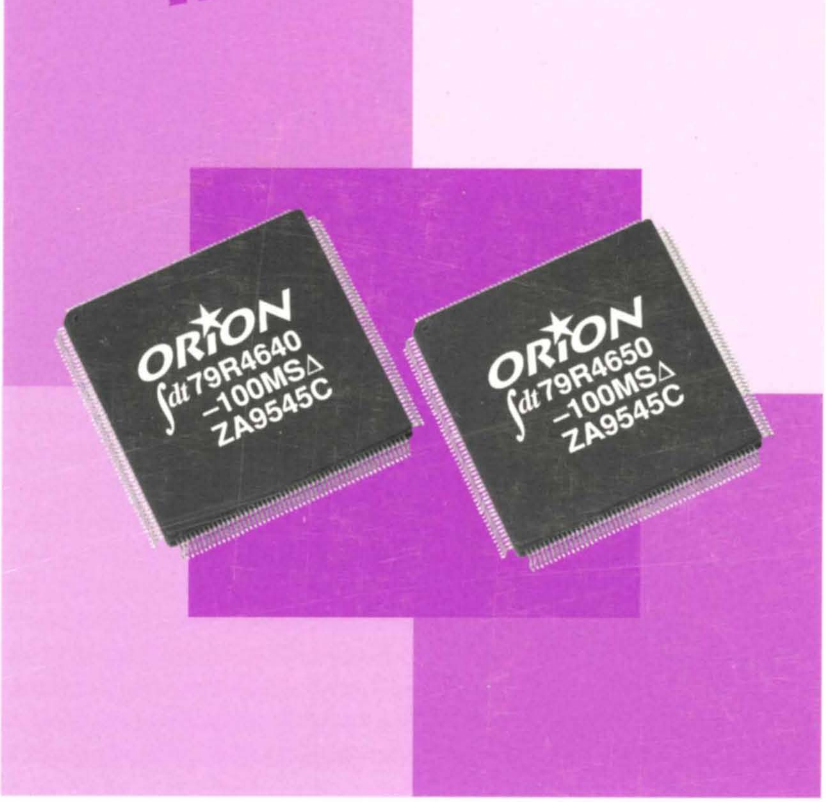
IDT79R4640 and IDT79R4650 RISC Processor



Hardware User's Manual

INTEGRATED DEVICE TECHNOLOGY, INC.

IDT79R4640™/IDT79R4650™ RISC Processor

Hardware User's Manual

Version 1.1

November 1995

Integrated Device Technology
2975 Stender Way
Santa Clara, CA 95054

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

LIFE SUPPORT POLICY

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

1. **Life support devices or systems are devices or systems that (a) are intended for surgical implant into the body, or (b) support or sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.**
2. **A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.**

About This Manual

This manual describes the operation of the IDT79R4640™/ IDT79R4650™, part of the Orion family of processors.

Note: Throughout this manual, references to the IDT79R4650 or R4650 also refer to the IDT79R4640 or R4640. The R4640 supports only the 32-bit bus width; otherwise, the R4640 and the R4650 are identical.

Summary of Contents

Chapter 1, "Overview," contains an overview of the R4650 microprocessor, including a detailed feature-by-feature comparison between the R4000 and the R4650.

Chapter 2, "CPU Instruction Set Overview," contains an overview of the central processing unit (CPU) instruction set. For a description of an individual CPU instruction refer to Appendix A, "CPU Instruction Set Details."

Chapter 3, "The CPU Pipeline," describes the basic operation of the CPU pipeline, including descriptions of the delay instructions (instructions that follow a branch or load instruction in the pipeline), interruptions to the pipeline flow caused by interlocks and exceptions, and R4650 implementation of an uncached store buffer.

Chapter 4, "Memory Management," describes the simple base-bounds mechanism used by R4650 for virtual-to-physical address translation.

Chapter 5, "CPU Exception Processing," describes the CPU exception processing, including a discussion of the format and use of each CPU exception register. Also included is a description of each exception's cause, together with the manner in which the CPU processes and services these exceptions.

Chapter 6, "The Floating-Point Unit," describes the R4650 floating-point unit (FPU) features, including the programming model, instruction set and formats, and the pipeline.

Chapter 7, "Floating-Point Exceptions," describes floating point unit (FPU) floating-point exceptions, including FPU exception types, exception trap processing, exception flags, saving and restoring state when handling an exception, and trap handlers for IEEE Standard 754 exceptions.

Chapter 8, "Processor Signal Descriptions," describes the signals used by and in conjunction with the R4650 processor. These signals include the System interface, the Clock/Control interface, the Interrupt interface, and the Initialization interface.

Chapter 9, "The Initialization Interface," describes the R4650 Initialization Interface, including the reset signal descriptions and types, initialization sequence, signals and timing dependencies, and boot modes, which are set at initialization time.

Chapter 10, "The Clock Interface," describes the clock signals (clocks) used in the R4650 processor, as well as information on basic system clocks and system timing parameters.

Chapter 11, "Cache Organization, Operation and Coherency," describes the on-chip cache memory, its place in the R4650 memory organization, and individual operations of the primary cache.

Chapter 12, "System Interface Overview," describes the system interface from both the processor and the external agent's point of view.

Chapter 13, "The Read Interface," discusses specifics of the read interface and read operations.

Chapter 14, "The Write Interface," discusses the Write protocol and associated operations.

Chapter 15, "The External Request Interface," discusses the External Request protocol and associated operations.

Chapter 16, "R4650 Processor Interrupts," describes the six hardware and single nonmaskable interrupts.

Chapter 17, "R4650 Error Checking," describes the Error Checking mechanism used in the R4650 processor.

Appendix A, "CPU Instruction Set Details," provides a detailed description on the operation of each R4650 instruction, listed alphabetically.

Appendix B, "FPU Instruction Set Details," provides a detailed description of each floating-point unit (FPU) instruction, listed alphabetically. Following each description is a discussion of exceptions that may result from executing the instruction.

Appendix C, "Cache Operations Timing," lists cycle operation counts and caveats for R4650 cache operations timing.

Appendix D, "Standby Mode Operation," describes the Standby Mode operation.

Appendix E, "Coprocessor 0 Hazards," identifies the R4650 Coprocessor 0 hazards.

Appendix F, "Integer Multiply Scheduling," describes the R4650 Integer Multiply Scheduling.

Where To Find More Product Information

Details about the R4640 or R4650 electrical interface can be found in the product's data sheet. Data sheets also include packaging and pin-out information.

For information about development tools, complementary support chips, and how to use this product in various applications, refer to IDT's online library of data sheets, applications notes, software reference manuals, and the IDT Advantage Program Guides.

Your local IDT sales representative can help you identify and use these resources.

TABLE OF CONTENTS

Integrated Device Technology, Inc.

LIST OF TABLES

LIST OF FIGURES

Introduction

The IDT79R4640™/IDT79R4650™ is a low-cost member of the IDT Orion family that is targeted to a variety of performance-hungry embedded applications. The R4650 continues the Orion tradition of high-performance through high-speed pipelines, high-bandwidth caches and bus interface, 64-bit architecture, and careful attention to efficient control. The R4650 reduces the cost of this performance—relative to the R4600—by removing functional units frequently not required for many embedded applications, such as double-precision floating point arithmetic and the Transition Lookaside Buffer (TLB).

Note: Throughout this manual, references to the IDT79R4650 or R4650 also refer to the IDT79R4640 or R4640. The R4640 is a device that only supports the 32-bit bus width; otherwise, the R4640 and the R4650 are identical.

The R4650 adds features relative to the R4600, reflective of its target applications. These features enable system cost reduction (e.g. optional 32-bit system interface) as well as higher performance for certain types of systems (such as cache locking, improved real-time support, and integer digital signal processing (DSP) capability).

The R4650 supports a wide variety of embedded processor-based applications, such as games systems, multi-media functions, internetworking/ data communications equipment, and office networking systems. Upwardly software-compatible with the R30xx RISController family and bus and upwardly software-compatible with the IDT Orion family, the R4650 will serve in many of the same applications. In addition, the R4650 will support applications that require DSP functions.

Performance

The R4650 brings Orion performance levels to lower cost systems. Orion performance is preserved by retaining large on-chip caches that are two-way set associative, a streamlined high-speed pipeline, high-bandwidth, 64-bit execution, and facilities such as early restart for data cache misses. These techniques combine to allow the system designer over 2GB/sec aggregate internal bandwidth, 533 MB/sec bus bandwidth, 175 Dhrystone MIPS, 44MFlops, and 66.7 M multiply-add/second (all at 133 MHz).

Upward Compatibility

The R4650 provides complete upward application-software compatibility with the IDT79R3000™ family of microprocessors, including the IDT RISController™79R3041™,79R3051™/79R3052™,79R3071™/79R3081™, 79R4600™, and the 79R4700™ families of microprocessors. An array of tools facilitates the rapid development of R4650-based systems, allowing a wide variety of customers to take advantage of the processor's high-performance capabilities while maintaining short time-to-market goals.

The 64-bit computing capability of the R4650 permits access to performance levels that were previously limited by the lower bandwidth and bit-manipulation rates inherent in 32-bit architectures.

For example, the R4650 can perform loads and stores from cached memory at the rates of 8-bytes every clock cycle, doubling the bandwidth of an equivalent 32-bit processor. This ability—coupled with the high clock rate for the R4650 pipeline—obtains new levels of performance from embedded systems.

A summary of features for the R4650 follows. For a detailed feature-by-feature comparison between the R4000 and the R4650, refer to Table 1.14.

Features

- High-performance embedded 64-bit microprocessor
  - 64-bit integer operations
  - 64-bit registers
  - 80MHz, 100MHz, 133MHz operation frequency
  - 5V and 3.3V versions
- High-performance DSP capability
  - 66.7 Million Integer Multiply-Accumulate Operations/sec @ 133 MHz
  - 44 MFlops floating point operations @133MHz
- High-performance microprocessor
  - 66.7 M Mul-Add/second at 133MHz
  - 44 MFLOP/s at 133MHz
  - >300,000 dhrystone (2.1)/sec capability at 133MHz (175 dhrystone MIPS)
- High level of integration
  - 64-bit, 175 MIPS integer CPU
  - 44MFlops Single precision floating-point unit
  - 8KB instruction cache; 8KB data cache
  - Integer DSP/multiply unit with 66.7M Mul-Add/sec
- Low-power operation
  - Less than 2W peak internal power at 100MHz
  - Active power management powers-down inactive units
  - Standby mode power consumption <200mW
- Upward software compatible with IDT RISController™ Family
- Large, efficient on-chip caches
  - Separate 8kB Instruction and 8kB Data caches
  - Over 1500MB/sec bandwidth from internal caches
  - 2-set associative
  - Write-back and write-through support
  - Cache locking to facilitate deterministic response
- Bus compatible with R4600/R4700 Orion family
  - System interfaces to 67 MHz, provides bandwidth up to 533 MB/S
  - Direct interface to 32-bit wide or 64-bit wide systems
  - Synchronized to external reference clock for multi-master operation
- Improved real-time support
  - Fast interrupt decode
  - Optional cache locking

Device Overview

The R4650 has a level of integration designed for high-performance and high-bandwidth computing. Key elements of the R4650 are illustrated below, with an overview of these features following. More detailed information will be presented in subsequent chapters.

Figure 1.1 presents a block level representation of the R4650's functional units.

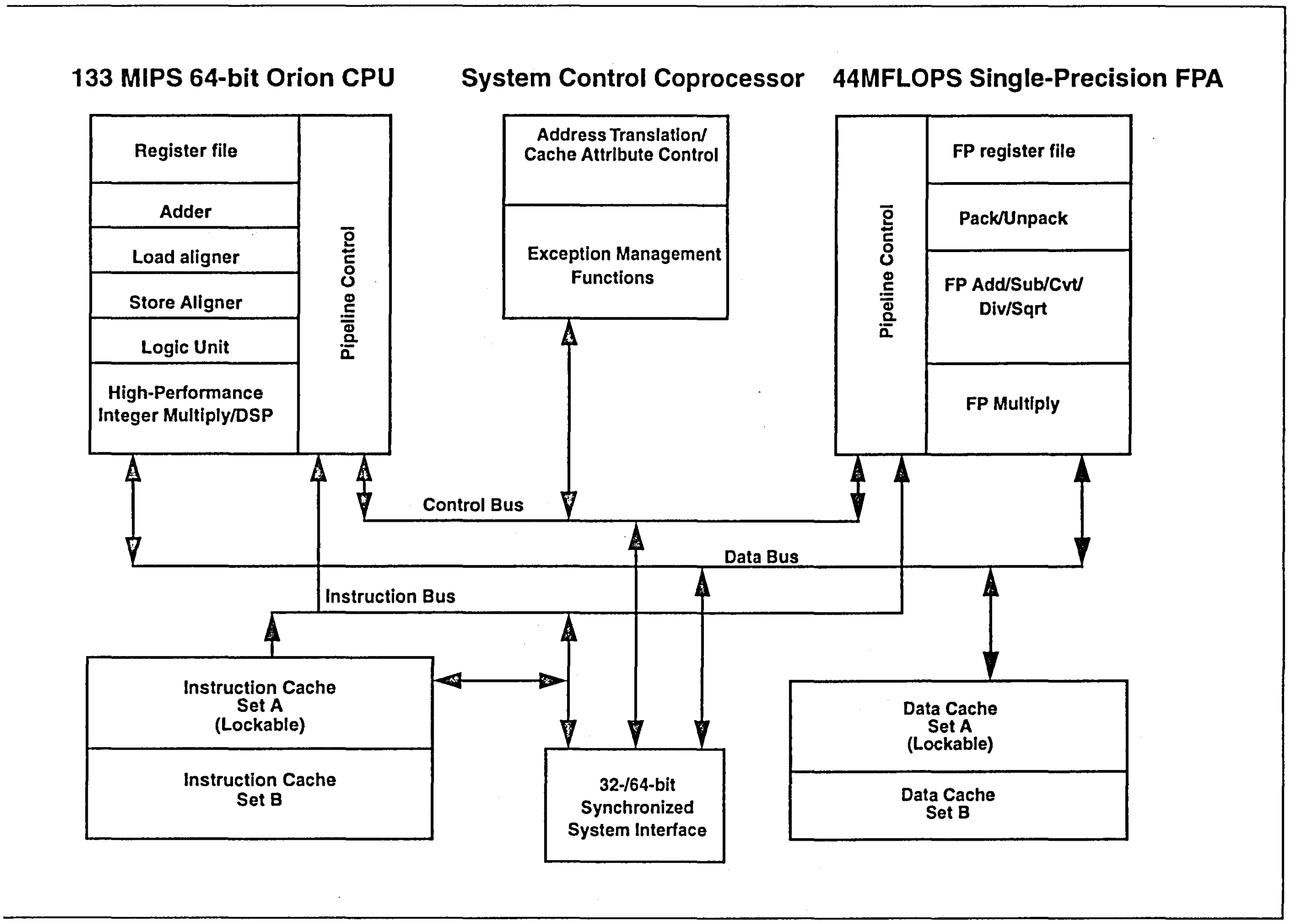

133 MIPS 64-bit Orion CPU
System Control Coprocessor
44MFLOPS Single-Precision FPA
Register file
Adder
Load aligner
Store Aligner
Logic Unit
High-Performance Integer Multiply/DSP
Pipeline Control
Address Translation/ Cache Attribute Control
Exception Management Functions
Pipeline Control
FP register file
Pack/Unpack
FP Add/Sub/Cvt/ Div/Sqrt
FP Multiply
Control Bus
Data Bus
Instruction Bus
Instruction Cache Set A (Lockable)
Instruction Cache Set B
32-/64-bit Synchronized System Interface
Data Cache Set A (Lockable)
Data Cache Set B

Figure 1.1 R4650 Block Diagram

Pipeline Overview

The R4650 implements a 5-stage pipeline similar to the IDT79R3000 and the IDT79R4600/R4700. The simplicity of this pipeline allows the R4650 to be a lower cost, lower powered processor than super-scalar or super-pipelined processors. Unlike superscalar processors, applications that have large data dependencies or require a great deal of load/stores can still achieve levels close to the peak performance of the processor.

Refer to Chapter 3 for a detailed discussion of the CPU pipeline operation, including descriptions of the instruction latencies, interruptions to the pipeline flow caused by interlocks and exceptions, and the R4650 implementation of a store buffer. For a detailed discussion of the FPU pipeline, refer to Chapter 6.

CPU Register Overview

The R4650 has thirty-two general-purpose 64-bit registers. These registers are used for scalar integer operations and address calculation. The register file consists of two read ports and one write port and is fully bypassed to minimize operation latency in the pipeline. Figure 1.2 shows the R4650 CPU registers.

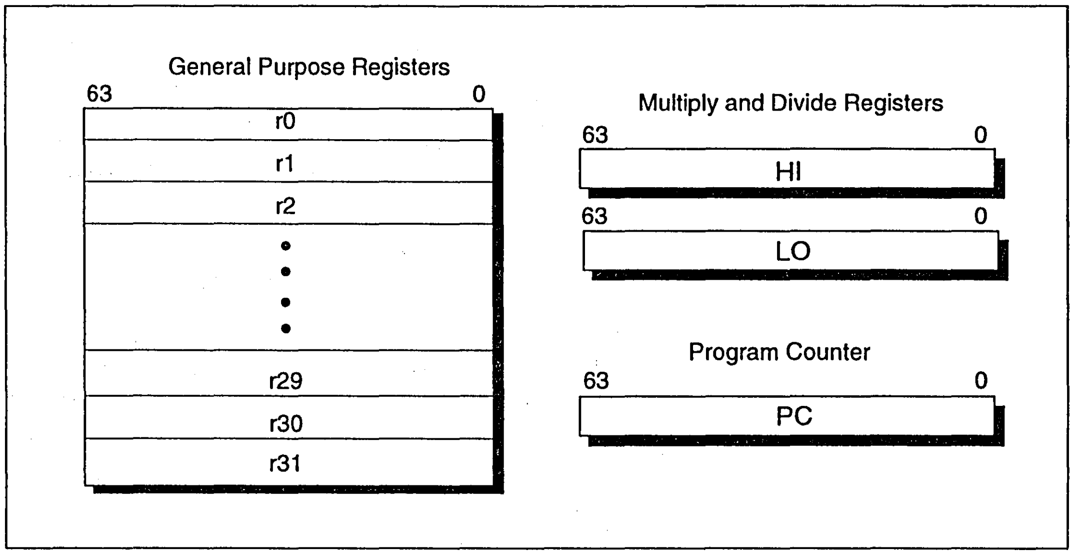


Figure 1.2 R4650 CPU Registers

Two of the CPU general purpose registers have the following assigned functions:

- *r0* is hardwired to a value of zero, and can be used as the target register for any instruction whose result is to be discarded. *r0* can also be used as a source when a zero value is needed.
- *r31* is used as an implicit return destination address register by the JAL and BAL series of instructions.
- The CPU also has these three special purpose registers:
- *PC* — Program Counter register
- *HI* — Multiply and Divide register higher result
- *LO* — Multiply and Divide register lower result

Also, the two Multiply and Divide registers (*HI, LO*) will store **1)** the product of integer multiply operations, or **2)** the quotient (in *LO*) and remainder (in *HI*) of integer divide operations.

The R4650 processor does not have a *Program Status Word* (PSW) register as such. The PSW function is covered by the *Status* and *Cause* registers incorporated within the System Control Coprocessor (CP0). CP0 registers are described later in this chapter.

CPU Instruction Set Overview

Each CPU instruction is 32 bits long. As shown in Figure 1.3, there are three instruction formats:

- immediate (I-type)
- jump (J-type)
- register (R-type)

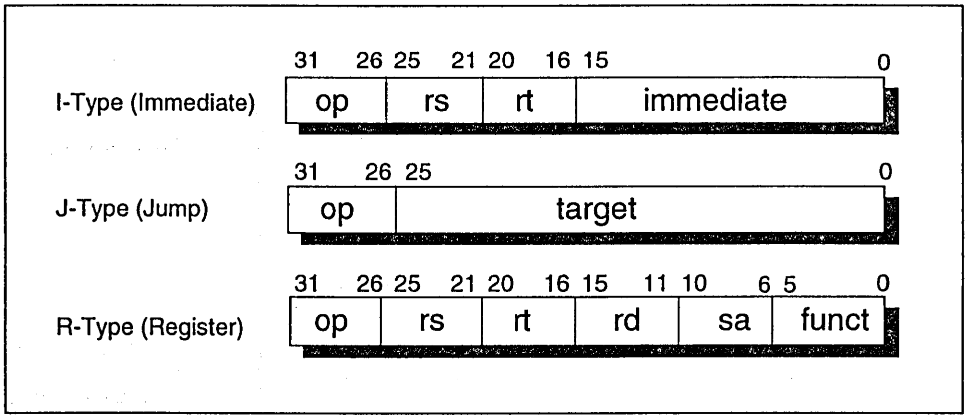


Figure 1.3 CPU Instruction Formats

Each format contains a number of different instructions, which are described further in this chapter. Fields of the instruction formats are described in Chapter 2.

By limiting the number of formats to these three, instruction decoding is simplified. Through this limitation, more complicated (and less frequently used) operations and addressing modes can be synthesized by the compiler, using sequences of these same simple instructions.

The instruction set can be further divided into the following groups:

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

Chapter 2 provides more detailed information on these instructions. And a complete description of each is located in Appendix A.

CPU Instruction Tables

Tables 1.1 through 1.13 lists CPU instructions common to MIPS R-Series processors, along with the level in which they first appeared. The last column of each table refers to the MIPS ISA level in which the instruction first appeared. Table 1.10 shows CP0 instructions.

OpCode	Description	MIPS ISA Level[†]
LB	Load Byte	I
LBU	Load Byte Unsigned	I
LH	Load Halfword	I
LHU	Load Halfword Unsigned	I
LW	Load Word	I
LWL	Load Word Left	I
LWR	Load Word Right	I
SB	Store Byte	I
SH	Store Halfword	I
SW	Store Word	I
SWL	Store Word Left	I
SWR	Store Word Right	I
LD	Load Doubleword	III
LDL	Load Doubleword Left	III
LDR	Load Doubleword Right	III
LL	Load Linked	II
LLD	Load Linked Doubleword	III
LWU	Load Word Unsigned	III
SC	Store Conditional	II
SCD	Store Conditional Doubleword	III
SD	Store Doubleword	III
SDL	Store Doubleword Left	III
SDR	Store Doubleword Right	III
SYNC	Sync	II

Note: [†]For Tables 1.1 through 1.17 this column refers to the level in which the instruction first appeared.

Table 1.1 Instruction Set: MIPS 1 /MIPS 2/MIPS 3 Load and Store Instructions

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

Table 1.2 CPU Instruction Set: MIPS 1 /MIPS 2/ MIPS 3 Arithmetic Instructions (ALU Immediate)

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

Table 1.3 CPU Instruction Set: Arithmetic (3-Operand, R-Type)

OpCode	Description	MIPS ISA Level
MAD	Multiply-Add	†
MADU	Multiply-Add Unsigned	†
MUL	3-Operand Multiply	†
MULT	Multiply (result in HI/LO)	I
MULTU	Multiply Unsigned (result in HI/LO)	I
DIV	Divide	I
DIVU	Divide Unsigned	I
MFHI	Move From HI	I
MTHI	Move To HI	I
MFLO	Move From LO	I
MTLO	Move To LO	I
DMULT	Doubleword Multiply	III
DMULTU	Doubleword Multiply Unsigned	III
DDIV	Doubleword Divide	III
DDIVU	Doubleword Divide Unsigned	III
Note: †These are IDT-proprietary extensions to the MIPS instruction set.		

Table 1.4 CPU Instruction Set: MIPS 1, MIPS 2, MIPS 3 Multiply and Divide Instructions

OpCode	Description	MIPS ISA Level
J	Jump	I
JAL	Jump And Link	I
JR	Jump Register	I
JALR	Jump And Link Register	I
BEQ	Branch on Equal	I
BNE	Branch on Not Equal	I
BLEZ	Branch on Less Than or Equal to Zero	I
BGTZ	Branch on Greater Than Zero	I
BLTZ	Branch on Less Than Zero	I
BGEZ	Branch on Greater Than or Equal to Zero	I
BLTZAL	Branch on Less Than Zero And Link	I
BGEZAL	Branch on Greater Than or Equal to Zero And Link	I
BEQL	Branch on Equal Likely	II
BNEL	Branch on Not Equal Likely	II
BLEZL	Branch on Less Than or Equal to Zero Likely	II
BGTZL	Branch on Greater Than Zero Likely	II
BLTZL	Branch on Less Than Zero Likely	II
BGEZL	Branch on Greater Than or Equal to Zero Likely	II
BLTZALL	Branch on Less Than Zero And Link Likely	II
BGEZALL	Branch on Greater Than or Equal to Zero And Link Likely	II
BCzTL	Branch on Coprocessor z True Likely	II
BCzFL	Branch on Coprocessor z False Likely	II

Table 1.5 CPU Instruction Set: Jump and Branch Instruction

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

Table 1.6 CPU Instruction Set: Shift Instructions

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

Table 1.7 Instruction Set: Coprocessor Instructions

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

Table 1.8 CPU Instruction Set: Special Instructions

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

Table 1.9 MIPS 2/MIPS 3 Exception Instructions

OpCode	Description	MIPS ISA Level
DMFC0	Doubleword Move From CP0	III
DMTC0	Doubleword Move To CP0	III
MTC0	Move to CP0	I
MFC0	Move from CP0	I
TLBR	Read Indexed TLB Entry	I
TLBWI	Write Indexed TLB Entry	I
TLBWR	Write Random TLB Entry	I
TLBP	Probe TLB for Matching Entry	I
CACHE	Cache Operation	R4xxx only
ERET	Exception Return	R4xxx only
WAIT	Enter Standby mode	Orion family

Table 1.10 R4650 CP0 Instructions

Data Formats and Addressing

The R4650 processor uses four data formats: a 64-bit doubleword, a 32-bit word, a 16-bit halfword, and an 8-bit byte. Byte ordering within each of the larger data formats—halfword, word, doubleword—can be configured in either big-endian or little-endian order. Endianness refers to the location of byte 0 within the multi-byte data structure. Figures 1.4 and 1.5 show the ordering of bytes within words and the ordering of words within multiple-word structures for the big-endian and little-endian conventions.

When the R4650 processor 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. Figure 1.4 illustrates this configuration.

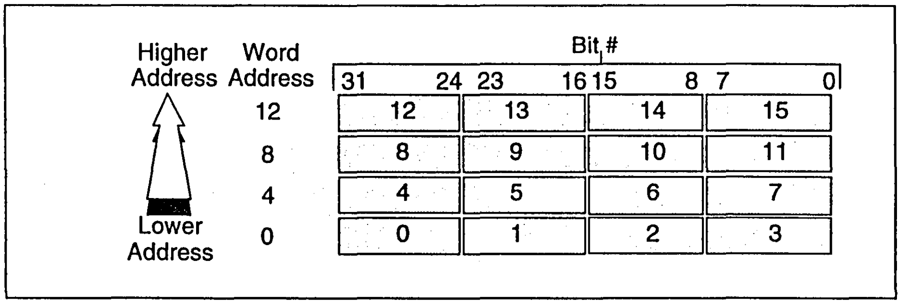


Figure 1.4 Big-Endian Byte Ordering

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. Figure 1.5 illustrates this configuration.

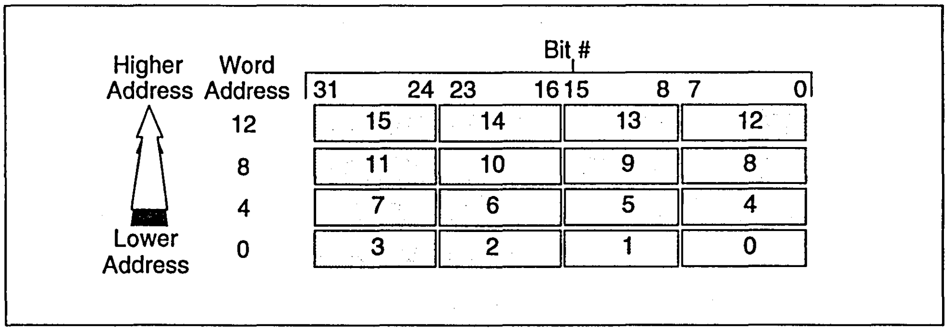


Figure 1.5 Little-Endian Byte Ordering

In this text, 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).

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

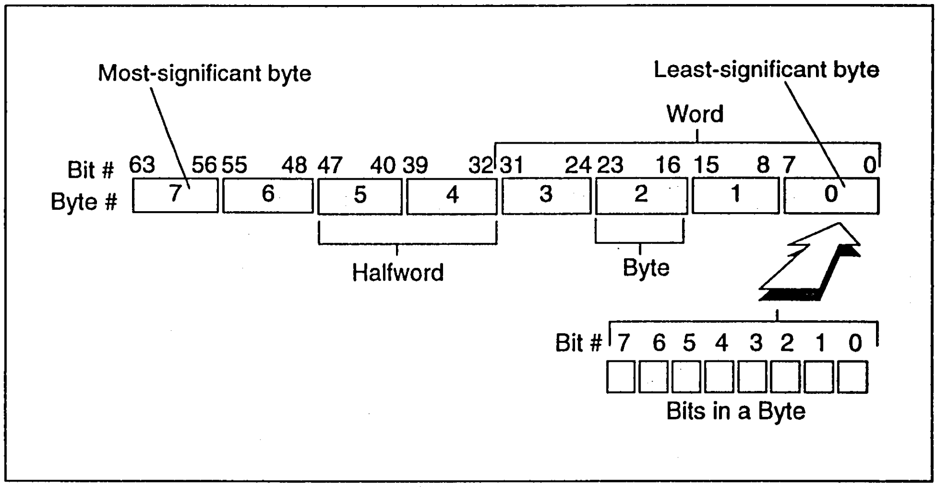


Figure 1.6 Little-Endian Data in a Doubleword

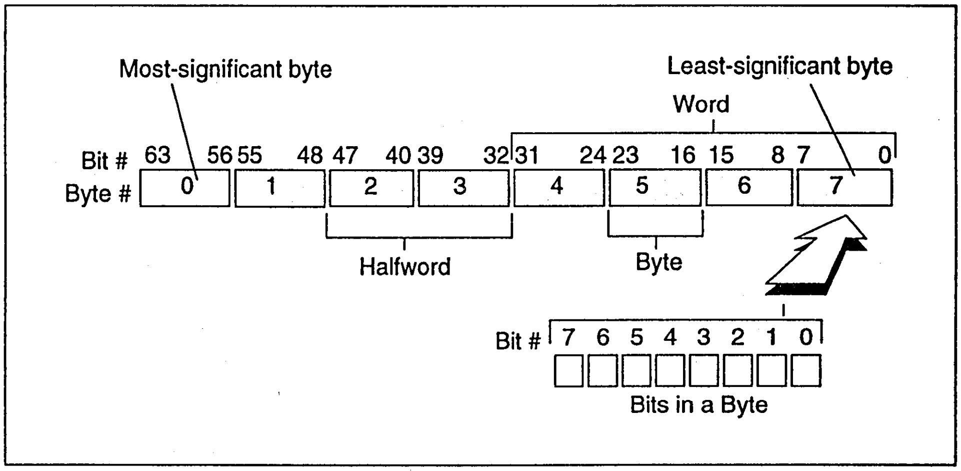


Figure 1.7 Big-Endian Data in a Doubleword

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

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

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

LWL	LWR	SWL	SWR
LDL	LDR	SDL	SDR

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

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

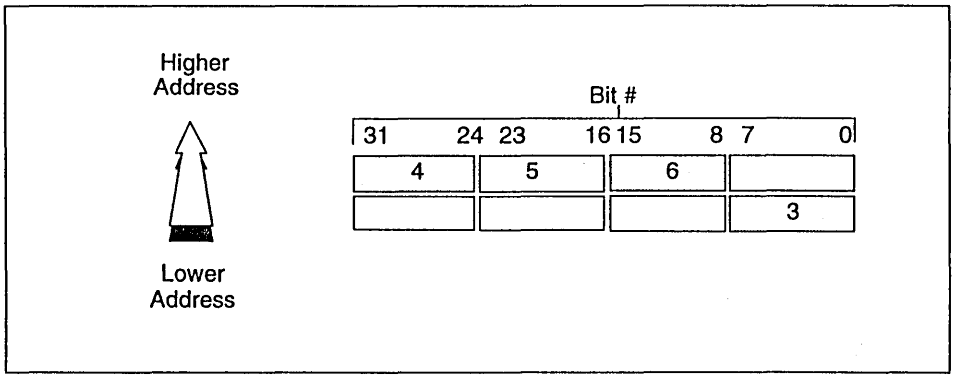


Figure 1.8 Big-Endian Misaligned Word Addressing

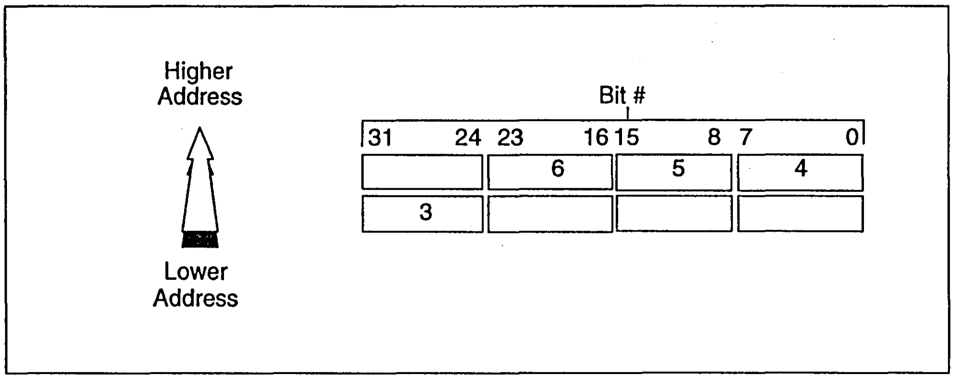


Figure 1.9 Little-Endian Misaligned Word Addressing

Coprocessors (CP0-CP2)

The MIPS ISA (MIPS III Instruction Set with IDT extensions) of the R4650 defines three coprocessors, designated CP0 through CP2:

- Coprocessor 0 (**CP0**) is incorporated on the CPU chip and supports the virtual memory system and exception handling. CP0 is also referred to as the *System Control Coprocessor.*
- Coprocessor 1 (**CP1**) is incorporated on the R4650, and implements the MIPS single-precision floating-point instruction set.
- Coprocessor 2 (**CP2**) is reserved for future use.

CP0 and CP1 of the R4650 are described in the sections that follow.

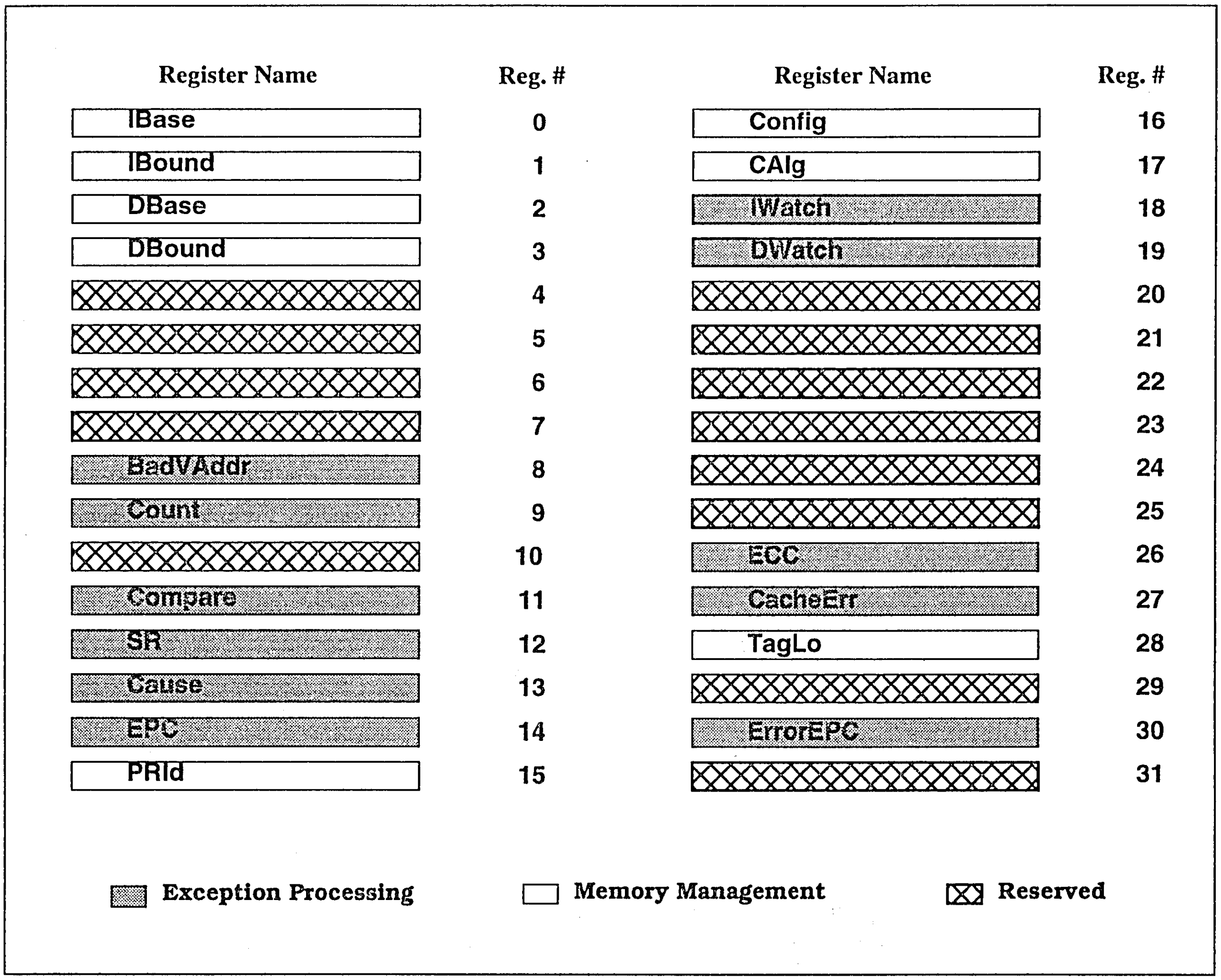
System Control Coprocessor, CP0

CP0 translates virtual addresses into physical addresses and manages exceptions and transitions between kernel and user states. CP0 also controls the cache subsystem, as well as providing diagnostic control and error recovery facilities.

CP0 is also used to control the power management for the R4650. This is the standby mode and it can be used to reduce the power consumption of the internal core of the CPU. The standby mode is entered by executing the WAIT instruction with the SysAD bus idle and is exited by any interrupt. This feature is discussed in Appendix D.

The CP0 registers shown in Figure 1.10 and described in Table 1.11 manipulate the memory management and exception handling capabilities of the CPU.

Note: Access to reserved or undefined CP0 register results are undefined. An exception may or may not result.



Register Name	Reg. #	Register Name	Reg. #
□ IBase	0	□ Config	16
□ IBound	1	□ CAlg	17
□ DBase	2	▨ IWatch	18
□ DBound	3	▨ DWatch	19
☒	4	☒	20
☒	5	☒	21
☒	6	☒	22
☒	7	☒	23
▨ BadVAddr	8	☒	24
▨ Count	9	☒	25
☒	10	▨ ECC	26
▨ Compare	11	▨ CacheErr	27
▨ SR	12	□ TagLo	28
▨ Cause	13	☒	29
▨ EPC	14	▨ ErrorEPC	30
□ PRId	15	☒	31

▨ Exception Processing □ Memory Management ☒ Reserved

Figure 1.10 R4650 CP0 Registers

Number	Register	Description
0	IBase	Provides the User Instruction address space Base
1	IBound	Provides the User Instruction address space Bound
2	DBase	Provides the User Data address space Base
3	DBound	Provides the User Data address space Bound
4	—	Reserved
5	—	Reserved
6	—	Reserved
7	—	Reserved
8	BadVAddr	Bad virtual address
9	Count	Timer Count
10	—	Reserved
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	CAlg	Cache attributes control
18	IWatch	A read/write register that specifies an Instruction virtual address that causes a Watch exception.
19	DWatch	A read/write register that specifies a Data virtual address that causes a Watch exception.
20	—	Reserved
21–25	—	Reserved
26	ECC	Secondary-cache error checking and correcting (ECC) and Primary parity
27	CacheErr	Cache Error and Status register
28	TagLo	Cache Tag register
29	—	Reserved
30	ErrorEPC	Error Exception Program Counter
31	—	Reserved

Table 1.11 System Control Coprocessor (CP0) Register Definitions

Floating-Point Co-Processor

The R4650 incorporates an entire single-precision floating-point co-processor on chip, including a floating-point register file and execution units. The floating-point co-processor forms a "seamless" interface with the integer unit, decoding and executing instructions in parallel with the integer unit.

Floating-Point Units

The R4650 floating-point execution units perform single-precision arithmetic, as specified in the IEEE Standard 754. The execution unit is broken into a separate multiply unit and a combined add/convert/divide/square root unit. Overlap of multiplies and add/subtract is supported. The multiplier is partially pipelined, allowing a new multiply to begin every 6 cycles.

As in the IDT79R4600, the R4650 maintains fully precise floating-point exceptions while allowing both overlapped and pipelined operations. Precise exceptions are extremely important in mission-critical environments, and highly desirable for debugging in any environment.

The floating-point unit's operation set includes floating-point add, subtract, multiply, divide, square root, conversion between fixed-point and floating-point format, and floating-point compare. These operations comply with IEEE Standard 754. Double-precision operations are not directly supported; attempts to execute double-precision floating point operations, or refer directly to double-precision registers, result in the R4650 signalling a "trap" to the CPU, enabling emulation of the requested function.

Table 1.12 gives the latencies of some of the floating-point instructions in internal processor cycles.

Operation	Instruction Latency
ADD	4
SUB	4
MUL	8
DIV	32
SQRT	31
CMP	3
FIX	4
FLOAT	6
ABS	1
MOV	1
NEG	1
LWC1	2
SWC1	1

Table 1.12 Floating-Point Operation

Virtual to Physical Address Mapping

The R4650 provides two modes of operation:

- user mode
- kernel mode

Kernel mode operation is typically used for exception handling and operating system kernel functions, including CP0 management and access to IO devices. In kernel mode, software has access to the entire address space and all of the co-processor 0 registers and can select whether to enable co-processor 1 accesses. The processor enters kernel mode at reset, or whenever an exception is recognized.

User mode operation is typically used for applications programs. User mode accesses are limited to a subset of the virtual address space, and can be inhibited from accessing CP0 functions. The 4 GB address space, which is shown in Table 1.13, is divided into addresses accessible in either kernel or user mode (kuseg), and addresses only accessible in kernel mode (kseg2:0).

0xFFFFFFFF 0xC0000000	Kernel virtual address space (kseg2) Unmapped, 1.0 GB
0xBFFFFFFF 0xA0000000	Uncached kernel physical address space (kseg1) Unmapped, 0.5GB
0x9FFFFFFF 0x80000000	Cached kernel physical address space (kseg0) Unmapped, 0.5GB
0x7FFFFFF 0x00000000	User virtual address space (useg) Mapped, 2.0GB

Table 1.13 Mode Virtual Addressing (32-bit mode)

Sharing common virtual addresses but mapped to separate physical addresses, the R4650 supports the use of multiple user tasks. This facility is implemented via the "base-bounds" registers contained in CP0.

When a user virtual address is asserted (load, store, or instruction fetch), the R4650 compares the virtual address with the contents of the appropriate "bounds" register (instruction or data). If the virtual address is "in bounds," the value of the corresponding "base" register is added to the virtual address to form the physical address for that reference. If the address is not within bounds, an exception is signalled.

This facility enables multiple user processes in a single physical memory without the use of a TLB. This type of operation is further supported by a number of development tools for the R4650, including real-time operating systems and "position independent" code.

Kernel mode addresses do not use the base-bounds registers, but rather undergo a fixed virtual to physical address translation.

A detailed explanation of this addressing mechanism is given in Chapter 4.

Base Bounds Registers

The R4650 implements a simple mechanism to support the mapping of virtual to physical addresses. In the R4650, the TLB structure found in the IDT79R4600 has been replaced by a *base-bounds* mechanism. When an address is translated, its page number is first compared against the Bounds register. If the address is "in range," the base register is added to the virtual address to form the physical address.

The R4650 contains two sets of base-bounds registers, one set for instruction address translation (IBase and IBounds registers) and one for data (DBase and DBounds registers). An operating system can support task protection by writing appropriate values to these registers at context switch time.

Finally, to allow a mix of cache attributes in a single system, the R4650 also implements a *Cache Algorithm (CAlg)* register in CP0. This register allows the operating system to define the cache management attributes of different portions of the address space. By using appropriate virtual addresses, memory can be treated as uncached, write-back, or write-through, with separate attributes for each of eight memory regions. In conjunction with the external system address decoder, software can then alias the same physical memory with different management algorithms, depending upon the data or program that is running.

Cache Memory

To keep the R4650's high-performance pipeline full and operating efficiently, the R4650 incorporates on-chip instruction and data caches that can be accessed in a single processor cycle. Each cache has its own 64-bit data path and can be accessed in parallel. The cache subsystem provides the integer and floating-point units with an aggregate bandwidth of over 1.5GB per second.

Instruction Cache

The R4650 incorporates a two-way set associative on-chip instruction cache. This virtually indexed, physically tagged cache is 8KB in size and is protected with word parity.

Because the cache is virtually indexed, the virtual-to-physical address translation occurs in parallel with the cache access, thus further increasing performance by allowing these two operations to occur simultaneously. The tag holds a 24-bit physical address and valid bit and is parity protected.

The instruction cache is 64-bits wide and can be refilled or accessed in a single processor cycle. Instruction fetches require only 32 bits per cycle, for a peak instruction bandwidth of 532 MB/sec at 133MHz. Sequential accesses take advantage of the 64-bit fetch to reduce power dissipation, and cache miss refill writes 64 bits per cycle to minimize the cache miss penalty. To maximize performance, the line size is eight instructions (32 bytes).

In addition, the contents of one set of the instruction cache (set "A") can be "locked" by setting a bit in a CP0 register. Locking the set prevents its contents from being overwritten by a subsequent cache miss; refill occurs then only into "set A".

This operation effectively "locks" time critical code into one 4KB set, while allowing the other set to service other instruction streams in a normal fashion. Thus, the benefits of cached performance are achieved, while deterministic real-time response is preserved.

Data Cache

For fast, single cycle data access, the R4650 includes an 8KB on-chip data cache that is two-way set associative with a fixed 32-byte (eight word) line size. Both the D-cache and the I-cache can be accessed each pipeline cycle; thus, the data bandwidth is over 1 MB/sec at 133 MHz, in addition to the 532 MB/sec instruction bandwidth.

The data cache is protected with byte parity and its tag is protected with a single parity bit. It is virtually indexed and physically tagged to allow simultaneous address translation and data cache access

The D-cache allows write-back and write-through operation functions of the address space to be individually controlled through a field in the CAlg register. Once initialized, software need only assert the desired virtual address to get the desired effect.

Associated with the data cache is the store buffer. When the R4650 executes a store instruction, this single-entry buffer gets written with the store data while the tag comparison is performed. If the tag matches, then the data is written into the data cache in the next cycle that the data cache is not accessed (the next non-load cycle). The store buffer allows the R4650 to execute a store every processor cycle and to perform back-to-back stores without penalty.

Write buffer

Writes to external memory, whether cache miss write-backs or stores to uncached or write-through addresses use the on-chip write buffer. The write buffer holds up to four 64-bit address and data pairs or 1 cache line to be written back. The entire buffer is used for a data cache write-back and allows the processor to proceed in parallel with memory update. For uncached and write-through stores, the write buffer has significantly increased performance over other R4000-family processors.

R4650 Clocks

The R4650 uses the system interface clock as its input clock. The pipeline speed is derived from this clock using a PLL to multiply up the input reference. It is assumed that the system designer manages the system clock distribution to fit the needs of the system. Thus, the R4650 does not output a system reference clock, but rather operates in synchronization with the input clock.

The R4650 does output one low frequency reference clock: the Mode clock. This clock operates at 1/256 the rate of the input clock, and it is used to clock in the serial initialization stream during reset.

System Interface

The R4650 supports a 64-bit system interface that is compatible with the R4400PC system interface. This interface operates from the input Reference clock.

The interface consists of a 64-bit address/data bus with 8 check bits and a 9-bit command bus. There are also 8 handshake signals and 6 interrupt inputs. The interface has a simple timing specification and is capable of transferring data between the processor and memory at a peak rate of 400MB/sec at 50MHz.

In addition, the R4650 supports a boot-time option to run the system interface as 32 bits wide, using basically the same protocols as a 64-bit system. This feature allows the system designer to reduce the costs of the overall memory system without sacrificing computational performance.

Figure 1.11 shows a typical system using the R4650. In this example there is DRAM, a boot EPROM, and an optional secondary cache.

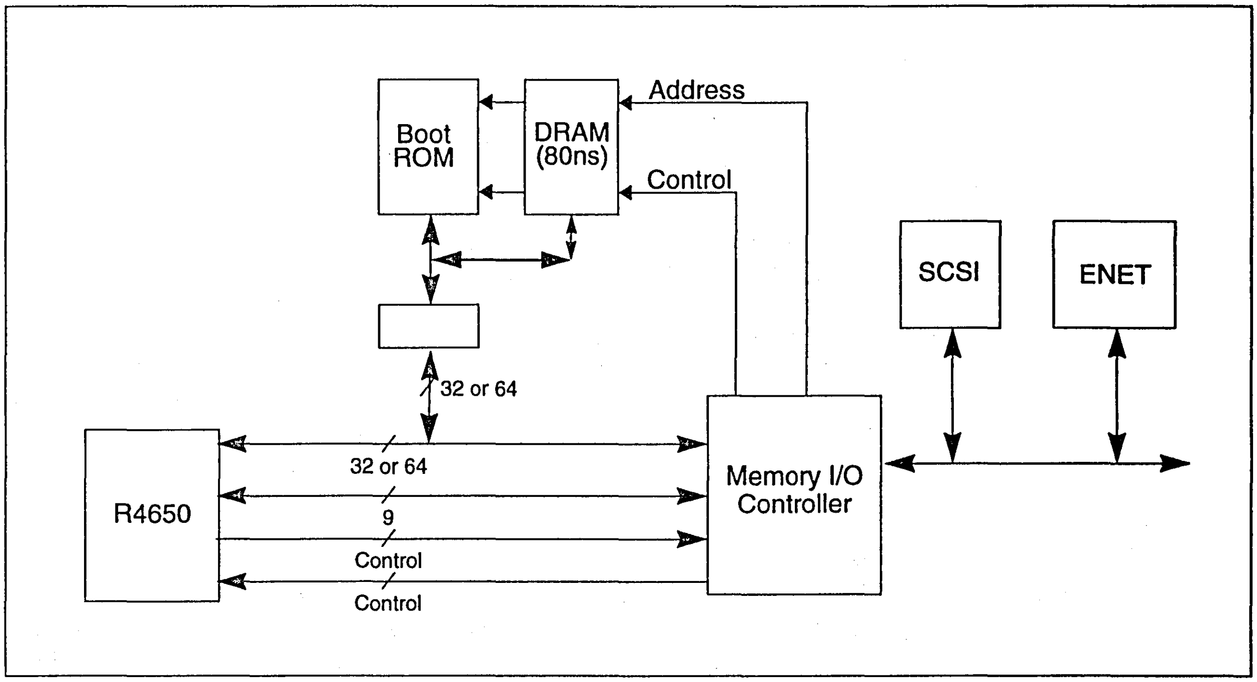


Figure 1.11 Typical System Block Diagram

Comparison of R4650 and R4600/R4700

Table 1.14 compares R4650 features with those of the R4600/R4700. This list is not exhaustive.

Attribute	R4600/R4700	R4650
I-Cache size	16KB	8KB
D-Cache size	16KB	8KB
Cacheability control	TLB, K0 field	CAlg
Memory translation	TLB	Base-Bounds
Floating point accelerator	Single- and double-precision	Single-precision only
Integer multiply	MIPS standard only 12 cycles	MIPS standard + 3 operand Mul (2-3 cycles)
Integer multiply-add	No	Yes 2-3 cycle repeat rate
Clock interface	Input clock at 1/2 pipeline; System clock derived from pipeline clock multiple output reference clocks.	Input clock is system clock; pipeline clock derived from there; no system output clock
Bus interface width	64-bit	32-bit or 64-bit
Watch registers	None	I-Watch and D-Watch
Cache locking	No	Yes (per set)
Separate Interrupt vector	No	Yes (optional)

Table 1.14 System Interface Comparison Between R4600 /R4700 PC and R4650

Integrated Device Technology, Inc.

CPU Instruction Set Overview

Chapter 2

Introduction

This chapter is an overview of the central processing unit (CPU) instruction set. For a description of an individual CPU instruction refer to Appendix A, "CPU Instruction Set Details."

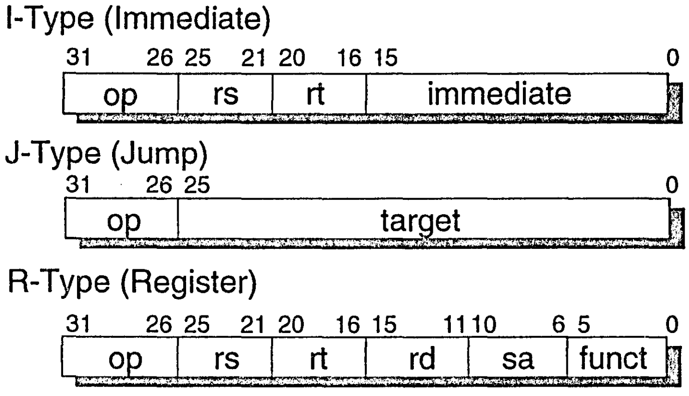
For an overview of the floating-point unit (FPU) instruction set refer to Chapter 6, "The Floating Point Unit." For a description of an individual FPU instruction refer to Appendix B, "FPU Instruction Set Details."

CPU Instruction Formats

Each CPU instruction consists of a single 32-bit word, aligned on a word boundary. There are three instruction formats, as shown in Figure 2.1:

- Immediate (I-type)
- Jump (J-type)
- Register (R-type)

The use of a small number of instruction formats simplifies instruction decoding (thus higher frequency operations) and allowing the compiler to synthesize more complicated (and less frequently used) operations and addressing modes from these three formats as needed.



Key to Figure:

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

Figure 2.1 CPU Instruction Formats

In the MIPS architecture, coprocessor instructions are implementation-dependent; refer to Appendix A for details of individual Coprocessor 0 instructions.

Load and Store Instructions

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

Scheduling a Load Delay Slot

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

In the R4650 processor, the instruction immediately following a load instruction can request the contents of the loaded register, however, in such cases, hardware interlocks insert additional real cycles. Consequently, scheduling load delay slots can be desirable, both for performance and R-Series (e.g., R3051) processor compatibility. However, the scheduling of load delay slots is not absolutely required.

Defining Access Types

Access type indicates the size of an R4650 processor data item to be loaded or stored, set by the load or store instruction opcode. Access types are defined in Appendix A.

Regardless of access type or byte ordering (endianness), the address given specifies the low-order byte in the addressed field. For a big-endian configuration, the low-order byte is the most-significant byte; for a little-endian configuration, the low-order byte is the least-significant byte.

The access type, together with the three low-order bits of the address, define the bytes accessed within the addressed doubleword, which is shown in Table 2.1. Only the combinations shown in this table are permissible. Other combinations will cause address error exceptions.

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

Table 2.1 Byte Access within a Doubleword

Computational Instructions

Computational instructions can be in either of the following formats:

- register (R-type) format, in which both operands are registers.
- immediate (I-type) format, in which one operand is a 16-bit immediate.

Computational instructions perform the following operations on register values:

- arithmetic
- logical
- shift
- multiply
- divide

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

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

Operations With 32-bit Operands

Operands to 32-bit operand opcodes must be in sign-extended form. 32-bit operand opcodes include all non-doubleword operations, such as: ADD, ADDU, SUB, SUBU, ADDI, SLL, SRL, SRA, SLLV, etc. The result of operations that use incorrect sign-extended 32-bit values is unpredictable.

Cycle Timing for Multiply and Divide Instructions

R4650 hardware *interlocks* if necessary in order to allow complete execution of the multiply and divide instructions. *Latency* is the number of clock cycles until the result is available. *Repeat* is the number of clock cycles until the instruction can be repeated. *Stall* is the number of clock cycles the CPU will automatically stall.

MFHI and MFLO instructions (which are described in more detail in Appendix A) are interlocked so that any attempt to read them before prior multiply or divide instructions complete delays the execution of these instructions until the prior instructions finish.

Table 2.2 gives the number of processor cycles (PCycles) required to resolve an interlock or stall between various multiply or divide instructions, and a subsequent MFHI or MFLO instruction.

Opcode	Operand * Size	Latency	Repeat	Stall
MULT/U, MAD/U	16 bit	3	2	0
	32 bit	4	3	0
MUL	16 bit	3	2	1
	32 bit	4	3	2
DMULT, DMULTU	any	6	5	0
DIV, DIVU	any	36	36	0
DDIV, DDIVU	any	68	68	0

* The R4650 automatically detects operand size.

Note: For more information about these computational instructions, refer to Appendix A.

Table 2.2 R4650 Integer Multiply Operation

Integrated Device Technology, Inc.

The CPU Pipeline

Chapter 3

Introduction

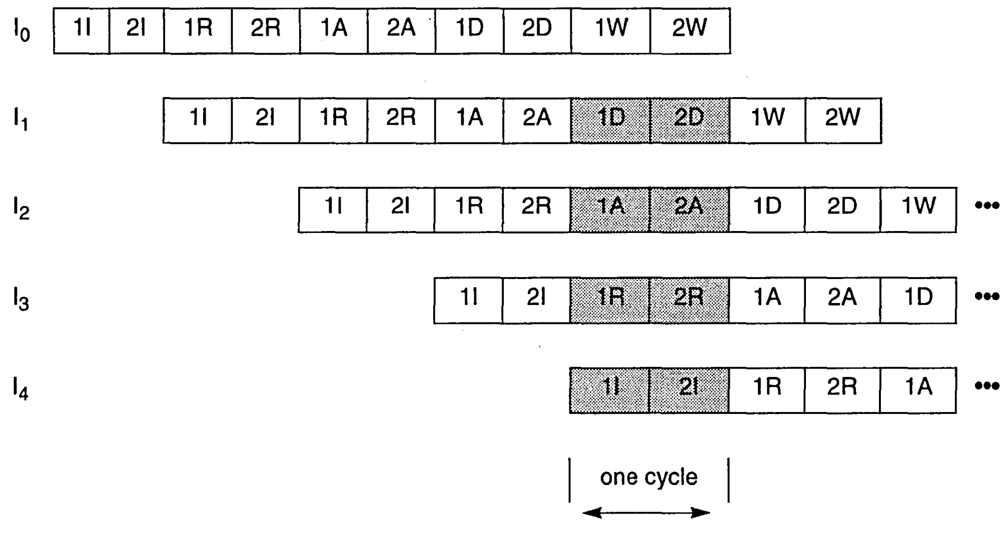
This chapter describes the basic operation of the CPU pipeline, including descriptions of the delay instructions (instructions that follow a branch or load instruction in the pipeline), interruptions to the pipeline flow caused by interlocks and exceptions, and R4650 implementation of an uncached store buffer. The FPU pipeline is described in a later chapter.

CPU Pipeline Operation

The R4650 uses a 5-stage pipeline similar to the R3000. The simplicity of this pipeline allows the R4650 to be lower cost and lower power than super-scalar or super-pipelined processors. Unlike the R3000, the R4650 does virtual to physical translation in parallel with cache access. This allows the R4650 to operate at over twice the frequency of the R3000 and to support a "base-bounds" register for address translation.

Compared to the 8-stage R4000 pipeline, the R4650 is more efficient because fewer stalls are required.

Once the pipeline has been filled, five instructions are executed simultaneously. Figure 3.1 shows the five stages of the instruction pipeline; the next section describes the pipeline stages.


I_0 1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
I_1 1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
I_2 1I 2I 1R 2R 1A 2A 1D 2D 1W •••
I_3 1I 2I 1R 2R 1A 2A 1D •••
I_4 1I 2I 1R 2R 1A •••
one cycle

Key to Figure:

1I-1R	Instruction cache access	2R	Instruction decode
1I-2I	Instruction virtual to physical address translation	1A-2A	Integer add, logical, shift
2A-2D	Data cache access and load align	1A	Data virtual address calculation
1D-2D	Data virtual to physical address translation	2A	Store align
2R	Register file read	1A	Branch decision
2R	Bypass calculation	2W	Register file write

Figure 3.1 Instruction Pipeline Stages

CPU Pipeline Stages

This section describes each of the phases of the five pipeline stages. Each stage has 2 phases:

- 1I - Instruction Fetch, Phase one
- 2I - Instruction Fetch, Phase two
- 1R - Register Fetch, Phase one
- 2R - Register Fetch, Phase two
- 1A - Execution, Phase one
- 2A - Execution, Phase two
- 1D - Data Fetch, Phase one
- 2D - Data Fetch, Phase two
- 1W - Write Back, Phase one
- 2W - Write Back, Phase two

1I - Instruction Fetch, phase one

The instruction address translation begins during the 1I phase.

2I - Instruction Fetch, phase two

During the 2I phase, the instruction cache fetch begins and the instruction address translation continues.

1R - Register Fetch, phase one

During the 1R phase, the following occurs:

- The instruction cache fetch finishes.
- The instruction cache tag is checked against the physical page frame number obtained from the address translation.

2R - Register Fetch, phase two

During the 2R phase, the following occurs:

- The instruction decoder decodes the instruction.
- Any required operands are fetched from the register file.
- Make a decision to either issue or slip (for an interlock condition).
- For a branch, the branch address is calculated.

1A - Execution, phase one

During the 1A phase, one of the following occurs:

- Any result from the A or D stages are bypassed.
- The arithmetic logic unit (ALU) starts the integer arithmetic, logical or shift operation.
- The ALU calculates the data virtual address for load and store instructions.
- The ALU determines whether the branch condition is true.

2A - Execution, phase two

During the 2A phase, one of the following occurs:

- The integer arithmetic, logical or shift operation will complete.
- A data cache access will start.
- Store data is shifted to the specified byte position(s).
- The data virtual to physical address translation will start.

1D - Data Fetch, phase one

During the 1D phase, one of the following occurs:

- The data cache access will continue.
- The data address translation completes.

2D - Data Fetch, phase two

During the 2D phase the data cache access will finish and the data is shifted down and extended. The data cache tag is checked against the physical address for any data cache access.

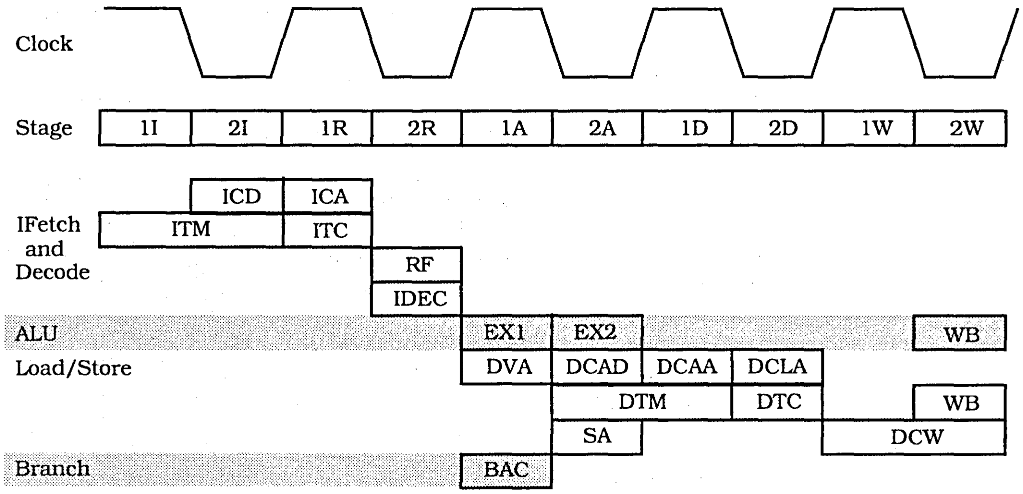
1W - Write Back, phase one

This phase is used internally by the processor to resolve all exceptions, in preparation for the register file write.

2W - Write Back, phase two

For register-to-register and load instructions, the result is written back to the register file during the 2W stage. Branch instructions perform no operation during this stage.

Figure 3.2 shows the activities occurring during each ALU pipeline stage, for load, store, and branch instructions.



Key to Figure:

ICD	Instruction cache address decode	ICA	Instruction cache array access
ITM	Instruction translation match	RF	Register operand fetch
ITC	Instruction tag check	EX1	Operation stage 1
IDEC	Instruction decode	WB	Write back to register file
EX2	Operation stage 2	DCAD	Data cache address decode
DVA	Data virtual address calculation	DCLA	Data cache load align
DCAA	Data cache array access	DTM	Data translation match
DTC	Data tag check	SA	Store align
DCW	Data cache write	BAC	Branch address calculation

Figure 3.2 CPU Pipeline Activities

Branch Delay

The CPU pipeline has a branch delay of one cycle and a load delay of one cycle. The one-cycle branch delay is a result of the branch decision logic operating during the 1A pipeline phase of the branch instruction. This allows the branch target address calculated in the previous phase to be used for the instruction access in the following 1I phase. The pipeline will begin the fetch of the branch path as well as the fall-through path in the cycle following the delay slot. After the branch decision is made, the processor will continue with the fetch of either the branch path (for a taken branch) or the fall-through path (for the non-taken branch).

Figure 3.3 illustrates the branch delay.

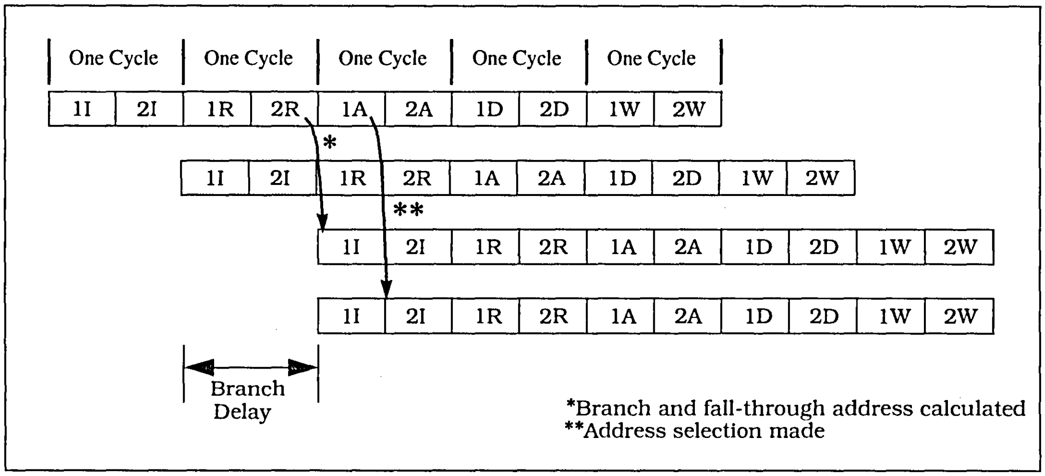

One Cycle
One Cycle
One Cycle
One Cycle
One Cycle
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
*
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
**
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
Branch Delay
*Branch and fall-through address calculated
**Address selection made

Figure 3.3 CPU Pipeline Branch Delay

Load Delay

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

Figure 3.4 shows the load delay of one pipeline cycle.

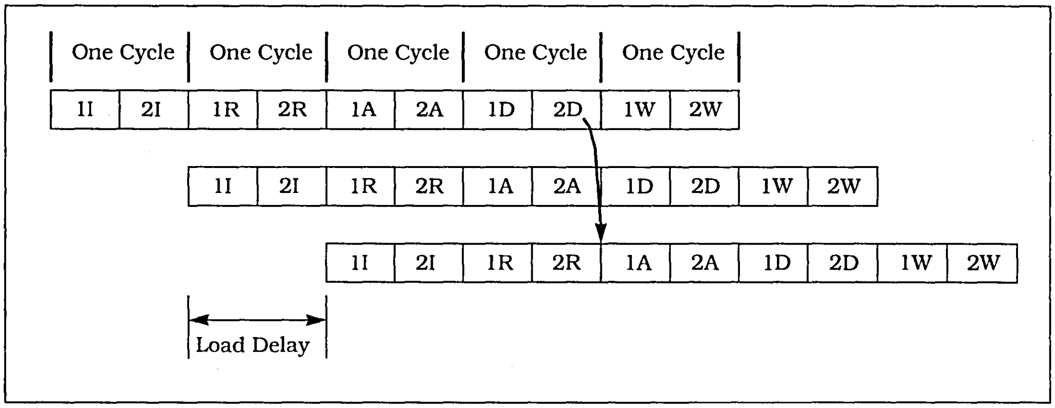

One Cycle
One Cycle
One Cycle
One Cycle
One Cycle
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
Load Delay

Figure 3.4 CPU Pipeline Load Delay

Interlock and Exception Handling

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

There are two types of interlocks:

- stalls, which are resolved by halting the pipeline
- slips, which require the back end of the pipeline to advance while the front end of the pipeline is held static

At each cycle, exception and interlock conditions are checked for all active instructions.

Because each exception or interlock condition corresponds to a particular pipeline stage, a condition can be traced back to the particular instruction in the exception/interlock stage, as shown in Table 3.1. For instance, a Reserved Instruction (RI) exception is raised in the execution (A) stage.

State	**Pipeline Stage**				
	I	**R**	**A**	**D**	**W**
Stall		ICM		DCM	
				CPE	
	I	**R**	**A**	**D**	**W**
Slip		LDI			
		MDSt			
		FCBsy			
	I	**R**	**A**	**D**	**W**
Exceptions	ITM	IBE	RI	DBE	
	IWatch	IPErr	CUn	NMI	
			BP	Reset	
			SC	DPErr	
			DTM	OVF	
			Intr	Trap	
			FPE		
			DWatch		

Table 3.1 Correspondence of Pipeline Stage to Interlock Condition

For a description of the pipeline interlocks and exceptions listed in Table 3.1, refer to Table 3.2 and Table 3.3.

Exception	Description
ITM	Instruction Translation Bound/Address Exception
Intr	External Interrupt
IBE	Instruction Bus Error
RI	Reserved Instruction
BP	Breakpoint
SC	System Call
CUn	Coprocessor Unusable
IPErr	Instruction Parity Error
OVF	Integer Overflow
FPE	FP Interrupt
ExTrap	EX Stage Traps
DTM	Data Translation Bound/Address Exception
DBE	Data Bus Error
DPErr	Data Parity Error
NMI	Non-maskable Interrupt (or Soft Reset)
Reset	Reset

Table 3.2 Pipeline Exceptions

Table 3.2 and Table 3.3 describe the pipeline interlocks and exceptions shown in Table 3.1 on page 5.

Interlock	Description
ICM	Instruction Cache Miss
CPE	Coprocessor Possible Exception
DCM	Data Cache Miss
LDI	Load Interlock
MDSt	Multiply/Divide Start
FCBsy	FP Coprocessor Busy

Table 3.3 Pipeline Interlocks

Exception Conditions

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

When an exceptional condition is detected for an instruction, the R4650 will kill it and all following instructions. When this instruction reaches the W stage, the exception flag causes it to write various CP0 registers with the exception state, change the current PC to the appropriate exception vector address and clear the exception bits of earlier pipeline stages.

This implementation allows all preceding instructions to complete execution and prevents all subsequent instructions from completing. Thus the value in the EPC is sufficient to restart execution. It also ensures that exceptions are taken in the order of execution; an instruction taking an exception may itself be killed by an instruction further down the pipeline that takes an exception in a later cycle.

Figure 3.5 shows the exception detection procedure (e.g., a reserved instruction exception).

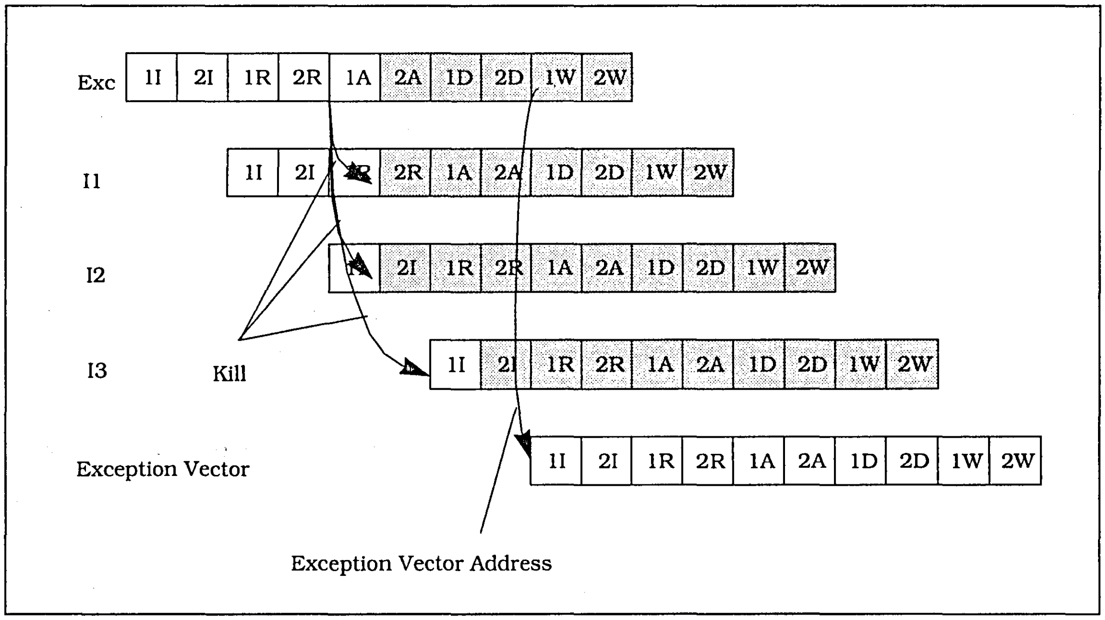

Exc
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
I1
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
I2
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
I3
Kill
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
Exception Vector
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
Exception Vector Address

Figure 3.5 Exception Detection

Stall Conditions

Stalls are used to stop the pipeline for conditions detected after the R pipe-stage. When a stall occurs, the processor will resolve the condition and then the pipeline will continue. Figure 3.6 shows a data cache miss stall.

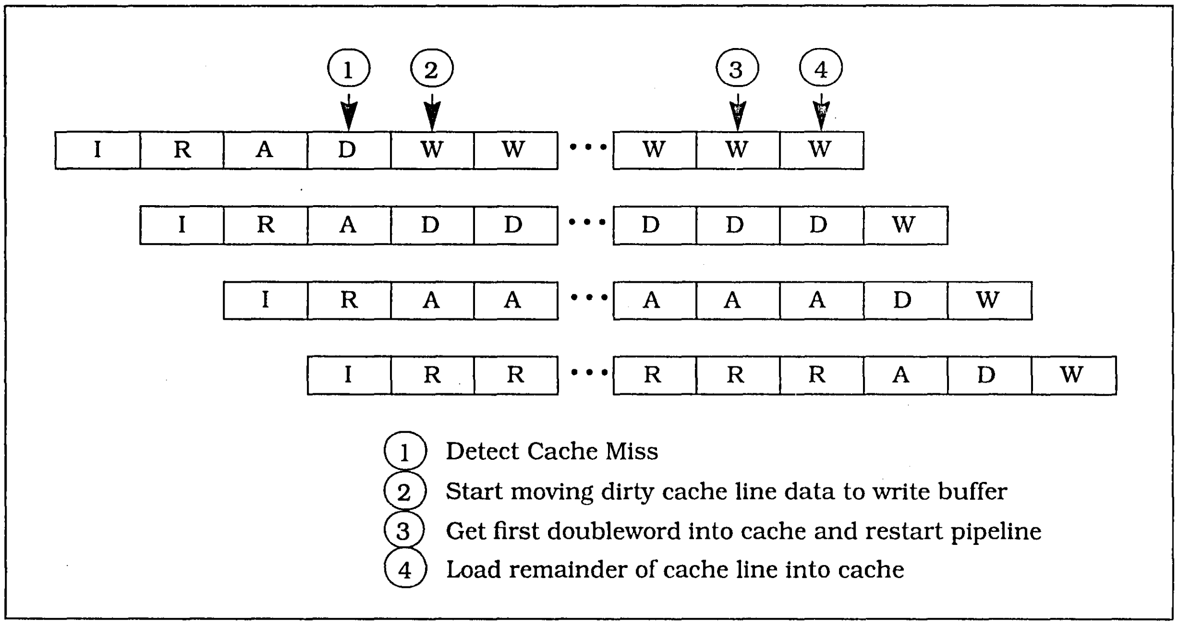
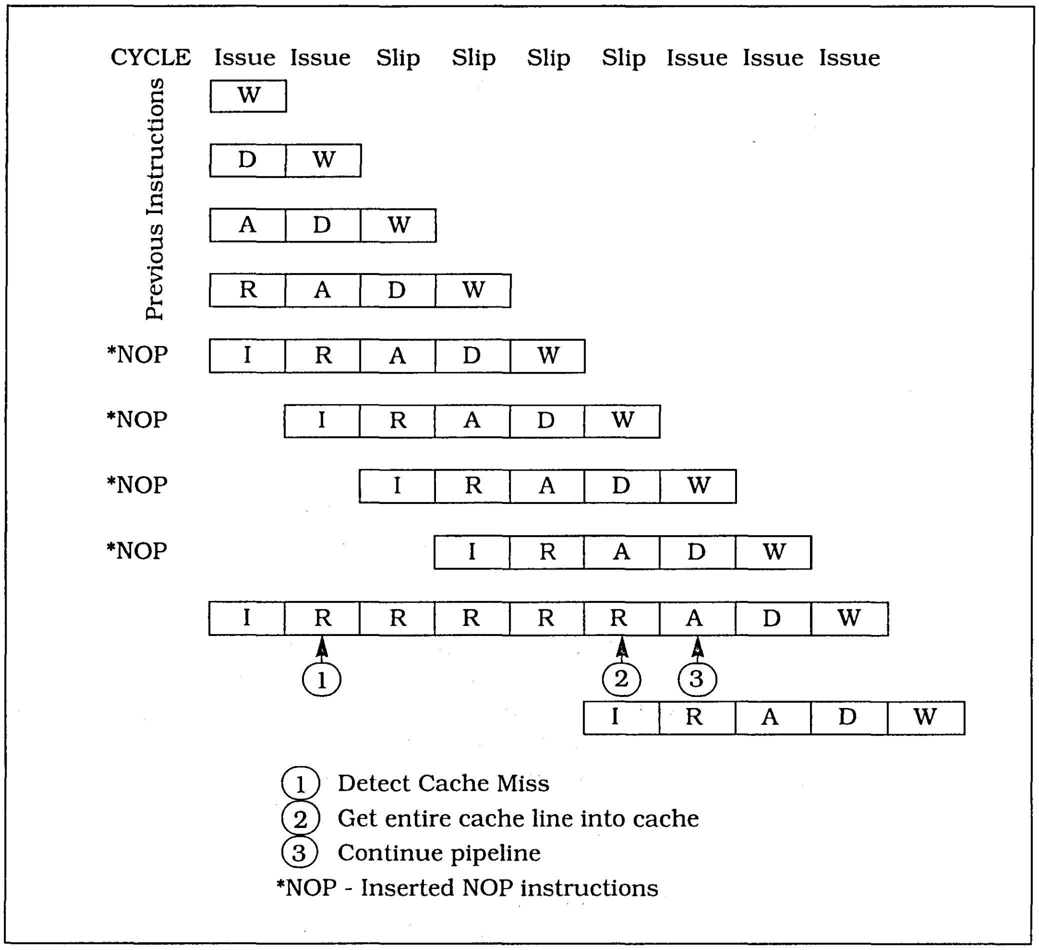

1
2
3
4
I R A D W W ••• W W W
I R A D D ••• D D D W
I R A A ••• A A A D W
I R R ••• R R R A D W
1 Detect Cache Miss
2 Start moving dirty cache line data to write buffer
3 Get first doubleword into cache and restart pipeline
4 Load remainder of cache line into cache

Figure 3.6 Data Cache Miss

The data cache miss is detected in the D pipe stage. If the cache line to be replaced is dirty — the W bit is set — the data is moved to the internal write buffer in the next cycle. The first doubleword of data is returned to the cache in 3 and the pipeline will then restart. The remainder of the cache line is returned in the subsequent cycles. The data to be written back will be returned to memory some time after the entire new cache line is returned.

Slip Conditions

During the 2R and 1A pipe-stages, internal logic will determine whether it is possible to start the current instruction in this cycle. If all of the source operands are available (either from the register file or via the internal bypass logic) and all the hardware resources necessary to complete the instruction will be available at the necessary time(s), then the instruction "issues"; otherwise, the instruction will "slip". Slipped instructions are retried on subsequent cycles until they issue. The backend of the pipeline (stages D and W) will advance normally during slips in an attempt to resolve the conflict. "NOPS" will be inserted into the bubble in the pipeline. Instructions killed by branch likely instructions, ERET or exceptions will not cause slips. Figure 3.7 shows an instruction cache miss.



(1) Detect Cache Miss
(2) Get entire cache line into cache
(3) Continue pipeline
*NOP - Inserted NOP instructions

Figure 3.7 Instruction Cache Miss

As shown in Figure 3.7, instruction cache misses are detected in R and the pipeline slips in its A stage. There can never be a write-back required for an instruction cache miss since dirty data can not exist in the I cache. Writes are not allowed to the I cache. Note that early restart is not employed for instruction cache misses, the requested cache line will be loaded into the cache in its entirety and, after that, the pipeline will restart.

R4650 Write Buffer

The R4650 contains a write buffer to improve the performance of writes to the external memory. Writes to external memory, whether cache miss write-backs or stores to uncached or write-through addresses, use this on-chip write buffer. The write buffer holds up to four 64-bit address and data pairs.

For a cache miss write-back, the entire buffer is used for the write-back data and allows the processor to proceed in parallel with the memory update. For uncached and write-through stores, the write buffer uncouples the CPU from the write to memory allowing increased performance over the R4000 family of processors. If the write buffer is full, additional stores will stall until there is room for them in the write buffer.

Introduction

The R4650 features a simple *base-bounds* mechanism for virtual-to-physical address translation. This mechanism supports multitasking without the overhead of Translation Lookaside Buffer (TLB) management. A companion mechanism that is implemented through the C*ache Algorithm* register allows control over the cache attributes of areas of the address space.

Base Bounds Registers

The R4650 implements a simple mechanism to support the mapping of virtual to physical addresses. The Translation Lookaside Buffer (TLB) structure found in the IDT79R4600 and IDT79R4700 is replaced by a base-bounds mechanism. When an address is translated, its page number is first compared against the Bounds register. If the address is "in range," the base register is added to the virtual address to form the physical address.

The R4650 contains two sets of base-bounds registers, one set for instruction address translation (IBase and IBounds registers) and one for data (DBase and DBounds registers). An operating system can support task protection by writing appropriate values to these registers at context switch time.

Finally, to allow a mix of cache attributes in a single system, the R4650 also implements a *Cache Algorithm (CAlg)* register in CP0. This register allows the operating system to define the cache management attributes of different portions of the address space. By merely using appropriate virtual addresses memory can be treated as uncached, write-back, or write-through, with separate attributes for each of eight memory regions. In conjunction with the external system address decoder, software can then alias the same physical memory with different management algorithms, depending upon the data or program that is running.

Address Spaces

This section describes the virtual and physical address spaces and the manner in which virtual addresses are converted or "translated" into physical addresses by the base-bounds unit.

Virtual Address Space

The processor virtual address is 32-bits wide. The R4650 truncates addresses at 32 bits, and ignores the upper 32 bits of 64-bit registers during address translation.

Figure 4.1 illustrates how the R4650 translates a virtual address into a physical address.

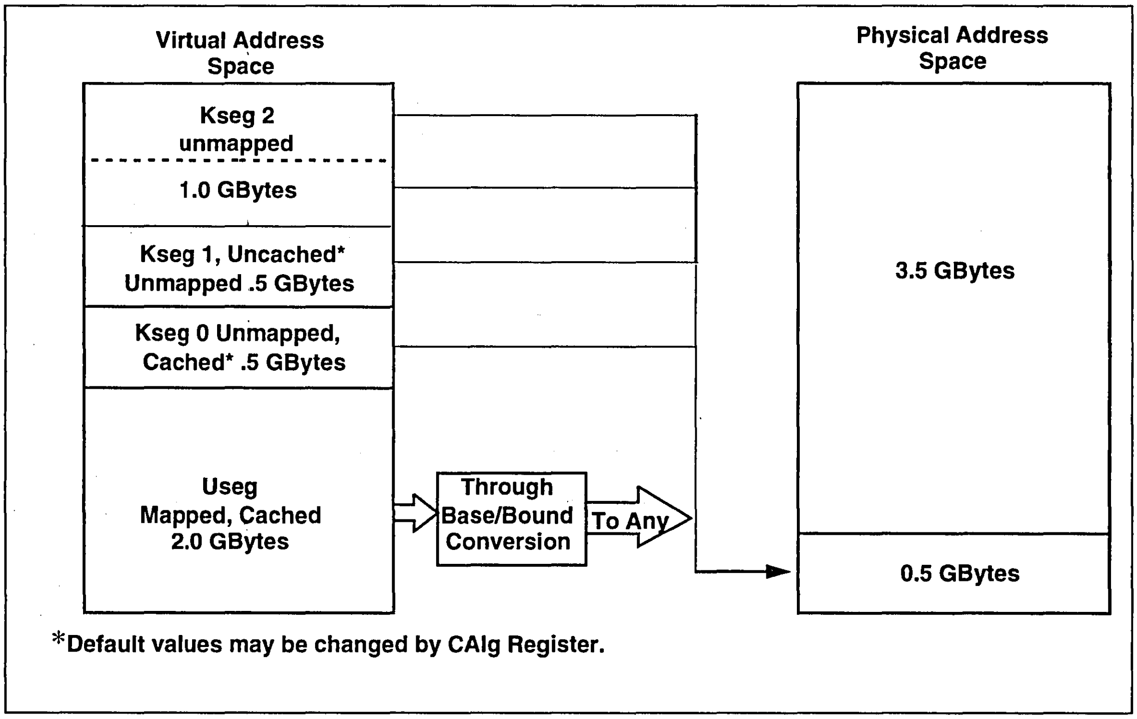

Virtual Address Space
Kseg 2 unmapped
1.0 GBytes
Kseg 1, Uncached* Unmapped .5 GBytes
Kseg 0 Unmapped, Cached* .5 GBytes
Useg Mapped, Cached 2.0 GBytes
Through Base/Bound Conversion
To Any
Physical Address Space
3.5 GBytes
0.5 GBytes
*Default values may be changed by CAlg Register.

Figure 4.1 Overview of R4650 Virtual-to-Physical Address Translation

Physical Address Space

Using a 32-bit address, the processor physical address space encompasses 4 Gigabytes. The section following describes the translation of a virtual address to a physical address.

Virtual-to-Physical Address Translation

The R4650 converts a virtual address to a physical address as shown in the following steps. The same procedure applies for either IBase/IBound or DBase/DBound, but the I and D registers are separate.

1. If bits 63:32 are generated by a load/store base+offset addition, they are discarded.
2. If VAddr(31) equals 1 and the CPU is in User mode, an address error exception is generated. However, if in Kernel mode, then the upper 3 bits of VAddr (bits 31:29) are removed and replaced by *000* to form the physical address.
3. If not a kernel address (VAddr(31)=0), then VAddr(30:12) is compared to Bound(30:12).
4. If VAddr is greater than the Bound address, then a Bound exception results.
5. Otherwise, the physical address equals (VAddr(31:12) + Base(31:12)), concatenated with VAddr(11:0). This is shown in Figure 4.2.

In parallel with the above operation, the cache access rules are obtained from the CAlg register, using VAddr(31:29) to select the appropriate CAlg field.

Virtual Address Base-Bounds

Figure 4.2 shows the virtual-to-physical-address translation of a 32-bit virtual address.

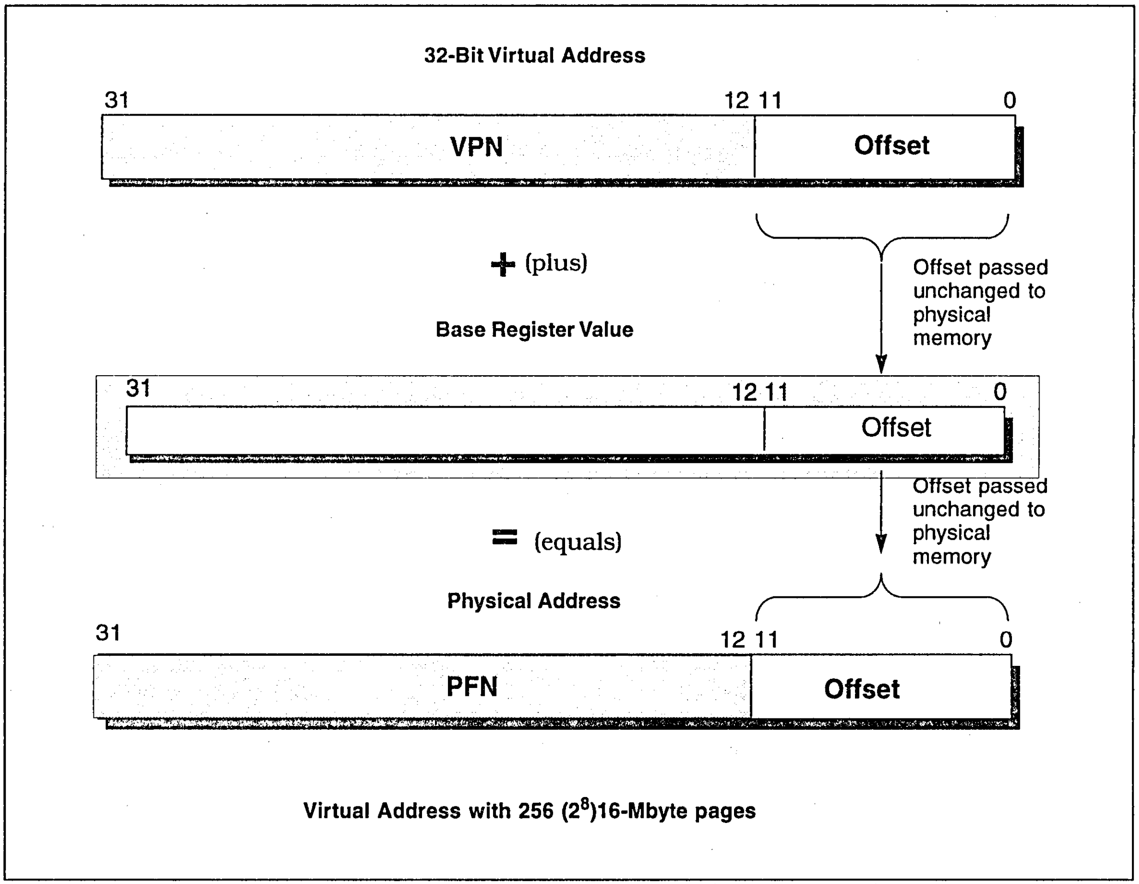

32-Bit Virtual Address
31
12 11
0
VPN
Offset
+ (plus)
Offset passed unchanged to physical memory
Base Register Value
31
12 11
0
Offset
Offset passed unchanged to physical memory
= (equals)
Physical Address
31
12 11
0
PFN
Offset
Virtual Address with 256 (2⁸)16-Mbyte pages

Figure 4.2 32-bit Virtual Address Translation

Operating Modes

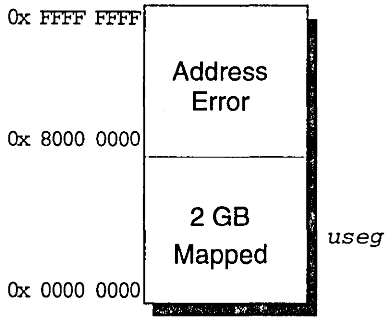
The processor has two operating modes:

- User mode
- Kernel mode

These modes are described in the following subsections.

User Mode Operations

In User mode, a single, uniform virtual address space—labelled User segment—is available; its size is 2 Gigabytes. Figure 4.3 shows the User mode virtual address space.



Note: Failure (i.e., bit 31 = 1) results in an Address Error exception.

Figure 4.3 User Mode Virtual Address Space

The User segment starts at address 0 and the current active user process resides in *useg*. The address translator identically maps all references to *useg* from both modes. The CAlg register controls cache accessibility.

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

- *UM = 1*
- *EXL = 0*
- *ERL = 0*

Table 4.1 lists the characteristics of the user mode segment *useg*.

Address Bit Values	Status Register Bit Values			Segment Name	Address Range	Segment Size
	UM	EXL	ERL			
32-bit	1	0	0	useg	0x0000 0000 through 0x7FFF FFFF	2 Gbyte (2^{31} bytes)

Table 4.1 User Mode Addressing

All valid User Mode virtual addresses have VAddr(31) cleared to 0; any attempt to reference an address with VAddr(31) set to 1 while in User mode causes an Address Error exception. The system maps all references to *useg* through the base-bound register, and bit settings within the CAlg register for the virtual address determine the cacheability of a reference.

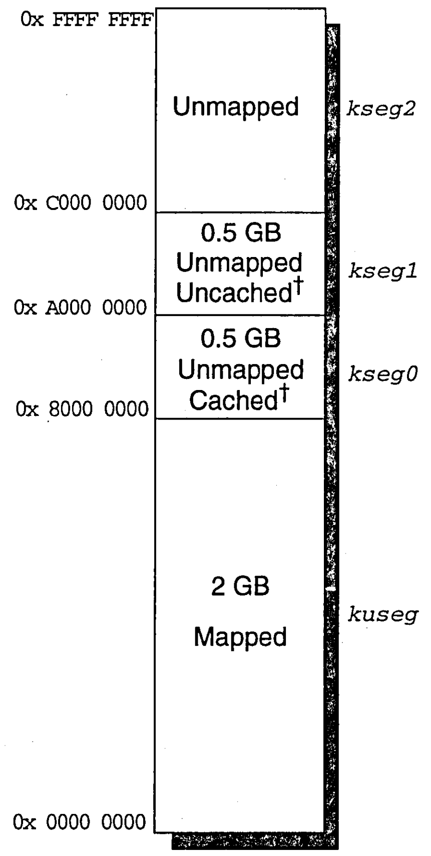
Kernel Mode Operations

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

- *UM = 0*
- *EXL* = 1
- *ERL* = 1

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

Kernel mode virtual address space is divided into regions differentiated by VAddr(31:29), as shown in Figure 4.4.



Note: †Default value; may be changed in CAlg register.

Figure 4.4 Kernel Mode Address Space

Table 4.2 lists the characteristics of the 32-bit kernel mode segments.

	Status Register Is One Of These Values					
Address Bit Values	**UM**	**EXL**	**ERL**	**Segment Name**	**Virtual Address Range**	**Segment Size**
A(31) = 0	UM = 0 or EXL = 1 or ERL =1			kuseg	0x0000 0000 through 0x7FFF FFFF	2 Gbytes (2^{31} bytes)
A(31:29) = 100_2				kseg0	0x8000 0000 through 0x9FFF FFFF	512 Mbytes (2^{29} bytes)
A(31:29) = 101_2				kseg1	0xA000 0000 through 0xBFFF FFFF	512 Mbytes (2^{29} bytes)
A(31:30) = 11_2				kseg2	0xC000 0000 through 0xFFFF FFFF	1 Gbyte (2^{32} bytes)

Table 4.2 u32-bit Kernel Mode Segments

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

In Kernel mode, when the most-significant bit of the virtual address, VAddr(31), is cleared, the 32-bit *kuseg* virtual address space is selected. It covers the full 2^{31} bytes (2 Gbytes) of the current user address space. The base-bounds mechanism will translate addresses in this region, and the CAlg register controls cacheability.

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

In Kernel mode, when the most-significant three bits of the virtual address are 100_2, 32-bit *kseg0* virtual address space is selected; it is the current 2^{29}-byte (512-Mbyte) kernel physical space.

References to *kseg0* are not mapped through the base-bounds registers. The physical address selected is defined by subtracting 0x8000 0000 from the virtual address (physical address = 000 | | VA[28:0]).

The *CAlg* register controls cacheability. At Reset kseg0 is cacheable and kseg1 is not.

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

In Kernel mode, when the most-significant three bits of the 32-bit virtual address are 101_2, 32-bit *kseg1* virtual address space is selected. It is the current 2^{29}-byte (512Mbyte) kernel physical space.

References to *kseg1* are not mapped through the base-bounds register. The physical address selected is defined by subtracting 0xA000 0000 from the virtual address (physical address = 000| | VA[28:0]).

By default, caches are disabled for accesses to these addresses, and physical memory (or memory-mapped I/O device registers) are accessed directly. However, CAlg allows this to be changed. At Reset kseg0 is cacheable and kseg1 is not.

32-bit Kernel Mode (*kseg2*)

In Kernel mode, when the most-significant two bits of the 32-bit virtual address are 11, the *kseg2* virtual address space is selected. The corresponding physical address is found by replacing the 3 most significant address bits with 000 (PAddr (31:0) = 000| | VAddr (28:0)). The *CAlg* register controls cacheability.

System Control Coprocessor

The System Control Coprocessor (CP0) is implemented as an integral part of the CPU, and supports memory management, address translation, exception handling, and other privileged operations. CP0 contains the base-bounds address in addition to the registers shown in Table 4.3. The following subsections describe how the processor uses the memory management-related registers.

Each CP0 register has a *register number*, which is a unique number that identifies it.

Number	Name	Function
0	IBase	Instruction address space base
1	IBound	Instruction address space bound
2	DBase	Data address space base
3	DBound	Data address space bound
4	-	not used
5	-	not used
6	-	not used
7	-	not used
8	BadVAddr	Virtual address on address exceptions
9	Count	Counts every other cycle
10	-	not used
11	Compare	Generate interrupt when Count = Compare
12	Status	Miscellaneous control/status
13	Cause	Exception/Interrupt information
14	EPC	Exception PC
15	PRId	Processor ID
16	Config	Device configuration info
17	CAlg	Cache attributes for the 8 512MB regions of the virtual address space
18	IWatch	Instruction breakpoint virtual address
19	DWatch	Data breakpoint virtual address
20	-	not used
21	-	not used
22	-	not used
23	-	not used
24	-	not used
25	-	not used
26	ECC	Error checking control
27	CacheErr	Error diagnostic info
28	TagLo	Cache addressing
29	-	not used
30	ErrorEPC	Cache Error exception PC
31	-	not used

Table 4.3 CP0 Registers

CP0 Registers

The following sections describe the CP0 registers (shown in Figure 4.5) that are assigned specifically as a software interface with memory management. The register number appears in parentheses after each register name in the following list:

- *IBase* (CP0 register 0)
- *IBound* (1)
- *DBase* (2)
- *DBound* (3)
- *PRId* (15)
- *CAlg* (17)
- *TagLo* (28)

IBase Register (0)

The *IBase* register provides the User Instruction address space Base address. Figure 4.5 shows the format of the *IBase* register; Table 4.4, which follows the figure, describes the *IBase* register fields.

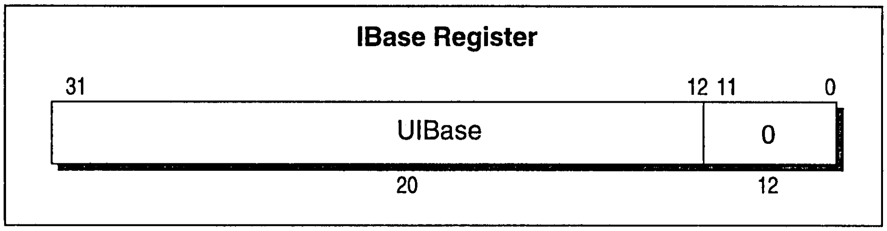


Figure 4.5 IBase Register

Field	Description
UIBase	Added to vAddr$_{31..12}$ for user space to get physical address
0	Reserved. Reads as 0, should be written as 0.

Table 4.4 IBase Register Field Descriptions

IBound Register (1)

The *IBound* register provides the User Instruction address space Bound address. Virtual addresses greater than this value cause address error exceptions. Figure 4.6 shows the format of the *IBound* register; Table 4.5, which follows the figure, describes the *IBound* register fields.

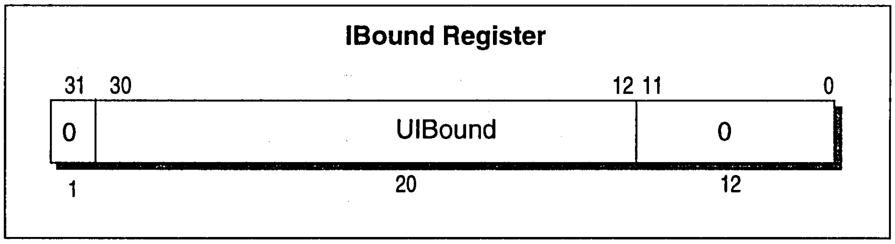


Figure 4.6 IBound Register

Field	Description
UIBound	Compared to vAddr$_{30..12}$ for user space to validate address
0	Reserved. Reads as 0, should be written as 0.

Table 4.5 IBound Register Field Descriptions

DBase Register (2)

The *DBase* register provides the User Data address space Base address. Figure 4.7 shows the format of the *DBase* register; Table 4.6, which follows the figure, describes the *DBase* register fields.

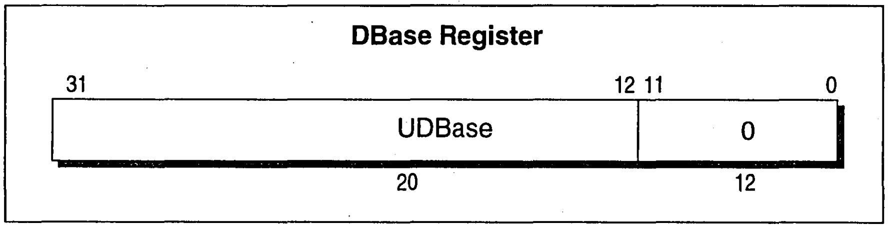


Figure 4.7 DBase Register

Field	Description
UDBase	Added to $vAddr_{31..12}$ for user space to get physical address
0	Reserved. Reads as 0, should be written as 0.

Table 4.6 DBase Register Field Descriptions

DBound Register (3)

The *DBound* register provides the User Data address space Bound. Figure 4.8 shows the format of the *DBound* register; Table 4.7, which follows the figure, describes the *DBound* register fields.

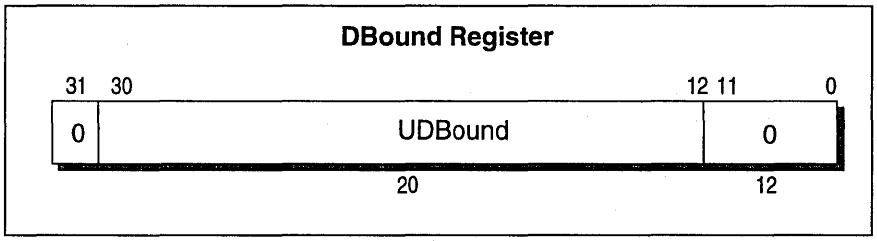


Figure 4.8 DBound Register

Field	Description
UDBound	Compared to $vAddr_{31..12}$ for user space to validate address
0	Reserved. Reads as 0, should be written as 0.

Table 4.7 DBound Register Field Descriptions

Processor Revision Identifier (PRId) Register (15)

The 32-bit, read-only *Processor Revision Identifier* (*PRId*) register contains information identifying the implementation and revision level of the CPU and CP0. Figure 4.9 shows the format of the *PRId* register; Table 4.8 describes the *PRId* register fields.

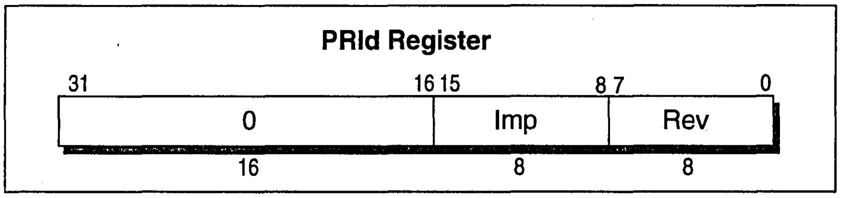


Figure 4.9 Processor Revision Identifier Register Format

Field	Description
Imp	Implementation number R4650 Imp = 0x22
Rev	Revision number
0	Reserved. Returns zeroes when read.

Table 4.8 PRId Register Fields

The low-order byte (bits 7:0) of the *PRId* register is interpreted as a revision number, and the high-order byte (bits 15:8) is interpreted as an implementation number. The implementation number of the R4650 processor is 0x22. The content of the high-order halfword (bits 31:16) of the register are reserved.

The revision number is stored as a value in the form *y.x*, where *y* is a major revision number in bits 7:4 and *x* is a minor revision number in bits 3:0.

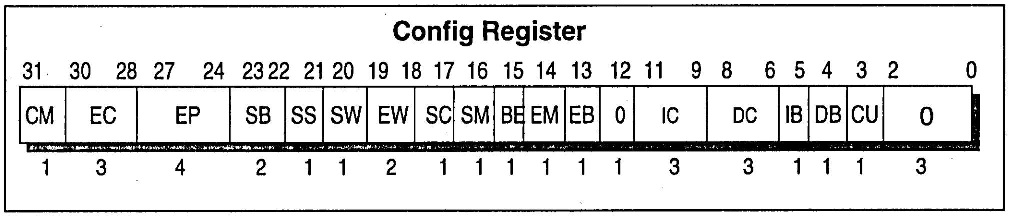
The revision number can distinguish some chip revisions, however there is no guarantee that changes to the chip will necessarily be reflected in the *PRId* register, or that changes to the revision number necessarily reflect real chip changes. For this reason, these values are not listed and software should not rely on the revision number in the *PRId* register to characterize the chip. Certain attributes, such as cache size, are independent of implementation number.

Config Register (16)

The *Config* register specifies various configuration options selected on R4650 processors; Table 4.9 lists these options.

Some configuration options, as defined by *Config* bits 31:3, are set by the hardware during reset and are included in the *Config* register as read-only status bits for the software to access.

Figure 4.10 shows the format of the *Config* register; Table 4.9, which follows the figure, describes the *Config* register fields.



Config Register

31	30 28	27 24	23 22	21	20	19 18	17	16	15	14	13	12	11 9	8 6	5	4	3	2 0
CM	EC	EP	SB	SS	SW	EW	SC	SM	BE	EM	EB	0	IC	DC	IB	DB	CU	0
1	3	4	2	1	1	2	1	1	1	1	1	1	3	3	1	1	1	3

Figure 4.10 Config Register Format

Field	Description
EC	Pipeline clock ratio: 0 → processor input clock frequency multiplied by 2 1 → processor input clock frequency multiplied by 3 2 → processor input clock frequency multiplied by 4 3 → processor input clock frequency multiplied by 5 4 → processor input clock frequency multiplied by 6 5 → processor input clock frequency multiplied by 7 6 → processor input clock frequency multiplied by 8 7 Reserved
EP (EW=1)	Write-back data rate: 0 → WWWWWWWWW 1 word every cycle 1 → WWxWWxWWxWW 2 words every 3 cycles 2 → WWxxWWxxWWxxWWxx 2 words every 4 cycles 3 → WxWxWxWxWxWxWxWx 2 words every 4 cycles 4 → WWxxxWWxxxWWxxxWWxxx 2 words every 5 cycles 5 → WWxxxxWWxxxxWWxxxxWWxxxx 2 words every 6 cycles 6 → WxxWxxWxxWxxWxxWxxWxxWxx 2 words every 6 cycles 7 → WWxxxxxWWxxxxxWWxxxxxWWxxxx 2 words every 7 cycles 8 → WxxxWxxxWxxxWxxxWxxxWxxxWxxxWxxx 2 words every 8 cycles
EP (EW=0)	Write-back data rate: 0 → DDDD 1 double word every cycle 1 → DDxDDx 2 double words every 3 cycles 2 → DDxxDDxx 2 double words every 4 cycles 3 → DxDxDxDx 2 double words every 4 cycles 4 → DDxxxDDxxx 2 double words every 5 cycles 5 → DDxxxxDDxxxx 2 double words every 6 cycles 6 → DxxDxxDxxDxx 2 double words every 6 cycles 7 → DDxxxxxDDxxxx 2 double words every 7 cycles 8 → DxxxDxxxDxxxDxxx 2 double words every 8 cycles
EW	SysAD bus size; 0 → 64 bits, 1 → 32 bits (from serial mode bits)
BE	BigEndianMem 0 → Little Endian 1 → Big Endian
IC	Primary I-cache Size (I-cache size = 2^{12+IC} bytes). In the R4650 processor this is set to 8 Kbytes (IC = 001).
DC	Primary D-cache Size (D-cache size = 2^{12+DC} bytes). In the R4650 processor this is set to 8 Kbytes (DC = 001).
IB	Primary I-cache line size 1 → 32 bytes (8 Words)
DB	Primary D-cache line size 1 → 32 bytes (8 Words)
Others	Reserved. Returns indicated values when read.

Table 4.9 Config Register Fields

CAlg Register (17)

The *CAlg* register is a read-write register that specifies the cache algorithm for each 512MB region of the virtual address space.

CAlg is initialized to 0x22233333 on Reset. Bits 31, 27, 23, 19, 15, 11, 7, and 3 are not implemented, and are reserved for future use. They read as zero and are ignored on write.

Figure 4.11 shows the format of the *CAlg* register; Table 4.10, which follows the figure, describes the *CAlg* register fields.

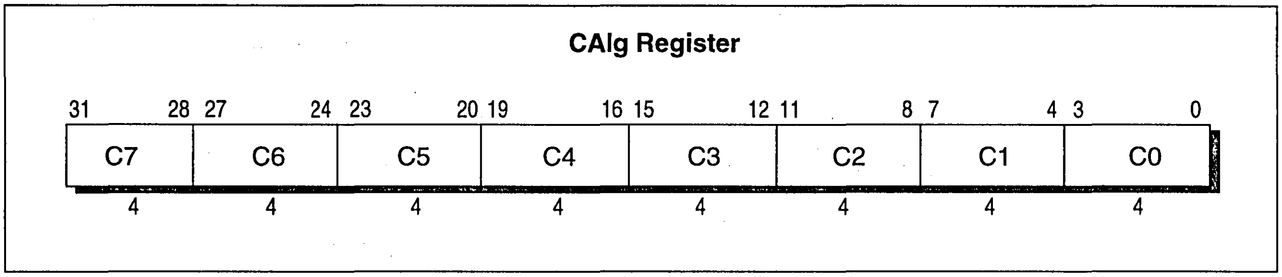


Figure 4.11 CAlg Register

The Cache algorithms are as follows:

0	Cached, non-coherent, write-through, no write-allocate
1	Cached, non-coherent, write-through, write-allocate
2	Uncached
3	Cached, non-coherent, write-back, write-allocate
4-15	Reserved

Field	Description
C0	Cache algorithm for 0x00000000 to 0x1FFFFFFF (part of useg/kuseg)
C1	Cache algorithm for 0x20000000 to 0x3FFFFFFF (part of useg/kuseg)
C2	Cache algorithm for 0x40000000 to 0x5FFFFFFF (part of useg/kuseg)
C3	Cache algorithm for 0x60000000 to 0x7FFFFFFF (part of useg/kuseg)
C4	Cache algorithm for 0x80000000 to 0x9FFFFFFF (k seg0)
C5	Cache algorithm for 0xA00000000 to 0xBFFFFFFF (k seg 1)
C6	Cache algorithm for 0xC0000000 to 0xDFFFFFFF (part of kseg2)
C7	Cache algorithm for 0xE0000000 to 0xFFFFFFFF (part of kseg2)

Table 4.10 CAlg Register Field Descriptions

Cache Tag Registers [TagLo (28)

The *TagLo* register is a 32-bit read/write register that holds the primary cache tag and parity during cache initialization, cache diagnostics, and cache error processing. The *Tag* register is written by the CACHE and MTC0 instructions.

The *P* field is ignored on Index Store Tag operations. Parity is computed by the store operation.

Figure 4.12 shows the register format for primary cache operations. Table 4.11 lists the field definitions of the *TagLo* register.

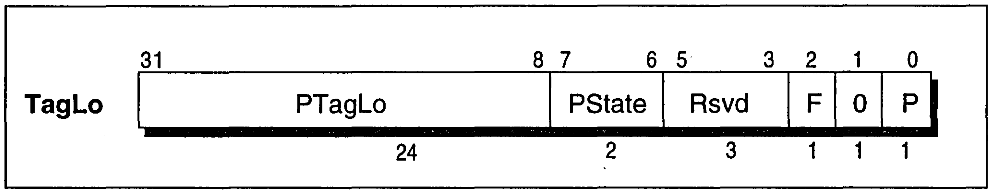


Figure 4.12 TagLo Register (P-cache) Format

Field	Description
PTagLo	Specifies the physical address bits 35:12
PState	Specifies the primary cache state
P	Specifies the primary tag even parity bit
F	The FIFO bit (used internally to implement FIFO refill of the cache)
Rsvd	Reserved. Must be written as zeroes.
0	Reserved. Must be written as zeroes; returns zeroes when read

Table 4.11 Cache Tag Register Fields

Virtual-to-Physical Address Translation Process

Figure 4.13 illustrates the Base-Bounds address translation process.

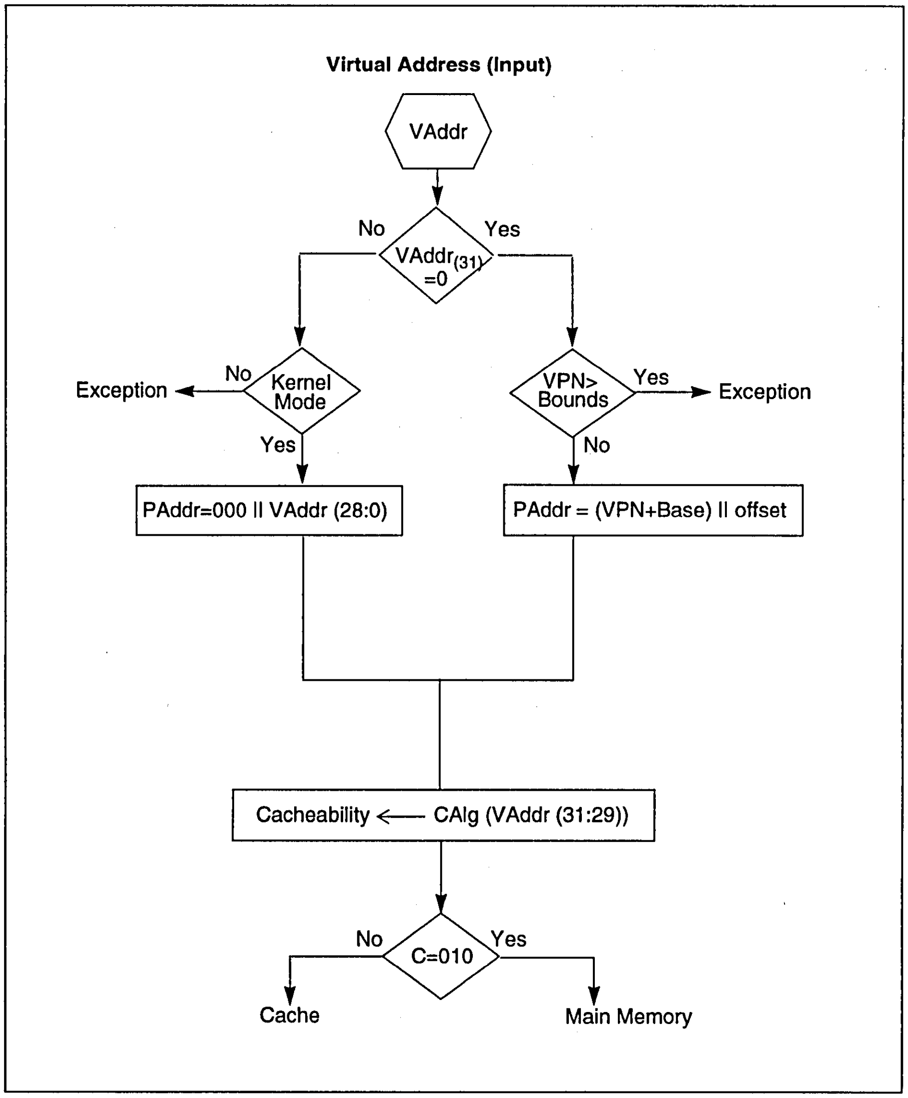


Figure 4.13 Base-Bounds Address Translation

CPU Exception Processing Chapter 5

This chapter describes the CPU exception processing, including a discussion of the format and use of each CPU exception register.

The chapter concludes with a description of each exception's cause, together with the manner in which the CPU processes and services these exceptions. For information about Floating-Point Unit exceptions, refer to Chapter 7.

How Exception Processing Works

The processor receives exceptions from a number of sources, including address translation errors, arithmetic overflows, I/O interrupts, and system calls. When the CPU detects one of these exceptions, the normal sequence of instruction execution is suspended and the processor enters Kernel mode. Refer to Chapter 4 for a description of system operating modes.

The processor then disables interrupts and forces execution of a software exception processor (called a *handler*) located at a fixed address. The handler may save the context of the processor, including the contents of the program counter, the current operating mode (User or Kernel), and the status of the interrupts (enabled or disabled). This context would be saved so it can be restored when the exception has been serviced.

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

The registers described later in the chapter assist in this exception processing by retaining address, cause and status information.

For a description of the exception handling process, refer to the flowcharts at the end of this chapter.

The Exception Processing Registers

This section describes the CP0 registers that are used in exception processing. Table 5.1 on page 5-2 lists these registers, along with their number. Each register has a unique identification number called a *register number*. For example, the *ECC* register is register number 26. The remaining CP0 registers are used in memory management, as described in Chapter 4.

Software examines the CP0 registers during exception processing to determine the cause of the exception and the state of the CPU at the time the exception occurred. Table 5.1 lists the register used in exception processing. A description of each register follows the table.

Register Name	Reg. No.
IWatch	18
DWatch	19
BadVAddr (Bad Virtual Address)	8
Count	9
Compare register	11
Status	12
Cause	13
EPC (Exception Program Counter)	14
ECC	26
CacheErr (Cache Error and Status)	27
ErrorEPC (Error Exception Program Counter)	30

Table 5.1 CP0 Exception Processing Registers

IWatch Register (18)

The *IWatch* register is a read/write register that specifies an Instruction virtual address that causes a Watch exception. When $VADDR_{31..2}$ of an instruction fetch 'matches IVAddr of this register, and the I bit is set, a Watch exception is taken. Matches that occur when EXL = 1 or ERL = 1 do not take the exception immediately, but are instead postponed until both EXL and ERL are cleared. The priority of IWatch exceptions is just below Instruction Address Error exceptions. Figure 5.1 shows the format of the *IWatch* register; Table 5.2, which follows the figure, describes the *IWatch* register fields.

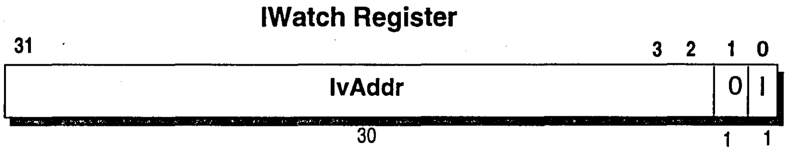


Figure 5.1 IWatch Register Format

Field	Description
IvAddr	Instruction virtual address that causes a watch exception (bits 31:2).
I	0 ---> IWatch disabled, 1 ---> IWatch enabled.
0	reserved for future use.
Note: IWatch.I is cleared on Reset.	

Table 5.2 IWatch Register Fields

DWatch Register (19)

DWatch is a read/write register that specifies a Data virtual address that causes a Watch exception. Data Watch exception is taken when VAddr $_{31..3}$ of a load matches DVAddr of this register and the R bit is set, or when VAddr $_{31..3}$ of a store matches DvAddr of this register and the W bit is set. Matches that occur when EXL = 1 or ERL = 1 do not take the exception immediately, but are instead postponed until both EXL and ERL are cleared. The priority of DWatch exceptions is just below Data Address Error exceptions. DWatch exceptions do not occur on CACHE ops. Figure 5.2 shows the format of the *DWatch* register; Table 5.3, which follows the figure, describes the *DWatch* register fields.

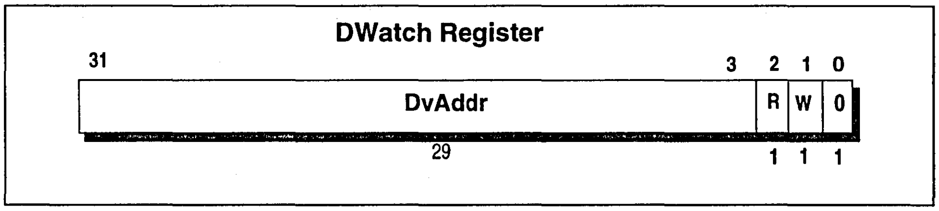


Figure 5.2 DWatch Register Format

Field	Description
DvAddr	Data virtual address that causes a watch exception.
R	0 ---> DWatch disabled for loads, 1 ---> DWatch enabled for loads.
W	0 ---> DWatch disabled for stores, 1---> DWatch enabled for stores.
0	reserved for future use.
Note: DWatch.R and DWatch.W are cleared on Reset.	

Table 5.3 DWatch Register Fields

Bad Virtual Address Register (BadVAddr) (8)

The Bad Virtual Address register (*BadVAddr*) is a read-only register that displays the most recent virtual address that caused one of the exceptions in the following list. The processor does not write to the *BadVAddr* register when the *EXL* bit in the *Status* register is set to a 1.

- Address Error (e.g., unaligned access)
- Bounds
- Virtual Coherency Data Access
- Virtual Coherency Instruction Fetch

Figure 5.3 shows the format of the *BadVAddr* register. The *BadVAddr* register does not save any information for bus errors, since bus errors are not addressing errors.

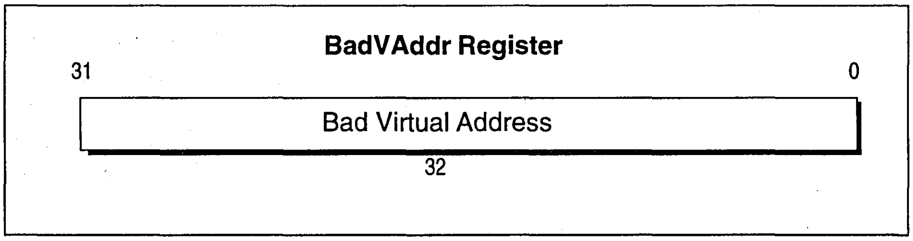


Figure 5.3 BadVAddr Register Format

Count Register (9)

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

This register can be read or written. It can be written for diagnostic purposes or system initialization; for example, to synchronize processors.

Figure 5.4 shows the format of the *Count* register.

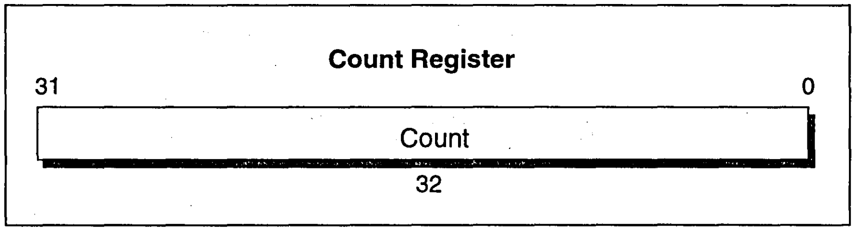


Figure 5.4 Count Register Format

Compare Register (11)

The *Compare* register acts as a timer, and (see also the *Count* register) maintains a stable value that does not change on its own. When the value of the *Count* register equals the value of the *Compare* register, interrupt bit *IP(7)* in the *Cause* register is set. If the timer interrupt was enabled at boot time, an interrupt will occur as soon as the interrupt is enabled. Writing a value to the *Compare* register, as a side effect, clears the timer interrupt.

For diagnostic purposes, the *Compare* register is a read/write register. However, in normal use the *Compare* register is write-only. Figure 5.5 shows the format of the *Compare* register.

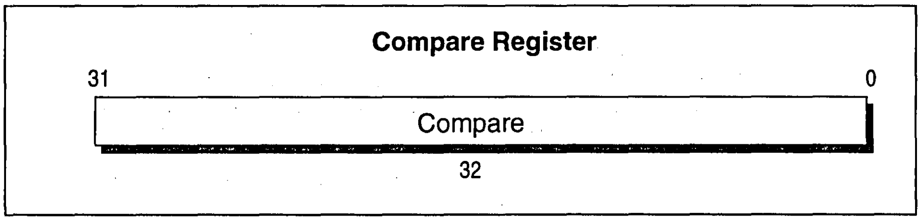


Figure 5.5 Compare Register Format

Status Register (12)

The *Status* register (SR) is a read/write register that contains the operating mode, interrupt enabling, and the diagnostic states of the processor. The following list describes the more important *Status* register fields.

Figure 5.6 shows the format of the *Status* register. Table 5.4, which follows the figure, describes the *Status* register fields.

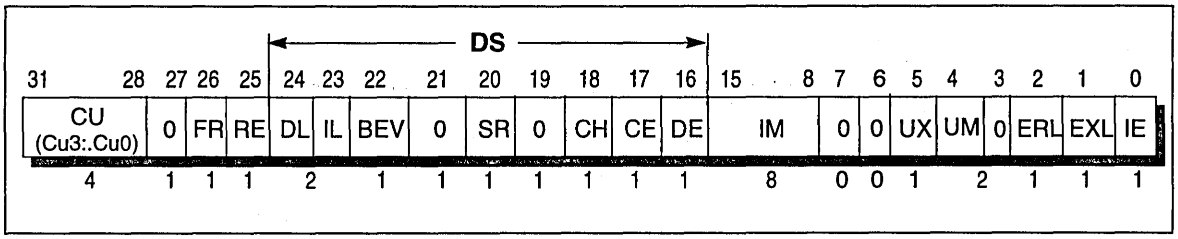


Figure 5.6 Status Register

Field	Description
CU	Controls the usability of each of the four coprocessor unit numbers. CP0 is always usable when in Kernel mode, regardless of the setting of the *CU_0* bit. 1 → usable 0 → unusable **Note:** In the MIPS 3 ISA, CP3 is no longer defined as a valid coprocessor unit.
FR	Enables additional floating-point registers 0 → 16 registers 1 → 32 registers
RE	*Reverse-Endian* bit, valid in User mode.
DL	Data cache lock, a new bit in R4650. Does not prevent refills into set A when set A is invalid. Does not inhibit update of the D-cache on store operations. 0 → normal operation 1→ refill into set A disabled
IL	Instruction cache lock, a new bit in R4650. Does not prevent refills into set A when set A is invalid. 0 → normal operation 1→ refill into set A disabled
BEV	Controls the location of exception vectors. 0 → normal 1→ bootstrap
SR	1→ Indicates a soft reset or NMI has occurred.
CH	Hit (tag match and valid state) or miss indication for last CACHE Hit Invalidate, Hit Write Back Invalidate, Hit Write Back, or Hit Set Virtual for a primary cache. 0 → miss 1 → hit
CE	Contents of the ECC register set or modify the check bits of the caches when CE = 1; see description of the *ECC* register.
DE	Specifies that cache parity errors cannot cause exceptions. 0 → parity remains enabled 1 → disables parity
0	Reserved. Read as 0, ignored on writes.
IM	*Interrupt Mask*: controls the enabling of each of the external, internal, and software interrupts. An interrupt is taken if interrupts are enabled, and the corresponding bits are set in both the *Interrupt Mask* field of the *Status* register and the *Interrupt Pending* field of the *Cause* register. IM[7:2] correspond to interrupts Int[5:0] and IM[1:0] to the software interrupts. 0→ disabled 1→ enabled
UX	Controls whether the 64-bit MIPS-3 instructions can be used in user mode. 0 → 32-bit only 1 → 64-bit enabled
UM	User Mode bit, a new bit in R4650. 0 → User 1 → Kernel (Simplification of KSU, remains subject to EXL and ERL, as on R4xxx.
ERL	Error Level 0 → normal 1 → error
EXL	Exception Level 0 → normal 1 → exception **Note:** When going from 0 to 1, IE should be disabled (0) first. This would be done when preparing to return from the exception handler, such as before executing the ERET instruction.
IE	Interrupt Enable 0 → disable interrupts 1 → enables interrupts

Table 5.4 Status Register Fields

Status Register Modes and Access States

Fields of the *Status* register set the modes and access states described in the sections that follow.

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

- *IE* = 1
- *EXL* = 0
- *ERL* = 0

If these conditions are met, the settings of the *IM* bits identify the interrupt.

Note: Setting the IE bit may be delayed by up to 3 cycles. If performing nested interrupts, re-enable the IE bit first.

Operating Modes: The following CPU *Status* register bit settings are required for User, Kernel, and Supervisor modes (see Chapter 4 for more information about operating modes).

- The processor is in User mode when all of these bits are set as follows:
  - *UM* = 0
  - *EXL* = 0
  - *ERL* = 0
- The processor is in Kernel mode when any of these bits are set as follows:
  - UM = 1
  - *EXL* = 1
  - *ERL* = 1

32-bit Virtual Addressing: The R4650 only supports 32-bit virtual addresses. It ignores bits 63:32 of memory addresses.

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

User Address Space Accesses: Access to the user address space is allowed in either Kernel or User mode.

Status Register Reset

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

Cause Register (13)

The 32-bit read/write *Cause* register describes the cause of the most recent exception.

Figure 5.7 shows the fields of this register; Table 5.5, which follows the figure, describes the *Cause* register fields. A 5-bit exception code (*ExcCode*) indicates the cause of the most recent exception, as listed in Table 5.6 on page 5-8.

All bits in the *Cause* register, with the exception of the *IP(1:0)* bits, are read-only. IP(1:0) bits are used for software interrupts. The *Cause.IV* bit is set to zero by a Reset.

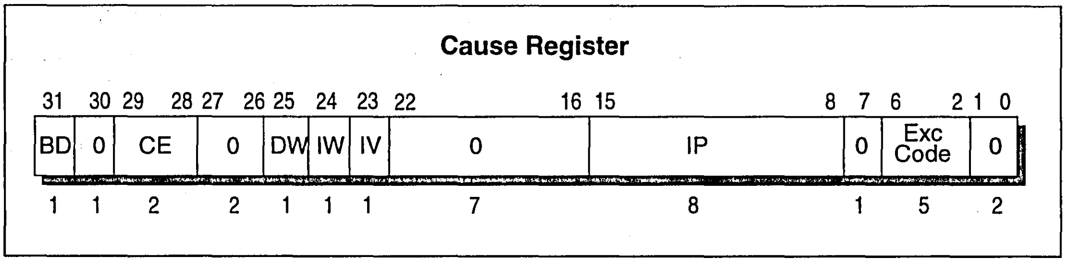


Figure 5.7 Cause Register Format

Field	Description
BD	Indicates whether the last exception taken occurred in a branch delay slot. 1 → delay slot 0 → normal
0	Reserved. Currently read as 0 and must be written as '0'.
CE	Coprocessor unit number referenced when a Coprocessor Unusable exception is taken..
DW	On a Watch exception, indicates that the DWatch register matched. On other exceptions this field is undefined.
IW	On a Watch exception, indicates that the IWatch register matched. On other exceptions this field is undefined.
IV	Enables the new dedicated interrupt vector. 1 → interrupts use new exception vector (200) 0 → interrupts use common exception vector (180)
IP	Indicates an interrupt is pending. 1 → interrupt pending 0 → no interrupt
ExcCode	Exception code field (see Table 5.6 on page 5-8)

Table 5.5 Cause Register Fields

Exception Code Value	Mnemonic	Description
0	Int	Interrupt
1	—	Reserved
2	IBound	Instruction bound exception (replaces TLB exception on load)
3	DBound	Data bound exception (replaces TLB exception on store)
4	AdEL	Address error exception (load or instruction fetch)
5	AdES	Address error exception (store)
6	IBE	Bus error exception (instruction fetch)
7	DBE	Bus error exception (data reference: load or store)
8	Sys	Syscall exception
9	Bp	Breakpoint exception
10	RI	Reserved instruction exception
11	CpU	Coprocessor Unusable exception
12	Ov	Arithmetic Overflow exception
13	Tr	Trap exception
14	—	Reserved
15	FPE	Floating-Point exception
16–22	—	Reserved
23	Watch	Watch exception
24-31	—	Reserved

Table 5.6 Cause Register ExcCode Field

Exception Program Counter (EPC) Register (14)

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

For synchronous exceptions, the *EPC* register contains either:

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

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

Figure 5.8 shows the format of the *EPC* register.

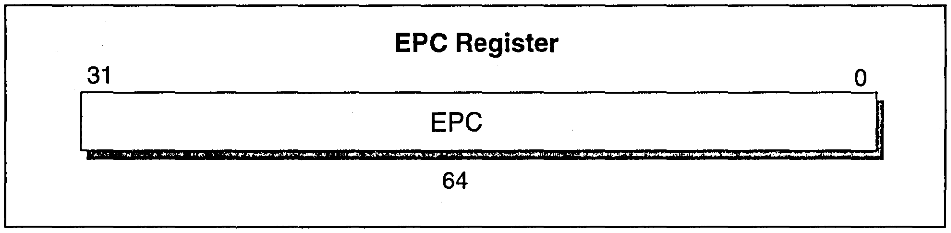


Figure 5.8 EPC Register Format

Error Checking and Correcting (ECC) Register (26)

The 8-bit *Error Checking and Correcting* (*ECC*) register reads or writes primary-cache data parity bits for cache initialization, cache diagnostics, or cache error processing. Tag parity is loaded from and stored to the *TagLo* register.

The *ECC* register is loaded by the Index Load Tag CACHE operation. Content of the ECC register are:

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

To force a cache parity value use the *Status CE* bit and the ECC register.

Figure 5.9 shows the format of the *ECC* register; Table 5.7, which follows the figure, describes the register fields.

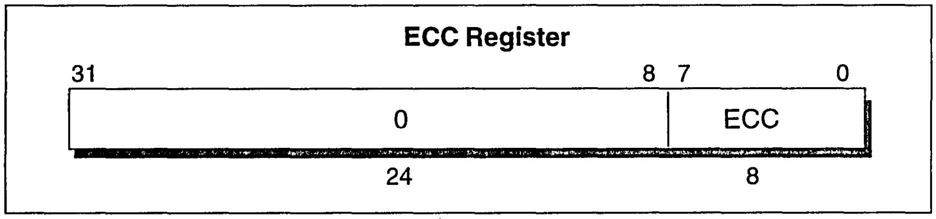


Figure 5.9 ECC Register Format

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

Table 5.7 ECC Register Fields

Cache Error (CacheErr) Register (27)

The 32-bit read-only *CacheErr* register processes parity errors in the primary cache. Parity errors cannot be corrected.

The *CacheErr* register holds cache index and status bits that indicate the source and nature of the error. It is loaded when a Cache Error exception is asserted. When a read response returns with bad parity, this exception is also asserted.

Figure 5.10 shows the format of the *CacheErr* register. Table 5.8, which follows the figure, describes the *CacheErr* register fields.

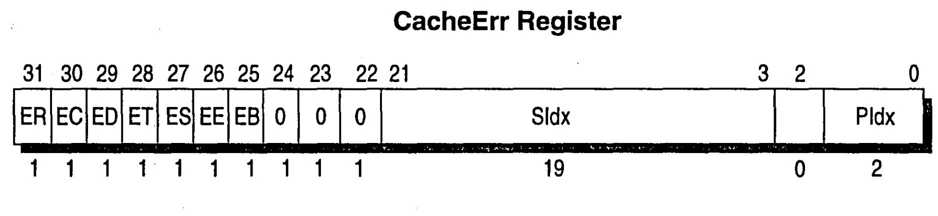


Figure 5.10 CacheErr Register Format

Field	Description
ER	Type of reference 0 → instruction 1 → data
EC	Cache level of the error 0 → primary 1 → reserved
ED	Indicates if a data field error occurred 0 → no error 1 → error
ET	Indicates if a tag field error occurred 0 → no error 1 → error
ES	Indicates the error occurred accessing processor-managed resources, in response to an external request. 0 → internal reference 1 → external reference Since the R4650 doesn't have any external events that would look in a cache (which is the only processor-managed resource), this bit would not be set under normal operating conditions.
EE	Set if the error occurred on the SysAD bus. Taking a cache error exception sets/clears this bit.
EB	Set if a data error occurred in addition to the instruction error (indicated by the remainder of the bits). If so, this requires flushing the data cache after fixing the instruction error.
SIdx	Physical address 21:3 of the reference that encountered the error.
PIdx	Virtual address 13:12 of the double word in error. To be used with SIdx to construct a virtual index for the primary caches. Only the lower two bits (bits 1 and 0) are vAddr; the high bit (bit 2) is zero.
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 5.8 CacheErr Register Fields

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

The *ErrorEPC* register is similar to the *EPC* register, except that *ErrorEPC* is used on parity error exceptions. It is also used to store the program counter (PC) on Reset, Soft Reset, and nonmaskable interrupt (NMI) exceptions.

The read/write *ErrorEPC* register contains the virtual address at which instruction processing can resume after servicing an error. This address can be either:

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

There is no branch delay slot indication for the *ErrorEPC* register.

Figure 5.11 shows the format of the *ErrorEPC* register.

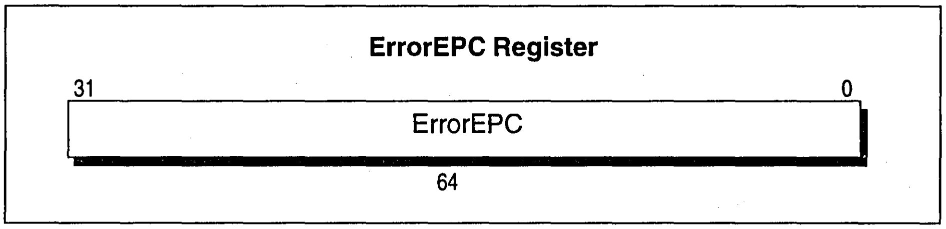


Figure 5.11 ErrorEPC Register Format

Processor Exceptions

This section describes the processor exceptions, their causes, processing by the hardware, and servicing by a handler (software). Exception types are described in the next section.

Processor Exception Examples

This section gives sample exception handler operations for the following exception types:

- reset
- soft reset
- nonmaskable interrupt (NMI)
- cache error
- interrupts
- remaining processor exceptions

When the *EXL* bit in the *Status* register is 0, either User or Supervisor operating mode is specified by the *KSU* bits in the *Status* register. When the *EXL* bit or the *ERL* bit is set to 1, the processor is in Kernel mode.

When the processor takes an exception, the *EXL* bit is set to 1, which means the system is in Kernel mode. After saving the appropriate state, the exception handler typically resets the *EXL* bit back to 0. When restoring the state and restarting, the handler sets the *EXL* bit back to 1. Returning from an exception also resets the *EXL* bit to 0 (see the ERET instruction in Appendix A).

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

Reset Exception Process Example

Figure 5.12 shows the Reset exception process.

```
T: undefined
   Config <- 0 || EC || EP || 00000000 || BE || 110 || 001 || 001 || 1 || 1 || 0 || undefined$^3$
   ErrorEPC ← PC
   SR ← $SR_{31:23}$ || 1 || 0 || 0 || $SR_{19:3}$ || 1 || $SR_{1:0}$
   PC ← 0x BFC0 0000
```

Figure 5.12 Reset Exception Processing

Cache Error Exception Process Example

Figure 5.13 shows the Cache Error exception process.

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

Figure 5.13 Cache Error Exception Processing

Soft Reset and NMI Exception Process Example

Figure 5.14 shows the Soft Reset and NMI exception process.

```
T: ErrorEPC ← PC
   SR ← $SR_{31:23}$ || 1 || 0 || 1 || $SR_{19:3}$ || 1 || $SR_{1:0}$
   PC ← 0x BFC0 0000
```

Figure 5.14 Soft Reset and NMI Exception Processing

Interrupt Exception Process Example

Figure 5.15 shows the process used for exceptions other than Reset, Soft Reset, NMI, and Cache Error.

```
T: $Cause \leftarrow BD \| 0 \| CE \| 0^{12} \| Cause_{15:8} \| 0 \| ExcCode \| 0^2$
   if $SR_1 = 0$ then        /* system in User or Supervisor mode with no current exception */
     $EPC \leftarrow PC$
   endif
   $SR \leftarrow SR_{31:2} \| 1 \| SR0$
   if Cause.IV then
     vector=200
   else
     vector=180
   if $SR_{22} = 1$ then        /* What is the BEV bit setting */
     $PC \leftarrow$ 0x BFC0 0200 + vector                /* access to uncached space */
   else
     $PC \leftarrow$ 0x 8000 0000 + vector                /* access to cached space */
   endif
```

Figure 5.15 Interrupt Exception Processing

General Exception Process Example

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

```
T: $Cause \leftarrow BD \| 0 \| CE \| 0^{12} \| Cause_{15:8} \| 0 \| ExcCode \| 0^2$
   if $SR_1 = 0$ then        /* system in User or Supervisor mode with no current exception */
     $EPC \leftarrow PC$
   endif
   $SR \leftarrow SR_{31:2} \| 1 \| SR0$
   if $SR_{22} = 1$ then        /* What is the BEV bit setting */
     $PC \leftarrow$ 0x BFC0 0200 + vector                /* access to uncached space */
   else
     $PC \leftarrow$ 0x 8000 0000 + vector                /* access to cached space */
   endif
```

Figure 5.16 General Exception Processing (Except Reset, Soft Reset, NMI, and Cache Error)

Processor Exception Vector Locations

The Reset, Soft Reset, and NMI exceptions are always vectored to location 0xBFC00000 (virtual address), corresponding to *kseg0.*

Addresses for all other exceptions are a combination of a *vector offset* and a *base address*. The base address is determined by the *BEV* bit of the *Status* register, as shown in Table 5.9.

BEV	R4650 Processor Vector Base	Cache Error Base
0	0x 8000 0000	0x A000 0000
1	0x BFC0 0200	0x BFC0 0200

Table 5.9 Exception Vector Base Addresses

Table 5.10 shows the vector offset that is added to the base address to create the exception address.

As shown in Figure 5.13, when *BEV* = 0, the vector base for the Cache Error exception changes from *kseg0* (0x80000000) to *kseg1* (0xA0000000). When *BEV*=1, the vector base for the Cache Error exception is 0xBFC00200. This is an uncached and unmapped space, allowing the exception to bypass the cache and TLB.

Exception	R4650 Processor Vector Offset
Cache Error	0x100
Interrupt[†]	0x200
Others	0x180
Note: [†]If cause .IV=1, otherwise interrupts use general vector offset.	

Table 5.10 Exception Vector Offsets

Priority of Exceptions

The remainder of this chapter describes exceptions in the order of their priority, as shown in Table 5.11. While more than one exception can occur for a single instruction, only the exception with the highest priority is reported.

Priority	Exception	Priority	Exception
1	Reset *(highest priority)*	9	Integer overflow, Trap, System Call, Breakpoint, Reserved Instruction, Coprocessor Unusable, or Floating-Point Exception
2	Soft Reset	10	Bound error — Data access
3	Nonmaskable Interrupt (NMI)	11	Address Error — Data access
4	Bound — Instruction fetch	12	Cache Error — Data access
5	Address — Instruction fetch	13	Watch — Data access
6	Watch — Instruction fetch	14	Bus error — Data access
7	Cache error — Instruction fetch	15	Interrupt *(lowest priority)*
8	Bus error — Instruction fetch		

Table 5.11 Exception Priority Order

Processor Exception Descriptions

In general, the exceptions described in the following sections are handled ("processed") by hardware, then serviced by software.

Reset Exception

This section explains the Reset exception.

Cause

The Reset exception occurs when the **ColdReset*** signal[1] is asserted and then deasserted. This exception is not maskable.

Processing

The CPU provides the special exception vector 0xBFC0 0000 for this exception.

The Reset vector resides in unmapped and uncached CPU address space, so the hardware does not need to initialize the cache to process this exception. In addition, 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 as follows:

- In the *Status* register, *SR* is cleared to 0, and *ERL* and *BEV* are set to 1. All other bits are undefined.
- Some of the *Config* Register bits are initialized from the boot-time mode stream.
- Cause register IV = 0.
- CAl_g = 0x22233333
- IWatch.I = 0
- DWatch.R=0, DWatch.W = 0

Reset exception processing is shown in Figure 5.12 on page 5-12.

Servicing

The Reset exception is serviced by:

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

Soft Reset Exception

This section explains the 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.

Processing

The Reset exception vector is used for this exception, located within unmapped and uncached address space so that the cache need not be initialized to process this exception. When a Soft Reset occurs, the *SR* bit of the *Status* register is set to distinguish this exception from a Reset exception.

[1] In the following sections (and throughout this manual) a signal name followed by an asterisk, such as **Reset***, is low active.

The primary purpose of the Soft Reset exception is to reinitialize the processor after a fatal error that occurs during normal operations. 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 and normal exception vectors need not be properly initialized. Soft Reset preserves the state of the caches and memory system, while resetting the bus state and cache state machine.

When this exception occurs, the contents of all registers are preserved exceptas follows:

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

Soft reset exception processing is shown in Figure 5.14.

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

This section explains the Nonmaskable Interrupt 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.

Unlike all other interrupts, this interrupt is not maskable; it occurs regardless of the settings of the *EXL*, *ERL*, and the *IE* bits in the *Status* register.

Processing

The Reset exception vector is used for this exception. This vector is located within unmapped and uncached address space so that the cache does not need to be initialized to process an NMI interrupt. When an NMI exception occurs, the *SR* bit of the *Status* register is set to differentiate this exception from a Reset exception.

Because an NMI can occur in the midst of another exception, it is not normally 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.

To terminate a pending read that has hung the best approach is to return a bus error. However, if you wish to use a CPU exception to indicate a hung read, Soft Reset is preferable to NMI.

When this exception occurs, the contents of all registers are preserved except for:

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

NMI exception processing is shown in Figure 5.14 on page 5-12.

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

This section explains the Address Error exception.

Cause

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

- load or store a doubleword that is not aligned on a doubleword boundary (except for use of special instruction)
- load, fetch, or store a word that is not aligned on a word boundary (except for use of special instruction)
- load or store a halfword that is not aligned on a halfword boundary
- reference the kernel address space from User mode (STATUS UM =1 and VADDR(31) = 1)

This exception is not maskable.

Processing

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

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

The *EPC* register contains the address of the instruction that caused the exception, unless this instruction is in a branch delay slot. If it is in a branch delay slot, the *EPC* register contains the address of the preceding branch instruction, and the *BD* bit of the *Cause* register is set to indicate this. Address Error exception processing is shown in Figure 5.15.

Servicing

Typically, the process that is executing at the time is handed a segmentation violation signal. This error is usually fatal to the process that incurs the exception.

To resume execution, the *EPC* register must be altered so that the unaligned reference instruction does not re-execute. This is accomplished by adding a value of 4 to the *EPC* register (*EPC* register + 4) before returning.

If an unaligned reference instruction is in a branch delay slot, interpretation of the branch instruction is required to resume execution.

Cache Error Exception

This section explains the Cache Error exception.

Cause

The Cache Error exception occurs when a primary cache parity error is detected. This exception is maskable by the *DE* bit of the *Status* register.

Processing

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, as follows:

- If the BEV bit = 0, the vector is 0xA000 0100.
- If the BEV bit = 1, the vector is 0xBFC0 0300.

No other registers are changed. Cache Error exception processing is shown in Figure 5.13.

Servicing

All errors should be logged. To correct cache parity errors the system uses the CACHE instruction to invalidate the cache block, overwrites the old data through a cache miss, and resumes execution with an ERET.

Other errors are not correctable and are likely to be fatal to the current process.

Bus Error Exception

This section explains the Bus Error exception.

Cause

A Bus Error exception is raised by board-level circuitry for events such as bus time-out, backplane bus parity errors, and invalid physical memory addresses or access types. This exception is not maskable.

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

Processing

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

The *EPC* register contains the address of the instruction that caused the exception, unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set. Bus Error processing is shown in Figure 5.16 on page 5-13.

Servicing

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

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

The virtual address of the load and store reference can then be obtained by interpreting the instruction. The physical address can simply be calculated from the virtual address and the base.

The process executing at the time of this exception is handed a bus error signal, which is usually fatal.

Integer Overflow Exception

This section explains the Integer Overflow exception.

Cause

An Integer Overflow exception occurs when an ADD, ADDI, SUB, DADD, DADDI or DSUB instruction[1] results in a 2's complement overflow. This exception is not maskable.

[1] See Appendix A for instruction description.

Processing

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

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

Integer Overflow exception processing is shown in Figure 5.16 on page 5-13.

Servicing

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

Trap Exception

This section discusses the Trap exception.

Cause

The Trap exception occurs when a TGE, TGEU, TLT, TLTU, TEQ, TNE, TGEI, TGEUI, TLTI, TLTUI, TEQI, or TNEI instruction[1] results in a TRUE condition. This exception is not maskable.

Processing

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

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

Trap exception processing is shown in Figure 5.16 on page 5-13.

Servicing

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

System Call Exception

This section explains the System Call exception.

Cause

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

Processing

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

The *EPC* register contains the address of the SYSCALL instruction unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction.

If the SYSCALL instruction is in a branch delay slot, the *BD* bit of the *Status* register is set; otherwise this bit is cleared.

System Call exception processing is shown in Figure 5.16 on page 5-13.

[1] See Appendix A for instruction description.

Servicing

When this exception occurs, control is transferred to the applicable system routine.

To resume execution, the *EPC* register must be altered so that the SYSCALL instruction does not re-execute. This is accomplished by adding a value of 4 to the *EPC* register (*EPC* register + 4) before returning.

If a SYSCALL instruction is in a branch delay slot, a more complicated algorithm, beyond the scope of this description, may be required.

Breakpoint Exception

This section explains the Breakpoint exception.

Cause

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

Processing

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

The *EPC* register contains the address of the BREAK instruction unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction.

If the BREAK instruction is in a branch delay slot, the *BD* bit of the *Status* register is set, otherwise the bit is cleared.

Breakpoint exception processing is shown in Figure 5.16 on page 5-13.

Servicing

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

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

If a BREAK instruction is in a branch delay slot, interpretation of the branch instruction is required to resume execution.

Reserved Instruction Exception

This section explains the Reserved Instruction exception.

Cause

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

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

64-bit operations are always valid in Kernel mode regardless of the value of the *KX* bit in the *Status* register.

This exception is not maskable.

Reserved Instruction exception processing is shown in Figure 5.16 on page 5-13.

Processing

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

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

Servicing

No instructions in the R4650 ISA are currently interpreted. The process executing at the time of this exception is handed an illegal instruction/ reserved operand fault signal. This error is usually fatal.

Coprocessor Unusable Exception

This section explains the Coprocessor Unusable exception.

Cause

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

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

This exception is not maskable.

Processing

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

Coprocessor Unusable exception processing is shown in Figure 5.16 on page 5-13.

Servicing

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

- If the process is entitled access to the coprocessor, the coprocessor is marked usable and the corresponding user state is restored to the coprocessor.
- If the process is entitled access to the coprocessor, but the coprocessor does not exist or has failed, interpretation of the coprocessor instruction is possible.
- If the *BD* bit is set in the *Cause* register, the branch instruction must be interpreted; then the coprocessor instruction can be emulated and execution resumed with the *EPC* register advanced past the coprocessor instruction.
- If the process is not entitled access to the coprocessor, the process executing at the time is handed an illegal instruction/privileged instruction fault signal. This error is usually fatal.

Floating-Point Exception

This section discusses the Floating-Point exception.

Cause

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

Processing

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

The contents of the *Floating-Point Control/Status* register indicate the cause of this exception.

Floating-Point exception processing is shown in Figure 5.16 on page 5-13.

Servicing

This exception is cleared by clearing the appropriate bit in the *Floating-Point Control/Status* register.

For an unimplemented instruction exception, the kernel should emulate the instruction; for other exceptions, the kernel should pass the exception to the user program that caused the exception.

Interrupt Exception

This section discusses the Interrupt exception.

Cause

The Interrupt exception occurs when one of the eight interrupt conditions is asserted. The significance of these interrupts is dependent upon the specific system implementation.

Each of the eight interrupts can be masked by clearing the corresponding bit in the *Int-Mask* field of the *Status* register, and all of the eight interrupts can be masked at once by clearing the *IE* bit of the *Status* register.

Processing

The R4650 may use the common exception vector or a dedicated vector for this exception, determined by the *Cause* register *IV* bit. The *Int* code in the *Cause* register is set.

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

Interrupt exception processing is shown in Figure 5.16 on page 5-13.

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.

Note: Due to the write buffer, a store to an external device may not occur until after other instructions in the pipeline finish. The user must ensure that the store will occur before the return from exception instruction (ERET) is executed, otherwise the interrupt may be serviced again even though there should be no interrupt pending.

IWatch Exception

This section explains the IWatch exception.

Cause

IWatch is a read-write register that specifies an instruction virtual address that causes a Watch exception. The exception occurs when the program address matches the IWatch Register, and IWatch.I is set.

Processing

The common exception vector is used for this exception. The Watch code of the *Cause* register is set with the *IW* bit set.

Servicing

This exception is typically used during system debug. Servicing is system-specific.

DWatch Exception

This section explains the DWatch exception.

Cause

DWatch is a read-write register that specifies a data virtual address that causes a Watch exception. The exception occurs either when the program does a load and the target address matches DWatch and DWatch.R is set, or when the program does a store and the target address matches DWatch and DWatch.W is set.

Processing

The common exception vector is used for this exception. The Watch code of the *Cause* register is set with the *DW* bit set.

Servicing

This exception is typically used during system debug. Servicing is system-specific.

IBound Exception

This section explains the IBound exception.

Cause

A virtual address in kuseg exceeded the value set for IBound. The IBound register provides the User Instruction address space Bound. User virtual addresses greater than this value cause IBound exceptions.

Processing

The common exception vector is used for this exception. The UIBound code of the *Cause* register is set.

Servicing

This exception indicates that the user is trying to access memory outside the allowed page. Servicing is system-specific.

DBound Exception

This section explains the DBound exception.

Cause

A virtual address in kuseg exceeded the value set for DBound. The DBound register provides the User Data address space Bound. User virtual addresses greater than this value cause DBound exceptions.

Processing

The common exception vector is used for this exception. The UDBound code of the *Cause* register is set.

Servicing

This exception indicates that the user is trying to access memory outside the allowed page. Servicing is system-specific.

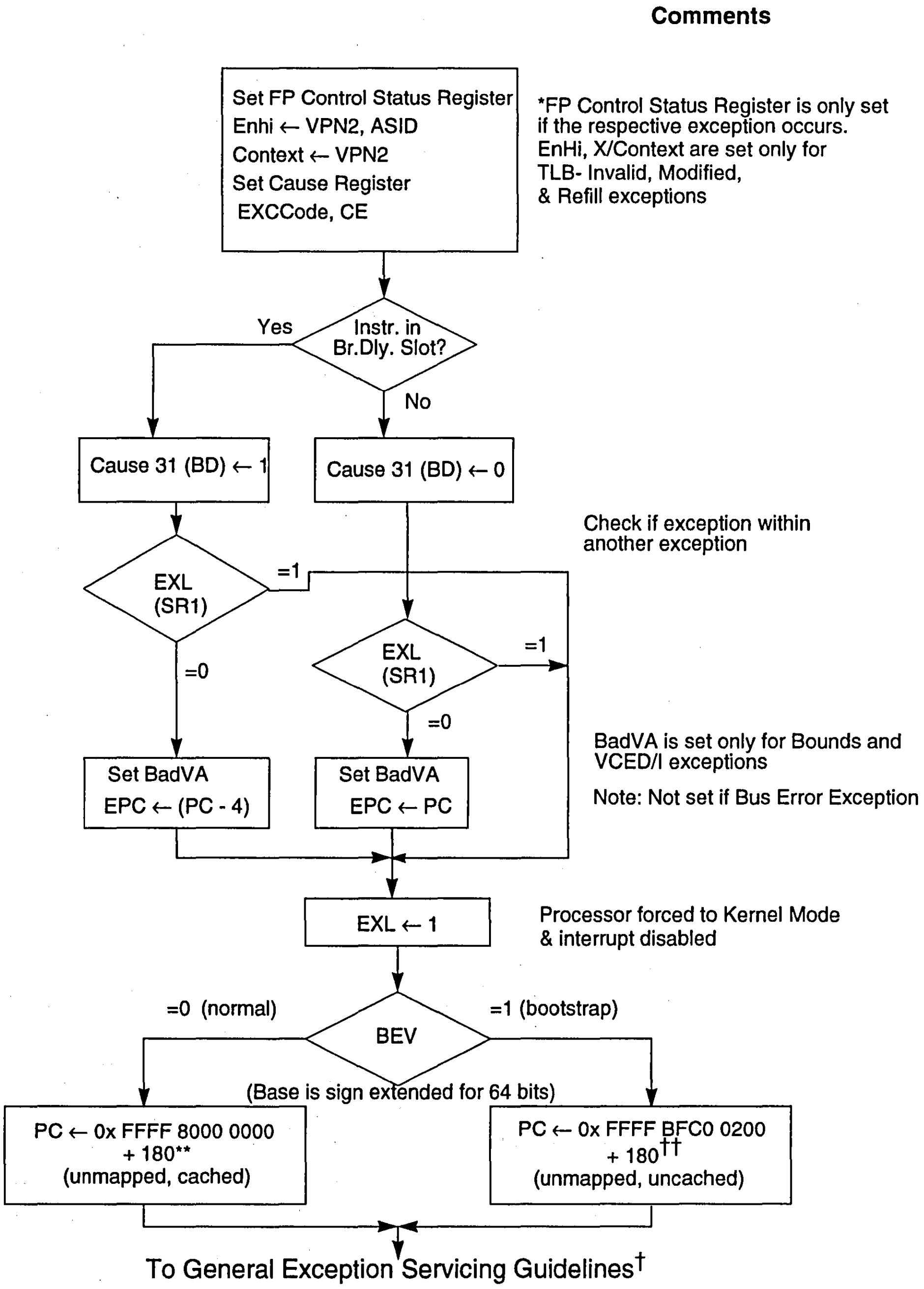
Exception Handling and Servicing Flowcharts

This section contains process flowcharts for the exceptions described in Table 5.12, as well as guidelines for the exception handlers.

Figure	Description
Figure 5.17, Figure 5.18	General exceptions and their exception handler
Figure 5.19	Cache error exception and its handler
Figure 5.20	Reset, soft reset and NMI exceptions, and a guideline to their handler.

Table 5.12 List of Exception Flowcharts

In general, the exceptions are handled by hardware (HW), and then the exceptions are serviced by software (SW).


Comments
Set FP Control Status Register
Enhi ← VPN2, ASID
Context ← VPN2
Set Cause Register
EXCCode, CE
*FP Control Status Register is only set if the respective exception occurs.
EnHi, X/Context are set only for TLB- Invalid, Modified, & Refill exceptions
Instr. in Br.Dly. Slot?
Yes
No
Cause 31 (BD) ← 1
Cause 31 (BD) ← 0
Check if exception within another exception
EXL (SR1)
=1
=0
EXL (SR1)
=1
=0
Set BadVA
EPC ← (PC - 4)
Set BadVA
EPC ← PC
BadVA is set only for Bounds and VCED/I exceptions
Note: Not set if Bus Error Exception
EXL ← 1
Processor forced to Kernel Mode & interrupt disabled
=0 (normal)
BEV
=1 (bootstrap)
(Base is sign extended for 64 bits)
PC ← 0x FFFF 8000 0000 + 180** (unmapped, cached)
PC ← 0x FFFF BFC0 0200 + 180†† (unmapped, uncached)
To General Exception Servicing Guidelines†

Exceptions other than Reset, Soft Reset, NMI, or CacheErr

Figure Notes:

†Interrupts can be masked by IE or IMs

†† 200 if cause.exc code ="Int"and cause.IV=1

Figure 5.17 General Exception Handler (HW)

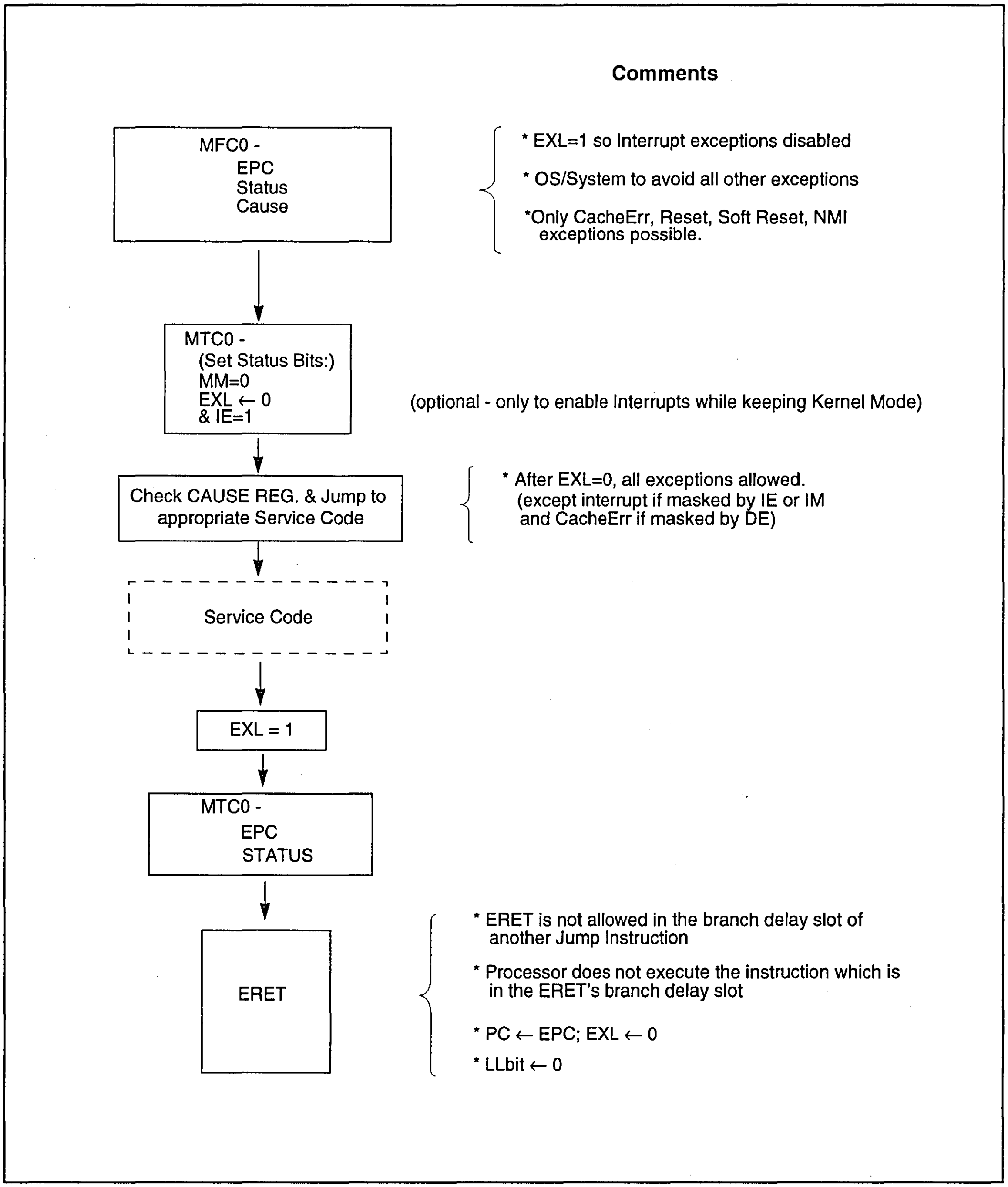

Comments
MFC0 -
EPC
Status
Cause
* EXL=1 so Interrupt exceptions disabled
* OS/System to avoid all other exceptions
*Only CacheErr, Reset, Soft Reset, NMI exceptions possible.
MTC0 -
(Set Status Bits:)
MM=0
EXL ← 0
& IE=1
(optional - only to enable Interrupts while keeping Kernel Mode)
Check CAUSE REG.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

Figure 5.18 General Exception Servicing Guidelines (SW)

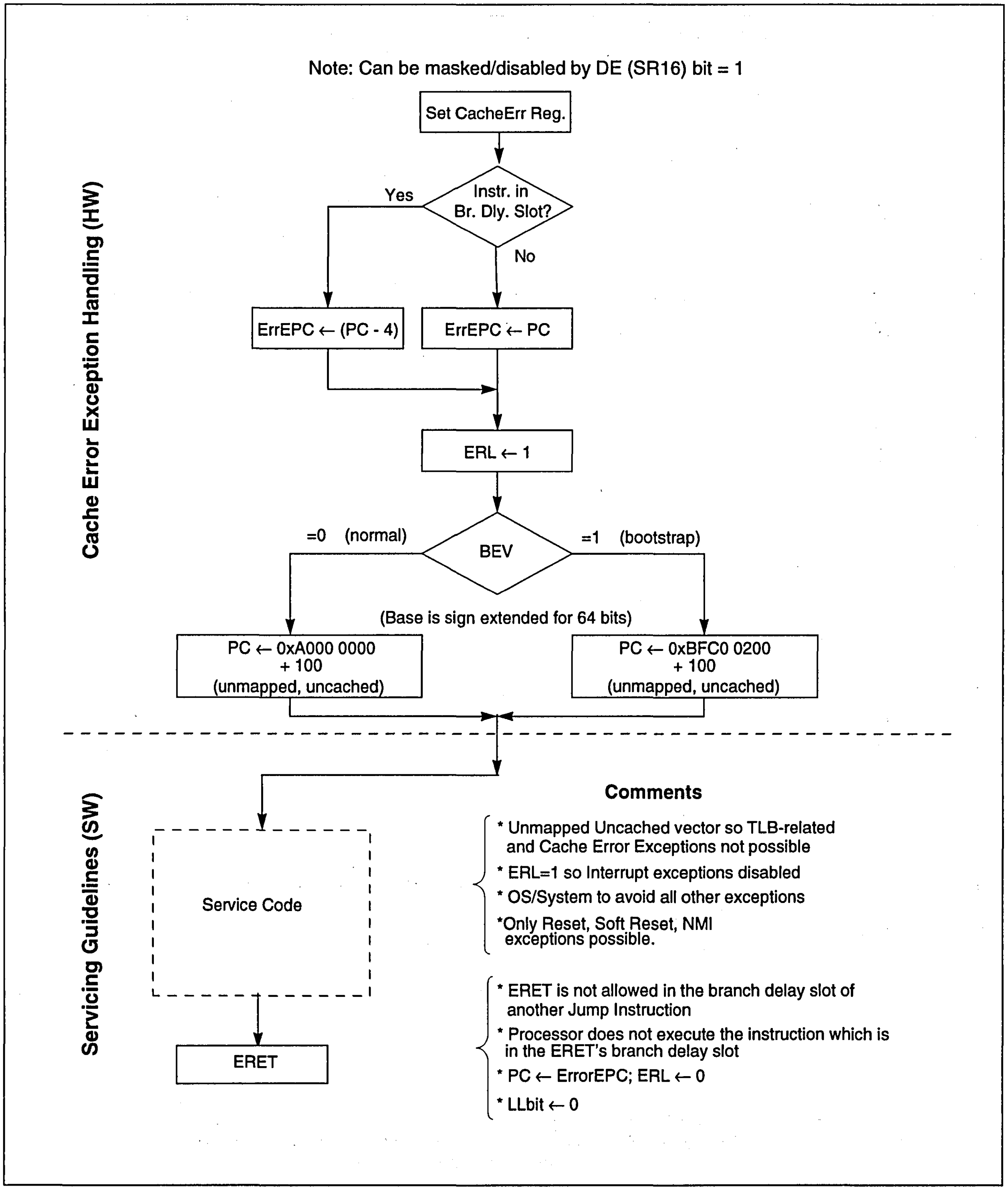

Note: Can be masked/disabled by DE (SR16) bit = 1
Cache Error Exception Handling (HW)
Set CacheErr Reg.
Instr. in Br. Dly. Slot?
Yes
No
ErrEPC ← (PC - 4)
ErrEPC ← PC
ERL ← 1
=0 (normal)
BEV
=1 (bootstrap)
(Base is sign extended for 64 bits)
PC ← 0xA000 0000 + 100 (unmapped, uncached)
PC ← 0xBFC0 0200 + 100 (unmapped, uncached)
Servicing Guidelines (SW)
Service Code
ERET
Comments
* Unmapped Uncached vector so TLB-related and Cache Error Exceptions not possible
* ERL=1 so Interrupt exceptions disabled
* OS/System to avoid all other exceptions
*Only Reset, Soft Reset, NMI exceptions possible.
* ERET is not allowed in the branch delay slot of another Jump Instruction
* Processor does not execute the instruction which is in the ERET's branch delay slot
* PC ← ErrorEPC; ERL ← 0
* LLbit ← 0

Figure 5.19 Cache Error Exception Handling (HW) and Servicing Guidelines (SW)

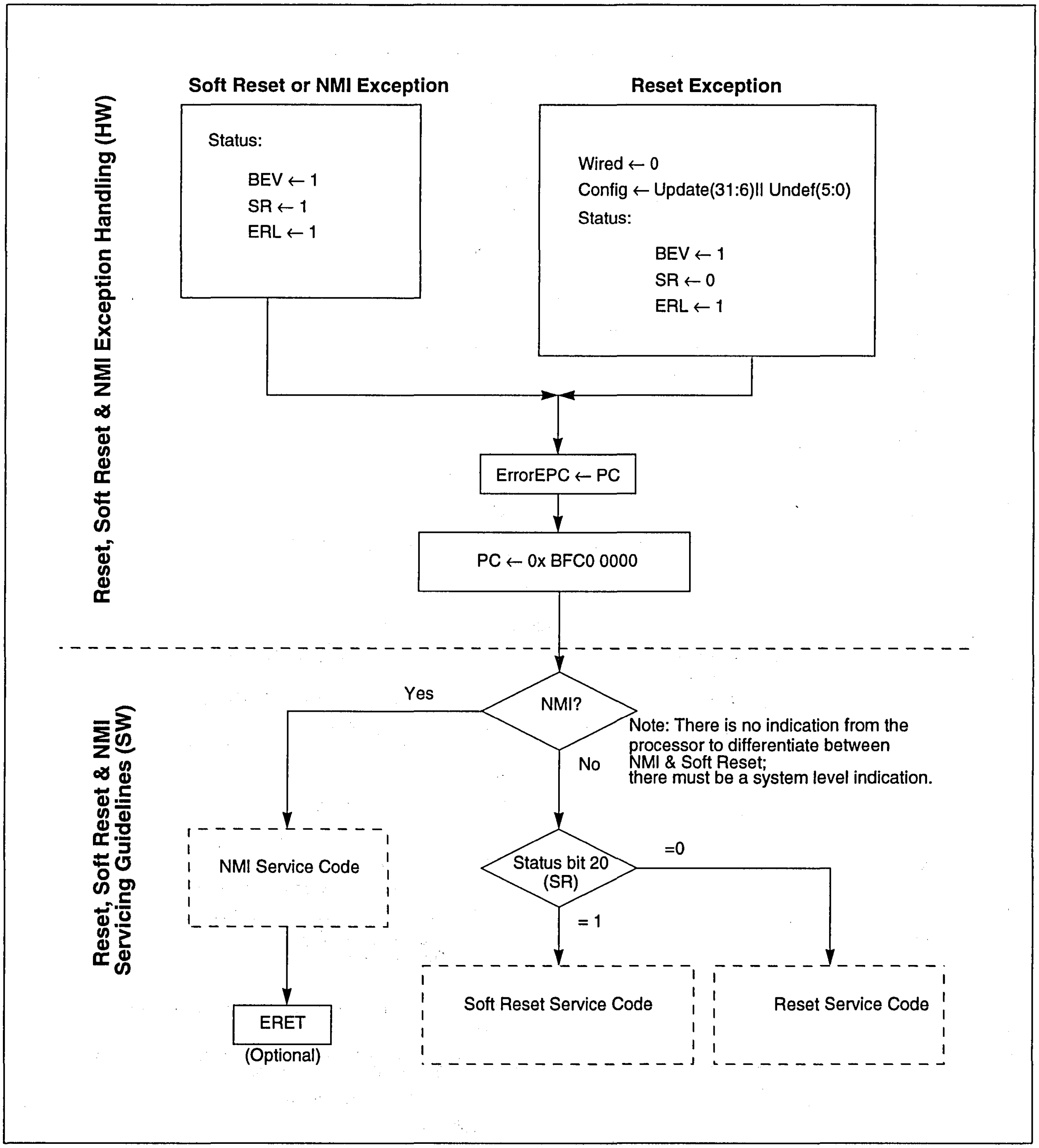

Reset, Soft Reset & NMI Exception Handling (HW)
Soft Reset or NMI Exception
Status:
BEV ← 1
SR ← 1
ERL ← 1
Reset Exception
Wired ← 0
Config ← Update(31:6)II Undef(5:0)
Status:
BEV ← 1
SR ← 0
ERL ← 1
ErrorEPC ← PC
PC ← 0x BFC0 0000
Reset, Soft Reset & NMI Servicing Guidelines (SW)
Yes
NMI?
No
Note: There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

Figure 5.20 Reset, Soft Reset & NMI Exception Handling (HW) and Servicing Guidelines (SW)

The Floating-Point Unit Chapter 6

Introduction

This chapter describes the R4650 floating-point unit (FPU) features, including the programming model, instruction set and formats, and the pipeline.

The FPU, with associated system software, conforms to the single-precision 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.

The FPU operates as a coprocessor for the CPU. It is assigned coprocessor label *CP1*, and extends the CPU instruction set to perform arithmetic operations on floating-point values.

The R4650 Floating-Point Coprocessor

The R4650 incorporates a single-precision floating-point coprocessor on chip, including a floating-point register file and execution units. The floating-point coprocessor forms a seamless interface with the integer unit, decoding and executing instructions in parallel with the integer unit.

Figure 6.1 illustrates the functional organization of the FPU.

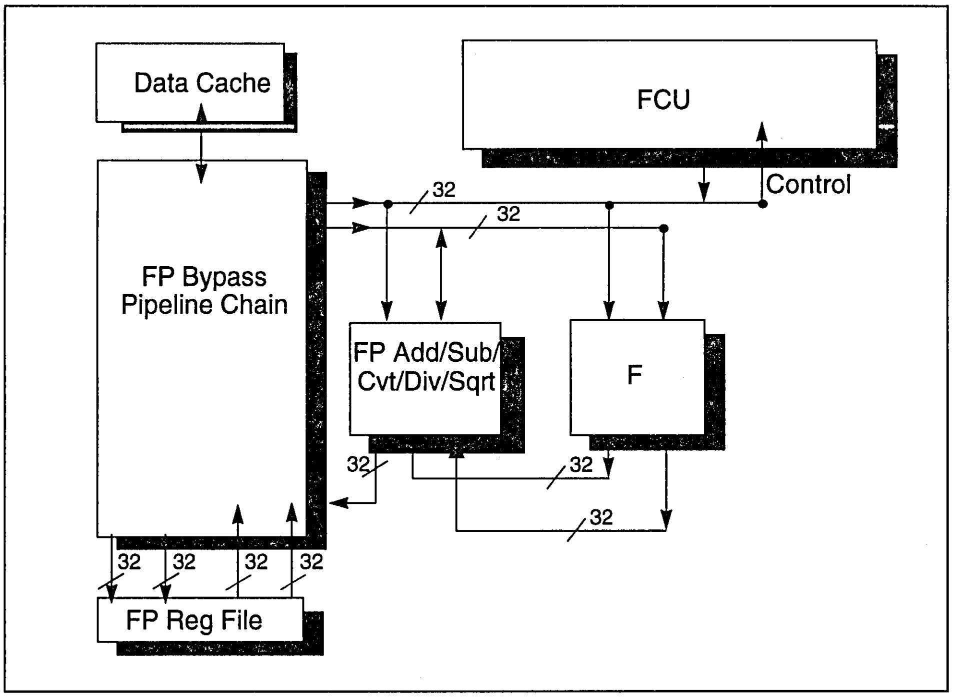


Figure 6.1 FPU Functional Block Diagram

FPU Features

This section briefly describes the operating model, the load/store instruction set, and the coprocessor interface in the FPU. A more detailed description is given in the sections that follow.

- **Single-Precision Operation**. The floating-point incorporates an adder, a multiplier, and a 32-entry, 32-bit register file for floating point operations. It also has a 32-bit control register. Overlap of multiply and add is supported.
- **Load and Store Instruction Set**. Like the CPU, the FPU uses a load- and store-oriented instruction set, with single-cycle load and store operations.
- **Tightly Coupled Coprocessor Interface**. The FPU resides on-chip to form a tightly coupled unit with a seamless integration of floating-point and fixed-point instruction sets.

FPU Programming Model

This section describes the set of FPU registers and their data organization. The FPU registers include *Floating-Point General Purpose* registers (*FGRs*) and two control registers: *Control/Status* and *Implementation/ Revision*.

Floating-Point General Registers (FGRs)

The FPU has a set of *Floating-Point General Purpose* registers (*FGRs*) that can be accessed in the following ways:

- As 32 general-purpose registers (32 FGRs), each of which is 32-bits wide. The CPU accesses these registers through move, load, and store instructions.
- As 16 floating-point registers (see the next section for a discussion of floating point registers), each of which is 32-bits wide, when the *FR* bit in the CPU *Status* register equals 0. The floating point registers hold values in single-precision floating-point format. Each floating point registers corresponds to adjacently numbered FGRs, as shown in Figure 6.2, when status FR=0. Attempts to access odd-numbered floating-point registers result in an unimplemented trap.
- As 32 floating-point registers (see the next section for a description of floating point registers), each of which is 32-bits wide, when the *FR* bit in the CPU *Status* register equals 1. The floating point registers hold values in single-precision floating-point format.

Each FPR corresponds to an FGR, as shown in Figure 6.2.

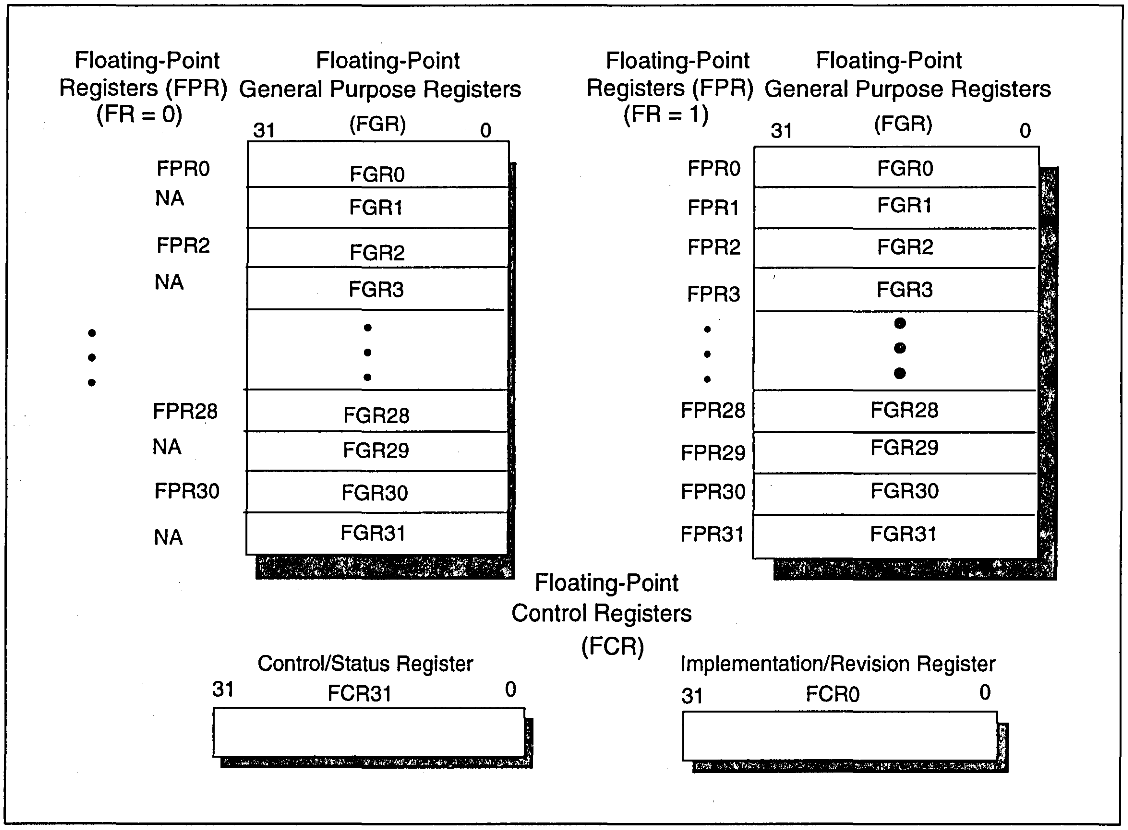

Floating-Point Registers (FPR) (FR = 0)
Floating-Point General Purpose Registers (FGR)
31
0
FPR0
NA
FPR2
NA
FPR28
NA
FPR30
NA
FGR0
FGR1
FGR2
FGR3
FGR28
FGR29
FGR30
FGR31
Floating-Point Registers (FPR) (FR = 1)
Floating-Point General Purpose Registers (FGR)
FPR0
FPR1
FPR2
FPR3
FPR28
FPR29
FPR30
FPR31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

Figure 6.2 FPU Registers

Floating-Point Registers

The FPU provides:

- 16 *Floating-Point* registers (*FPRs*) for *Status.FR* = 0, or
- 32 *Floating-Point* registers (*FPRs*) for *Status.FR* = 1.

These 32-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 1, the *FPR* references a single 32-bit *FGR*.

The *FPRs* hold values in single-precision floating-point format. If the *FR* bit equals 0, only even numbers (as shown in Figure 6.2) can be used to address *FPRs*. When the *FR* bit is set to 1 all *FPR* register numbers are valid.

Floating-Point Control Registers

The FPU has 32 control registers (*FCRs*) that can only be accessed by move operations. The *FCRs* are described below:

- The *Implementation/Revision* register *(FCR0)* holds revision information about the FPU.
- The *Control/Status* register *(FCR31)* controls and monitors exceptions, holds the result of compare operations, and establishes rounding modes.
- *FCR1* to *FCR30* are reserved.

Table 6.1 lists the assignments of the *FCR* registers.

FCR Number	**Use**
FCR0	Coprocessor implementation and revision register
FCR1 to FCR30	Reserved
FCR31	Rounding mode, cause, trap enables, and flags

Table 6.1 Floating-Point Control Register Assignments

Implementation and Revision Register, (FCR0)

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

Figure 6.3 shows the layout of the register; Table 6.2, which follows the figure, describes the *Implementation and Revision* register (*FCR0*) fields.

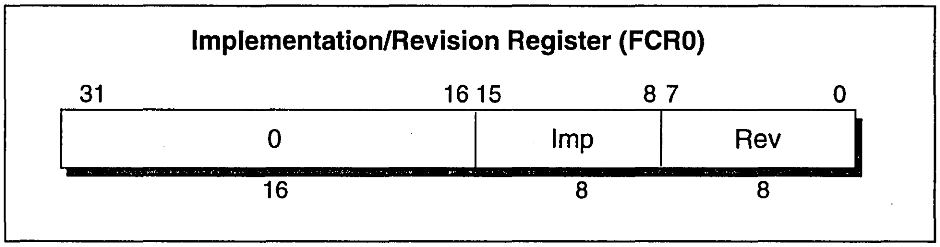


Figure 6.3 Implementation/Revision Register

Field	**Description**
Imp	Implementation number (0x22 in R4650)
Rev	Revision number in the form of *y.x*
0	Reserved.

Table 6.2 FCR0 Fields

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

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

The revision number distinguishes some chip revisions; however, there is no guarantee that changes to the chip are necessarily reflected by the revision number, or that changes to the revision number necessarily reflect real chip changes. For this reason revision number values are not listed, and software should not rely on the revision number to characterize the chip.

Control/Status Register (FCR31)

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

Figure 6.4 shows the format of the *Control/Status* register, and Table 6.3, which follows the figure, describes the *Control/Status* register fields.

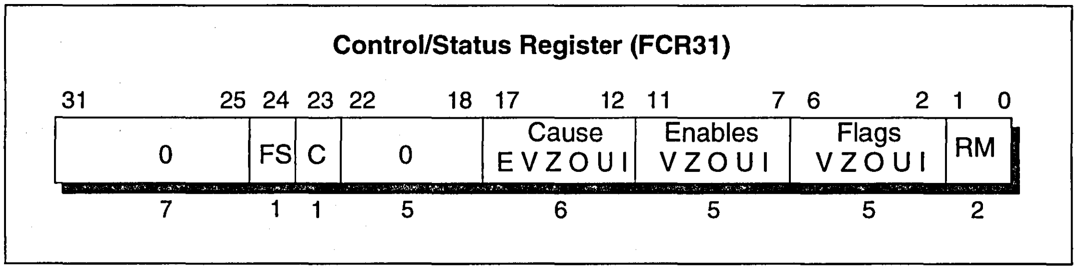


Figure 6.4 FP Control/Status Register Bit Assignments

Field	Description
FS	When set, denormalized results are flushed to 0 instead of causing an unimplemented operation exception.
C	Condition bit. See description of *Control/Status* register *Condition* bit.
Cause	Cause bits. See Figure 6.5 and the description of *Control/Status* register *Cause, Flag,* and *Enable* bits.
Enables	Enable bits. See Figure 6.5 and the description of *Control/Status* register *Cause, Flag,* and *Enable* bits.
Flags	Flag bits. See Figure 6.5 and the description of *Control/Status* register *Cause, Flag,* and *Enable* bits.
RM	Rounding mode bits. See Table 6.4, found on page 8, and the description of *Control/Status* register *Rounding Mode Control* bits.

Table 6.3 Control/Status Register Fields

Figure 6.5 shows the *Control/Status* register *Cause, Flag,* and *Enable* fields.

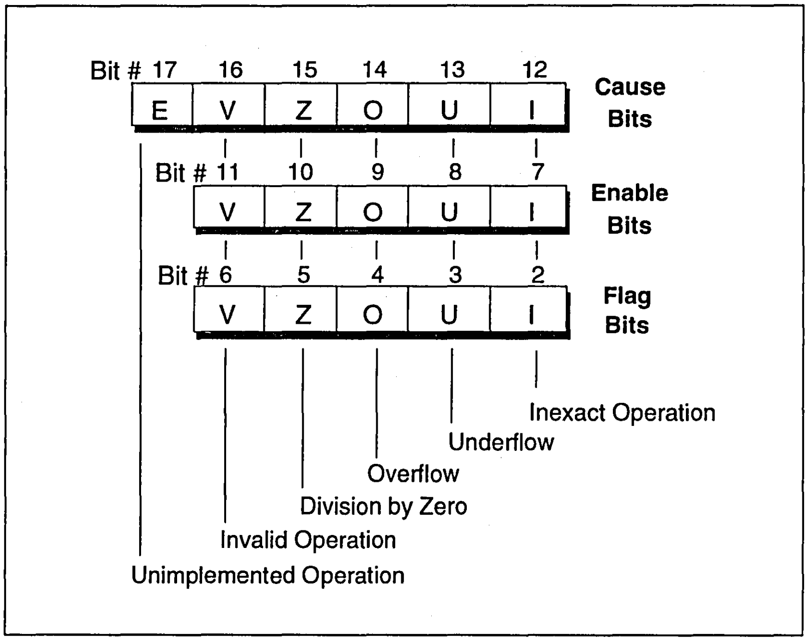


Figure 6.5 Control/Status Register Cause, Flag, and Enable Fields

Accessing the Control/Status Register

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

The bits in the *Control/Status* register can be set or cleared by writing to the register using a Move Control To Coprocessor 1 (CTC1) instruction. CTC1 is not issued until all previous floating-point operations are complete.

IEEE Standard 754

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

Control/Status Register FS Bit

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

Control/Status Register Condition Bit

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

Control/Status Register Cause, Flag, and Enable Fields

Figure 6.5 illustrates the *Cause*, *Flag*, and *Enable* fields of the *Control/Status* register.

Cause Bits

Bits 17:12 in the *Control/Status* register contain *Cause* bits, which reflect the results of the most recently executed instruction. These bits are illustrated in Figure 6.5. 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. If more than one exception occurs on a single instruction, each appropriate bit is set.

The *Cause* bits are written by each floating-point operation (but not by load, store, or move operations). The Unimplemented Operation (*E*) bit is set to a 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.

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

Enable Bits

A floating-point operation that sets an enabled *Cause* bit forces an immediate exception, as does setting both *Cause* and *Enable* bits with CTC1. The floating-point exception or interrupt is enabled when the corresponding enable be is set.

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

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.

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.

Flag Bits

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

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

Control/Status Register Rounding Mode Control Bits

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

As shown in Table 6.4, these bits specify the rounding mode that the FPU uses for all floating-point operations.

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

Table 6.4 Rounding Mode Bit Decoding

Floating-Point Formats

The FPU performs 32-bit (single-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.

The floating-point accelerator (FPA) does not perform 64-bit (double-precision) operations. Thus, instructions requiring 64-bit data support in the FPA cause the unimplemented exception to be signaled, allowing software emulation if desired.

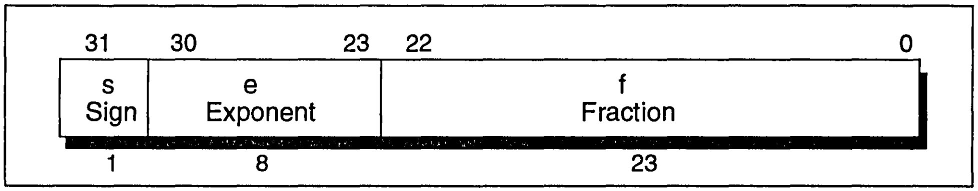


Figure 6.6 Single-Precision Floating-Point Format

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

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

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

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

Each representable nonzero numerical value has just one encoding.

The value of a number, v, is determined by the equations shown in Table 6.5.

No.	Equation
(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 $v = (-1)^s 2^{Emin}(0.f)$
(5)	if $E = E_{min}-1$ and $f = 0$, then $v = (-1)^s 0$

Table 6.5 Equations for Calculating Values in Single-Precision Floating-Point Format

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 6.6 defines the values for the format parameters.

Parameter	Single Precision Format
f	24
E_{max}	+127
E_{min}	-126
Exponent *bias*	+127
Exponent width in bits	8
Integer bit	hidden
Fraction width in bits	24
Format width in bits	32

Table 6.6 Floating-Point Format Parameter Values

Table 6.7 shows minimum and maximum floating-point values.

Type	Value
Float Minimum	1.40129846e–45
Float Minimum Norm	1.17549435e–38
Float Maximum	3.40282347e+38

Table 6.7 Minimum and Maximum Floating-Point Values

Binary Fixed-Point Format

Binary fixed-point values are held in 2's complement format. Unsigned fixed-point values are not directly provided by the floating-point instruction set. Figure 6.7 illustrates binary fixed-point format. Table 6.8, which follows the figure, lists the binary fixed-point format fields.

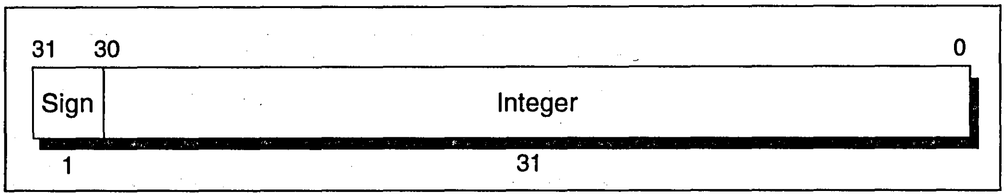


Figure 6.7 Binary Fixed-Point Format

Field	Description
sign	sign bit
integer	integer value

Table 6.8 Binary Fixed-Point Format Fields

Floating-Point Instruction Set Overview

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

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

Table 6.9 through Table 6.12 list the instruction set of the FPU. A complete description of each instruction is provided in Appendix B.

Key to Formats in Table 6.9 through Table 6.12

In the instruction formats shown in Table 6.9 through Table 6.12, the *fmt* appended to the instruction opcode specifies the data format: *s* specifies single-precision binary floating-point, *d* specifies double-precision binary floating-point, *w* specifies 32-bit binary fixed-point, and L specifies 64-bit binary fixed-point.

OpCode	Description
LWC1	Load Word to FPU
SWC1	Store Word from FPU
LDC1	Load Doubleword to FPU[1]
SDC1	Store Doubleword from FPU[1]
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[1]
DMFC1	Doubleword Move from FPU[1]

Note:
[1] This opcode causes an unimplemented exception in the R4650.

Table 6.9 FPU Instruction Summary: Load, Move and Store Instructions

OpCode[3 ,4]	Description
CVT.S.fmt	Floating-point Convert to Single FP[2]
CVT.D.fmt	Floating-point Convert to Double FP[1]
CVT.W.fmt	Floating-point Convert to Single Fixed Point[2]
ROUND.w.fmt	Floating-point Round
ROUND.L.fmt[1]	Floating-point Round
TRUNC.w.fmt	Floating-point Truncate
TRUNC.L.fmt[1]	
CEIL.w.fmt	Floating-point Ceiling
CEIL.L.fmt[1]	
FLOOR.w.fmt	Floating-point Floor
FLOOR.L.fmt[1]	

Notes:
[1] This opcode causes an unimplemented exception in the R4650.
[2] The CVT.fmt.D opcode also causes an unimplemented exception in the R4650.
[3] For definitions of the abbreviations.*fmt*, *s*, *d*, and *w* refer to the text preceding Table 6.9.
[4] An unimplemented exception is signalled when fmt = "D" or fmt = "L".

Table 6.10 FPU Instruction Summary: Conversion Instructions

OpCode[1,2]	Description
ADD.fmt	Floating-point Add
SUB.fmt	Floating-point Subtract
MUL.fmt	Floating-point Multiply
DIV.fmt	Floating-point Divide
ABS.fmt	Floating-point Absolute Value
MOV.fmt	Floating-point Move
NEG.fmt	Floating-point Negate
SQRT.fmt	Floating-point Square Root

Notes:
[1] For definitions of the abbreviations.*fmt*, *s*, *d*, and *w* refer to the text preceding Table 6.9.
[2] For all entries in the OPCODE column *.fmt* must be set to *.S* or a trap will be signaled.

Table 6.11 FPU Instruction Summary: Computational Instructions

OpCode[1,2]	Description
C.cond.fmt	Floating-point Compare
BC1T	Branch on FPU True
BC1F	Branch on FPU False
BC1TL	Branch on FPU True Likely
BC1FL	Branch on FPU False Likely

Notes:
[1] For definitions of the abbreviations*.fmt*, *s*, *d*, and *w* refer to the text preceding Table 6.9.
[2] For all entries in the OPCODE column, if *.fmt* is set to *.D* a trap will be signaled.

Table 6.12 FPU Instruction Summary: Compare and Branch Instructions

Floating-Point Load, Store, and Move Instructions

This section discusses the manner in which the FPU uses the load, store and move instructions listed in Table 6.9. Appendix B provides a detailed description of each instruction.

Transfers Between FPU and Memory

All data movement between the FPU and memory is accomplished by using the instructions Load Word To Coprocessor 1 (LWC1) or Store Word To Coprocessor 1 (SWC1), which reference a single 32-bit word of the FPU general registers.

These load and store operations are unformatted. Since no format conversions are performed, no floating-point exceptions can result from these operations.

Transfers Between FPU and CPU

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

- Move To Coprocessor 1 (MTC1)
- Move From Coprocessor 1 (MFC1)

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

Load Delay and Hardware Interlocks

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

Data Alignment

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

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

Endianness

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

Floating-Point Conversion Instructions

Conversion instructions perform conversions between the various data formats such as single-precision, fixed- or floating-point formats. Table 6.10 lists conversion instructions. Appendix B, "FPU Instruction Set Details," describes each instruction.

Floating-Point Computational Instructions

Computational instructions perform arithmetic operations on floating-point values, in registers. Table 6.11 lists the computational instructions and Appendix B provides a detailed description of each instruction. There are two categories of computational instructions:

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

Branch on FPU Condition Instructions

Table 6.12 lists the Branch on FPU (coprocessor unit 1) condition instructions that can test the result of the FPU compare (C.cond) instructions. Appendix B gives a detailed description of each instruction.

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 6.12, found on page 12, lists the compare instructions. Table 6.13 lists the mnemonics for the compare instruction conditions. The.*W* and.*S* formats are allowed for in the R4650. The.*D* format causes a trap to be signaled. For detailed descriptions of these instructions, refer to Appendix B, "FPU Instruction Set Details."

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

Table 6.13 Mnemonics and Definitions of Compare Instruction Conditions

FPU Instruction Pipeline Overview

The FPU provides an instruction pipeline that parallels the CPU instruction pipeline. It shares the same five-stage pipeline architecture with the CPU. Refer to Chapter 3 for details about the pipeline architecture.

Instruction Execution

Figure 6.8 illustrates the 5-stage FPU pipeline. This is the same as that of the integer pipeline but allows for the longer execution times of the floating-point instructions.

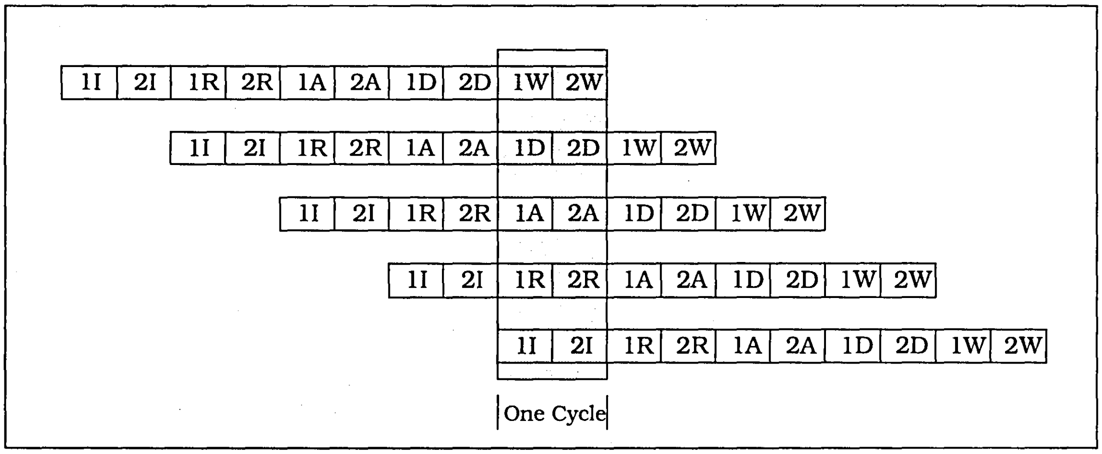

1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
1I 2I 1R 2R 1A 2A 1D 2D 1W 2W
One Cycle

Figure 6.8 FPU Instruction Pipeline

Figure 6.8 assumes that one instruction is completed every PCycle, but most FPU instructions 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.

Floating-point operations proceed in parallel with non-floating-point operations. Floating-point operations are not allowed to overlap each other, with two exceptions:

- An add operation may start 2 cycles after the start of a multiply and thus will be completely overlapped by the multiply.
- A multiply operation may overlap for up to 2 cycles, and start 6 cycles after another multiply.

Non-floating-point operations as well as other integer operations may be executed in parallel with the floating-point operations. All of this is handled automatically by internal hardware in the R4650.

Instruction Execution Cycle Time

Unlike the CPU, which executes almost all instructions in a single cycle, more time may be required to execute FPU instructions.

Table 6.14 gives the minimum latency of each floating-point operation.

Operation	Pipeline Cycles		Operation	Pipeline Cycles	
	Single	Double		Single	Double
ADD.fmt	4	(b)	BC1T	1	
SUB.fmt	4	(b)	BC1F	1	
MUL.fmt	8	(b)	BC1TL	1	
DIV.fmt	32	(b)	BC1FL	1	
SQRT.fmt	31	(b)	LWC1, LDC1	2	
ABS.fmt	1	(b)	SWC1, SDC1	1	
MOV.fmt	1	(b)	TRUNC.W.fmt	4	(b)
NEG.fmt	1	(b)	MTC1, DMTC1	2	
ROUND.W.fmt	4	(b)	MFC1, DMFC1	2	
CEIL.W.fmt	4	(b)	CTC1	3	
FLOOR.W.fmt	4	(b)	CFC1	2	
CVT.S.fmt	(a)	(b)	CMP	3	(b)
CVT.D.fmt	(b)	(b)	FIX	4	(b)
CVT.W.fmt	4(a)	(b)	FLOAT	6	(b)
C.fmt.cond	3	(b)			

Notes:
[a] If .fmt = .D or.fmt = .L, a trap will occur.
[b] These operations cause a trap.

Table 6.14 Floating-Point Operation Latencies

Instruction Scheduling Constraints

The FPU resource scheduler only issues instructions to the FPU op units (adder and multiplier) when no hardware use conflicts will occur. In addition, some overlap possibilities are disallowed to keep the scheduler simple (and/or increase performance).

FPU Multiplier Constraints

The FPU multiplier is partially pipelined in the R4650, allowing a new multiply to begin every 6 cycles.

FPU Adder Constraints

The FPU scheduler may issue an add operation (ADD.S or SUB.S) 2 cycles after a multiply (MUL.S).

Resource Scheduling Rules

The FPU Resource Scheduler issues instructions while adhering to the rules described below. These scheduling rules optimize functional unit executions. If the rules are not followed, the hardware interlocks to guarantee correct operation.

DIV.[S] can start only when all of the following conditions are met in the 1A phase.

- The *adder* is idle (division is performed in the adder).
- The *multiplier* is idle.

MUL.[S] can start only when all of the following conditions are met in the 1A phase.

- The *multiplier* is one of the following:
  - idle.
  - Started execution at least 6 cycles earlier on the current multiply
- The *adder* is idle.

SQRT.[S] can start when the following conditions are met in the 1A phase.

- The *adder* is idle.
- The *multiplier* must be idle.

CVT.fmt instructions can only start when all of the following conditions are met in the 1A phase.

- The *adder* is idle.
- The *multiplier* is idle.

ADD.[S] or **SUB.[S]** can start only when all of the following conditions are met in the 1A phase.

- The *adder* is idle
- The *multiplier* is either:
  - idle.
  - started execution of the current multiply at least 2 cycles earlier.

NEG.[S] or **ABS.[S]** can start only when all of the following conditions are met in the 1A phase.

- The *adder* is idle.
- The *multiplier* is idle.

C.COND.[S] can start only when all of the following conditions are met in the 1A phase.

- The *adder* is idle.
- The *multiplier* is idle.

Floating-Point Exceptions Chapter 7

Introduction

This chapter describes floating point unit (FPU) floating-point exceptions, including FPU exception types, exception trap processing, exception flags, saving and restoring state when handling an exception, and trap handlers for IEEE Standard 754 exceptions.

A floating-point exception occurs whenever the FPU cannot handle either the operands or the results of a floating-point operation in its normal way. The FPU responds by generating an exception to initiate a software trap or by setting a status flag. In particular, the R4650 will trap on 64-bit floating point accelerator (FPA) operations, signalling an unimplemented exception.

Exception Types

The FP *Control/Status* register described in Chapter 6 contains an *Enable* bit for each exception type. Exception *Enable* bits determine whether an exception will cause the FPU to initiate a trap or set a status flag.

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

The FPU supports the five IEEE Standard 754 exceptions, which are shown in the following list. *Cause* bits, *Enables*, and *Flag* bits (status flags) are used.

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

The FPU adds a sixth exception type, the Unimplemented Operation (E). This exception indicates the use of a software implementation. The Unimplemented Operation exception has no *Enable* or *Flag* bit. Whenever this exception occurs, an unimplemented exception trap is taken.

Figure 7.1 illustrates the *Control/Status* register bits that support exceptions.

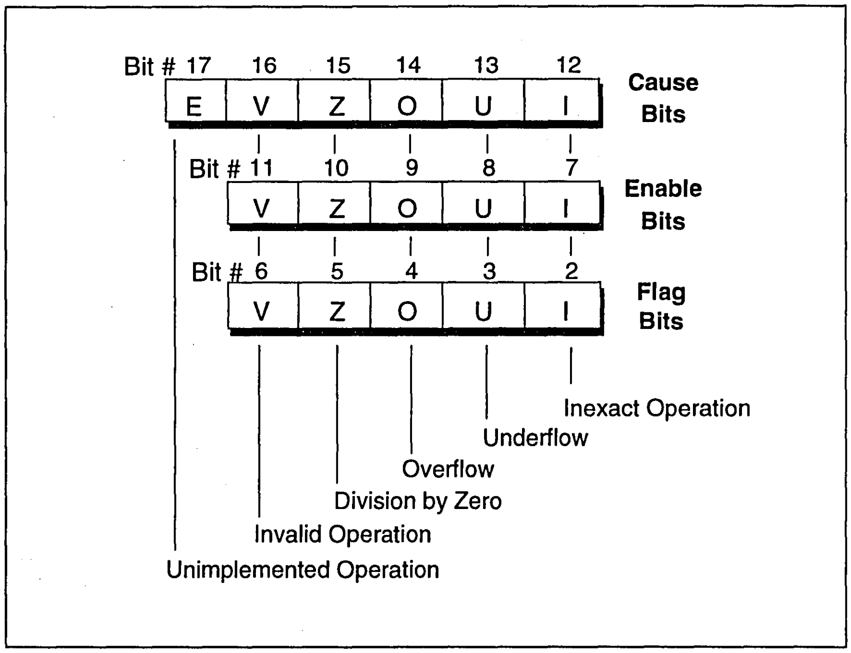


Figure 7.1 Control/Status Register Exception/Flag/Trap/Enable Bits

Each of the five IEEE Standard 754 exceptions (V, Z, O, U, I) is associated with a trap under user control, and is enabled by setting one of the five *Enable* bits. When an exception occurs and its corresponding Enable bit is not set, both the corresponding Cause and Flag bits are set. When an exception occurs and its corresponding Enable bit is set, the corresponding Cause bit is set 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 the floating-point coprocessor is the cause of the exception trap. The Floating-Point Exception (FPE) code is used, and the *Cause* bits of the floating-point *Control/Status* register indicate the reason for the floating-point exception. In effect, these bits are an extension of the system coprocessor *Cause* register.

Flags

A *Flag* bit is provided for each IEEE exception. This *Flag* bit is set to a 1 on the assertion of its corresponding exception, 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 by software either individually or as a group.

When no exception trap is signaled, the floating-point coprocessor takes a default action, providing a substitute value for the exception-causing result of the floating-point operation. The particular default action taken depends upon the type of exception.

Table 7.1 lists the default action taken by the FPU for each of the IEEE exceptions.

Field	Description	Rounding Mode	Default action
I	Inexact exception	Any	Supply a rounded result
U	Underflow exception	Any	Take unimplemented unless FCSR.FS bit is set.
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 $+\infty$
		RM	Modify positive overflows to the format's largest finite number; modify negative overflows to $-\infty$
Z	Division by zero	Any	Supply a properly signed ∞
V	Invalid operation	Any	Supply a quiet Not a Number (NaN)

Table 7.1 Default FPU Exception Actions

The FPU detects the eight exception causes internally. 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 conditions of the IEEE Standard 754.

FPA Internal Result	IEEE Standard 754	Trap Enable	Trap Disable	Notes
Inexact result	I	I	I	Loss of accuracy
Exponent overflow	O,I[a]	O,I	O,I	Normalized exponent > E_{max}
Division by zero	Z	Z	Z	Zero is (exponent = E_{min}-1, mantissa = 0)
Overflow on convert	V	E	E	Source out of integer range
Signaling NaN source	V	V	V	Signaling NaN source produces quiet NaN result
Invalid operation	V	V	V	0/0, etc.
Exponent underflow	U	E	E	Normalized exponent < E_{min}
Denormalized source	None	E	E	Exponent = E-1 and mantissa <> 0
Note: [a]The IEEE Standard 754 specifies an inexact exception on overflow only if the overflow trap is disabled.				

Table 7.2 FPU Exception-Causing Conditions

FPU Exceptions

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 condition.

Inexact Exception (I)

The FPU generates the Inexact exception if the rounded result of an operation is not exact or if it overflows. The FPU usually examines the operands of floating-point operations before execution actually begins, to determine (based on the exponent values of the operands) if the operation can *possibly* cause an exception. If there is a possibility of an instruction causing an exception trap, the FPU uses a coprocessor stall to execute the instruction.

It is impossible, however, for the FPU to predetermine if an instruction will produce an inexact result. If Inexact exception traps are enabled, the FPU uses the coprocessor stall mechanism to execute all floating-point operations that require more than two cycles. 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
- 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 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 Standard 754-specified functions implemented in software, such as Remainder: *x* REM *y*, where *y* is 0 or *x* is infinite; conversion of a floating-point number to a decimal format whose value causes an overflow, is infinity, or is NaN; and transcendental functions, such as ln (–5) or cos–1(3). Refer to Appendix B for examples or routines to handle these cases.

Trap Enabled Results: The original operand values are undisturbed.

Trap Disabled Results: The FPU sets the Invalid Operation Exception flag and a quiet NaN is delivered to the destination register.

Division-by-Zero Exception (Z)

The Division-by-Zero exception is signaled on an implemented divide operation if the divisor is zero and the dividend is a finite nonzero number. Software can simulate this exception for other operations that produce a signed infinity, such as ln(0), sec(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, with an unbounded exponent range, is larger than the largest finite number of the destination format. This exception also sets the Inexact exception and *Flag* bits.

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.

Underflow Exception (U)

Two related events contribute to the Underflow exception. IEEE Standard 754 allows detection of these events in a variety of ways. The events are:

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

The MIPS architecture requires tiny numbers to be detected after rounding. Tiny numbers can be detected by one of the following methods:

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

The MIPS architecture requires that loss of accuracy be detected as an inexact result. Loss of accuracy can be detected by one of the following two methods:

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

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 and FCSR.FS is clear, then take an unimplemented exception. When an underflow trap is not enabled and FCSR.FS is set, raise Inexact and return either 0 or $\pm 2^{Emin}$, as appropriate for the current rounding mode.

Unimplemented Instruction Exception (E)

Any attempt to execute an instruction with an unsupported operation code or format code sets the *Unimplemented* bit in the *Cause* field in the FPU *Control/Status* register and traps. The operand and destination registers remain undisturbed and the instruction may be emulated in software. Any of the IEEE Standard 754 exceptions can arise from the emulated operation, and these exceptions in turn are simulated. In the case of the R4650, 64-bit FPA operations, including Compare, Cvt, Arithmetic, Load/Store, and Move will cause this exception to be signaled.

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

- Denormalized operand
- Quiet NaN operand
- Underflow
- Reserved opcodes
- Unimplemented formats
- Conversion of a floating-point number to a fixed point format when an overflow occurs or when the source operand value is Infinity or a NaN.
- Operations that are invalid for their format (for instance, CVT.S.S)

Denormalized and NaN operands are only trapped if the instruction is a convert or computational operation. Moves and compares 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 new, and are not expected to be widely used in early implementations. Loopholes are provided in the architecture so that these conditions can be implemented with assistance provided by software, maintaining full compatibility with the IEEE Standard 754.

Trap Enabled Results: The original operand values are undisturbed.

Trap Disabled Results: This trap cannot be disabled.

Saving and Restoring State

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

When the coprocessor *Control/Status* register (*FCR31*) is read, and the coprocessor is executing one or more floating-point instructions, the instruction(s) in progress are either completed or reported as exceptions. The architecture requires that no more than one of these pending instructions can cause an exception. Information indicating the type of exception is placed in the *Control/Status* register. When state is restored, state information in the status word indicates that exceptions are pending.

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

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

Trap Handlers for IEEE Standard 754 Exceptions

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

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

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

On Overflow or Underflow exceptions (except for conversions), and on Inexact exceptions, the trap handler gains access to the correctly rounded result by examining source registers and simulating the operation in software.

On Overflow or Underflow exceptions encountered on floating-point conversions, and on Invalid Operation and Divide-by-Zero exceptions, the trap handler gains access to the operand values by examining the source registers of the instruction.

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

Integrated Device Technology, Inc.

Processor Signal Descriptions

Chapter 8

Introduction

This chapter describes the signals used by and in conjunction with the R4650 processor. The signals include the System interface, the Clock/ Control interface, the Interrupt interface, and the Initialization interface.

Signals are listed in bold, and low active signals have a trailing asterisk. For example, the low-active Read Ready signal is **RdRdy***. The signal description also tells if the signal is an input (the processor receives it) or output (the processor sends it out).

Figure 8.1 illustrates the functional groupings of the processor signals.

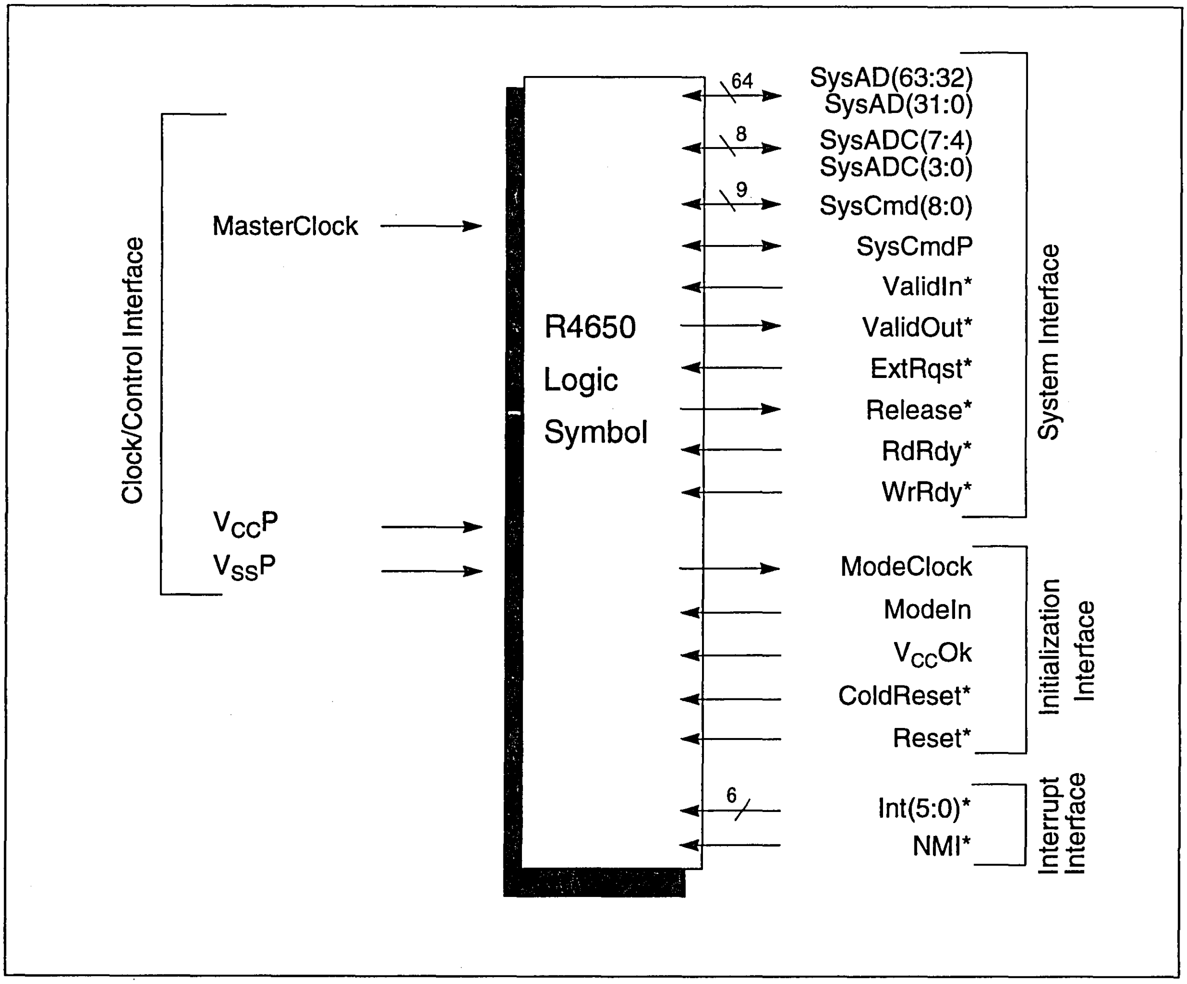


Figure 8.1 R4650 Processor Signals

System Interface Signals

System interface signals provide the connection between the R4650 processor and the other components in the system. Table 8.1 lists the system interface signals that apply when the CPU is in 64-bit system interface mode.

Name	Definition	Direction	Description
ExtRqst*	External request	Input	An external agent asserts **ExtRqst*** to request use of the System interface. The processor grants the request by asserting **Release***.
Release*	Release interface	Output	In response to the assertion of**ExtRqst*** or a CPU read request, the processor asserts **Release***, signalling to the requesting device that the System interface is available.
RdRdy*	Read ready	Input	The external agent asserts **RdRdy*** to indicate that it can accept a processor read request.

SysAD(63:32)
SysAD(31:0)
	System address/ data bus	Input/ Output	A 64-bit address and data bus for communication between the processor and an external agent. During address phases only **SysAd(31:0)** contains valid address information.

SysADC(7:4)
SysADC(3:0)
	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	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, always driven low.
ValidIn*	Valid input	Input	The external agent asserts **ValidIn*** when it is driving a valid address or data on the SysAD bus and a valid command or data identifier on the SysCmd bus.
ValidOut*	Valid output	Output	The processor asserts **ValidOut*** when it is driving a valid address or data on the SysAD bus and a valid command or data identifier on the SysCmd bus.
WrRdy*	Write ready	Input	An external agent asserts **WrRdy*** when it can accept a processor write request.

Table 8.1 System Interface Signals in 64-Bit Mode

Table 8.2 lists the system interface signals that apply when the CPU is in 32-bit system interface mode. In this mode **SysAD (63:32)** and **SysADC (7:6)** are not used, regardless of Endianness.

Name	Definition	Direction	Description
ExtRqst*	External request	Input	An external agent asserts **ExtRqst*** to request use of the System interface. The processor grants the request by asserting **Release***.
Release*	Release interface	Output	In response to the assertion of**ExtRqst*** or a CPU read request, the processor asserts **Release***, signalling to the requesting device that the System interface is available.
RdRdy*	Read ready	Input	The external agent asserts **RdRdy*** to indicate that it can accept a processor read request.
SysAD(31:0)	System address/ data bus	Input/ Output	A 64-bit address and data bus for communication between the processor and an external agent. **SysAD (63:32)** is not used in 32-bit mode, regardless of Endianness.
SysADC(3:0)	System address/ data check bus	Input/ Output	A 4-bit bus containing check bits for the **SysAD** bus.
SysCmd(8:0)	System command/ data identifier	Input/ Output	A 9-bit bus for command and data identifier transmission between the processor and an external agent.
SysCmdP	System command/ data identifier bus parity	Input/ Output	A single, even-parity bit for the **SysCmd** bus, always driven low.
ValidIn*	Valid input	Input	The external agent asserts **ValidIn*** when it is driving a valid address or data on the SysAD bus and a valid command or data identifier on the SysCmd bus.
ValidOut*	Valid output	Output	The processor asserts **ValidOut*** when it is driving a valid address or data on the SysAD bus and a valid command or data identifier on the SysCmd bus.
WrRdy*	Write ready	Input	An external agent asserts **WrRdy*** when it can accept a processor write request.

Table 8.2 System Interface Signals in 32-Bit System Interface Mode

Clock/Control Interface Signals

The Clock/Control interface signals make up the interface for clocking and maintenance.

Table 8.3 lists the Clock/Control interface signals. The same clock signals are used for both 32-bit and 64-bit system interface modes.

Name	Definition	Direction	Description
MasterClock	Master clock	Input	Master clock input that establishes the processor operating frequency. It is multiplied internally by 2, 3, 4, 5, 6, 7, or 8 to generate the pipeline clock (**PClock**)
V_{CC}P	Quiet V_{CC} for PLL	Input	Quiet V_{CC} for the internal phase locked loop.
V_{SS}P	Quiet V_{SS} for PLL	Input	Quiet V_{SS} for the internal phase locked loop.

Table 8.3 Clock/Control Interface Signals

Interrupt Interface Signals

The Interrupt interface signals make up the interface that is used by external agents to interrupt the R4650 processor. Six hardware interrupts (**Int*(5:0)**) and one NMI are available on the R4650. Table 8.4 lists the Interrupt interface signals. The same signals are used for 32-bit and 64-bit system interface modes.

Name	Definition	Direction	Description
Int*(5:0)	Interrupt	Input	Six general processor interrupts, bit-wise OR'd with bits 5:0 of the interrupt register.
NMI*	Nonmaskable interrupt	Input	Nonmaskable interrupt, OR'd with bit 6 of the interrupt register.

Table 8.4 Interrupt Interface Signals

Initialization Interface Signals

The Initialization interface signals make up the interface by which an external agent initializes the processor operating parameters. Table 8.5 lists the Initialization interface signals. The same signals are used for 32-bit and 64-bit system interface modes.

Name	Definition	Direction	Description
ColdReset*	Cold reset	Input	This signal must be asserted for a power on reset or a cold reset. **ColdReset*** must be deasserted synchronously with **MasterClock**.
ModeClock	Boot mode clock	Output	Serial boot-mode data clock output; runs at the Master Clock frequency divided by 256: (**MasterClock**/256).
ModeIn	Boot mode data in	Input	Serial boot-mode data input.
Reset*	Reset	Input	This signal must be asserted for any reset sequence. It can be asserted synchronously or asynchronously for a cold reset, or synchronously to initiate a warm reset. **Reset*** must be deasserted synchronously with **MasterClock**.
VCCOk	V_{CC} is OK	Input	When asserted, this signal indicates to the processor that $V_{CC} > V_{CC}$min for more than 100 milliseconds and will remain stable. The assertion of **VCCOk** initiates the initialization sequence.

Table 8.5 Initialization Interface Signals

Table 8.6 lists the R4650 processor signals and their possible states in 64-bit system interface mode.

Description	Name	I/O	Asserted State	3-State	Reset State
System address/data bus	**SysAD(63:0)**	I/O	High	Yes	a
System address/data check bus	**SysADC(7:0)**	I/O	High	Yes	a
System command/data identifier bus	**SysCmd(8:0)**	I/O	High	Yes	a
System command/data identifier bus parity	**SysCmdP**	I/O	High	Yes	a
Valid input	**ValidIn***	I	Low	No	NA
Valid output	**ValidOut***	O	Low	Yes	b
External request	**ExtRqst***	I	Low	No	NA
Release interface	**Release***	O	Low	Yes	b
Read ready	**RdRdy***	I	Low	No	NA
Write ready	**WrRdy***	I	Low	No	NA
Interrupts	**Int*(5:0)**	I	Low	No	NA
Nonmaskable interrupt	**NMI***	I	Low	No	NA
Boot mode data in	**ModeIn**	I	High	No	NA
Boot mode clock	**ModeClock**	O	High	No	c
Master clock	**MasterClock**	I	High	No	NA
V_{CC} is OK	**VCCOk**	I	High	No	NA
Cold reset	**ColdReset***	I	Low	No	NA
Reset	**Reset***	I	Low	No	NA

Key to Reset State Column:

a All I/O pins (SysAD[63:0], SysADC[7:0], etc.) remain 3-stated until the Reset* signal deasserts.

b All output only pins (ValidOut*, Release*, etc.), except the clocks, are 3-stated until the ColdReset* signal deasserts.

c ModeClock is always driven.

NANot applicable to input pins.

Table 8.6 R4650 Processor Signal Summary

Table 8.7 lists the R4650 processor signals and their possible states in 32-bit system interface mode. In this mode **SysADC(63:32)** and **SysADC(7:4)** are not defined.

Description	Name	I/O	Asserted State	3-State	Reset State
System address/data bus	**SysAD(31:0)**	I/O	High	Yes	a
System address/data check bus	**SysADC(3:0)**	I/O	High	Yes	a
System command/data identifier bus	**SysCmd(8:0)**	I/O	High	Yes	a
System command/data identifier bus parity	**SysCmdP**	I/O	High	Yes	a
Valid input	**ValidIn***	I	Low	No	NA
Valid output	**ValidOut***	O	Low	Yes	b
External request	**ExtRqst***	I	Low	No	NA
Release interface	**Release***	O	Low	Yes	b
Read ready	**RdRdy***	I	Low	No	NA
Write ready	**WrRdy***	I	Low	No	NA
Interrupts	**Int*(5:0)**	I	Low	No	NA
Nonmaskable interrupt	**NMI***	I	Low	No	NA
Boot mode data in	**ModeIn**	I	High	No	NA
Boot mode clock	**ModeClock**	O	High	No	c
Master clock	**MasterClock**	I	High	No	NA
V_{CC} is OK	**VCCOk**	I	High	No	NA
Cold reset	**ColdReset***	I	Low	No	NA
Reset	**Reset***	I	Low	No	NA

Key to Reset State Column:

d All I/O pins (SysAD[63:0], SysADC[7:0], etc.) remain 3-stated until the Reset* signal deasserts.

e All output only pins (ValidOut*, Release*, etc.), except the clocks, are 3-stated until the ColdReset* signal deasserts.

f ModeClock is always driven.

NANot applicable to input pins.

Table 8.7 R4650 Processor Signal Summary

Integrated Device Technology, Inc.

The Initialization Interface

Chapter 9

Introduction

This chapter describes the R4650 Initialization Interface,including the reset signal descriptions and types, initialization sequence, signals and timing dependencies, and boot modes, which are set at initialization time.

Signal names are listed in bold letters—for instance the signal **VCCOk** indicates the Vcc voltage is stable. Low-active signals are indicated by an asterisk at the end of the name, as in **ColdReset***.

Functional Overview

The R4650 processor has the following three types of resets. Refer to Figure 9.1 on page 4, Figure 9.2 on page 5, and Figure 9.3 on page 5 for timing diagrams of these resets.

- **Power-on reset**:Starts when the power supply is turned on and completely reinitializes the internal state machine of the processor without saving any state information.
- **Cold reset**:Restarts all clocks, but the power supply remains stable. A cold reset completely reinitializes the internal state machine of the processor without saving any state information.
- **Warm reset**:Restarts processor, but does not affect clocks. A warm reset preserves the processor internal state.

These resets use the **VCCOk**, **ColdReset***, and **Reset*** input signals, which are summarized in the next subsection. Descriptions of each type of reset operation is described.

The Initialization interface is a serial interface that operates at the frequency of the **MasterClock** divided by 256 (i.e. **MasterClock**/256). This low-frequency operation allows the initialization information to be stored in a low-cost Serial EEPROM.

Reset and Initialization Signal Descriptions

This section describes the three reset signals, **VCCOk, ColdReset***, and **Reset***, and the two initialization signals, **ModeIn** and **ModeClock**.

VCCOk: When asserted[1], **VCCOk** indicates to the processor that Vcc has been above the minimum Vcc for more than 100 milliseconds (ms) and is expected to remain stable. The assertion of **VCCOk** initiates the reading of the bcot-time mode control serial stream. This is described in the subsection "Initialization Sequence" on page 3.

ColdReset*: The **ColdReset*** signal must be asserted (low) for either a power-on reset or a cold reset. ColdReset* must be de-asserted synchronously with **MasterClock.**

Reset*: The **Reset*** signal must be asserted for any reset sequence. It can be asserted synchronously or asynchronously for a cold reset, or synchronously to initiate a warm reset. **Reset***must be de-asserted synchronously with **MasterClock**

ModeIn: Serial boot mode data in.

ModeClock: Serial boot mode data out, at the **MasterClock** frequency divided by 256 (**MasterClock**/256).

Table 9.1 lists the processor signals and their possible states.

[1] *Asserted* means the signal is true, or in its valid state. For example, the low-active **Reset*** signal is said to be asserted when it is in a low (true) state; the high-active **VCCOk** signal is true when it is asserted high.

Description	Name	I/O	Asserted State	3-State	Reset State
System address/data bus	**SysAD(63:0)**	I/O	High	Yes	a
System address/data check bus	**SysADC(7:0)**	I/O	High	Yes	a
System command/data identifier bus	**SysCmd(8:0)**	I/O	High	Yes	a
System command/data identifier bus parity	**SysCmdP**	I/O	High	Yes	a
Valid input	**ValidIn***	I	Low	No	NA
Valid output	**ValidOut***	O	Low	Yes	b
External request	**ExtRqst***	I	Low	No	NA
Release interface	**Release***	O	Low	Yes	b
Read ready	**RdRdy***	I	Low	No	NA
Write ready	**WrRdy***	I	Low	No	NA
Interrupts	**Int*(5:0)**	I	Low	No	NA
Nonmaskable interrupt	**NMI***	I	Low	No	NA
Boot mode data in	**ModeIn**	I	High	No	NA
Boot mode clock	**ModeClock**	O	High	No	d
Master clock	**MasterClock**	I	High	No	NA
Vcc is within specified range	**VCCOk**	I	High	No	NA
Cold reset	**ColdReset***	I	Low	No	NA
Reset	**Reset***	I	Low	No	NA

Key to Reset State Column:

a	All I/O pins (**SysAD[63:0]**, **SysADC[7:0]**, etc.) remain 3-stated until the **Reset*** signal deasserts.
b	All output only pins (**ValidOut***, **Release***, etc.), except the clocks, are 3-stated until the **ColdReset*** signal deasserts.
c	All clocks, except **ModeClock**, are 3-stated until **VCCOk** asserts.
d	**ModeClock** is always driven.
NA	Not applicable to input pins.

Table 9.1 R4650 Processor Signal Summary

Power-on Reset

Figure 9.1, Figure 9.2, and Figure 9.3 illustrate the power-on, cold, and warm resets.

The sequence for a power-on reset is as follows:

1. Power-on reset applies a stable Vcc of at least the Vcc minimum value to the processor. During this time, **VCCOk** is deasserted, **ColdReset*** and **Reset*** are asserted and the **MasterClock** input oscillates.
2. After at least 100 ms of stable Vcc and **MasterClock**, the **VCCOk** signal is asserted to the processor. The assertion of **VCCOk** begins the initialization of the processor. After the mode bits have been read in, the processor allows its internal phase locked loop to lock, stabilizing the processor internal clock, **PClock**.
3. **ColdReset*** is asserted for at least 64K (or 216) clock cycles after the assertion of VCCOk. Once the processor reads the boot-time mode control serial data stream, **ColdReset*** can be deasserted. **ColdReset*** must be deasserted synchronously with **MasterClock**.
4. After **ColdReset*** is deasserted synchronously, **Reset*** is deasserted to allow the processor to begin running. Reset* must be held asserted for at least 64 **MasterClock** cycles after the deassertion of **ColdReset***. **Reset*** must be deasserted synchronously with **MasterClock**.

Note: **ColdReset*** must be asserted when **VCCOk** asserts. The behavior of the processor is undefined if **VCCOk** asserts while **ColdReset*** is deasserted.

Cold Reset

A cold reset can begin anytime after the processor has read the initialization data stream, causing the processor to start with the Reset exception.

A cold reset requires the same sequence as a power-on reset except that the power is presumed to be stable before the assertion of the reset inputs and the deassertion of **VCCOk**.

To begin the reset sequence, **VCCOk** must be deasserted for a minimum of 100 ms before reassertion.

Warm Reset

To execute a warm reset, the **Reset*** input is asserted synchronously with **MasterClock**. It is then held asserted for at least 64 **MasterClock** cycles before being deasserted synchronously with MasterClock. The processor internal clock, **PClock**, is not affected by a warm reset. The boot-time mode control serial data stream is not read by the processor on a warm reset. A warm reset forces the processor to start with a Soft Reset exception.

MasterClock generates any 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 depends on whether or not a cache miss sequence has been interrupted by resetting the processor state machines.

Initialization Sequence

The boot-mode initialization sequence begins immediately after **VCCOk** is asserted. As the processor reads the serial stream of 256 bits through the **ModeIn** pin, the boot-mode bits initialize all fundamental processor modes. (The signals used are described in Chapter 8).

The initialization sequence is as follows:

1. The system deasserts the **VCCOk** signal. The **ModeClock** output is held asserted.
2. The processor synchronizes the **ModeClock** output at the time **VCCOk** is asserted. The first rising edge of **ModeClock** occurs at least 256 **MasterClock** cycles after **VCCOk** is asserted. There could be more clock cycles due to internal delays on the **VccOK** signal. After the first rising edge, each additional rising edge will be 256 master clock cycles.
3. Each bit of the initialization stream is presented at the **ModeIn** pin after each rising edge of the **ModeClock**. The processor samples 256 initialization bits from the **ModeIn** input.

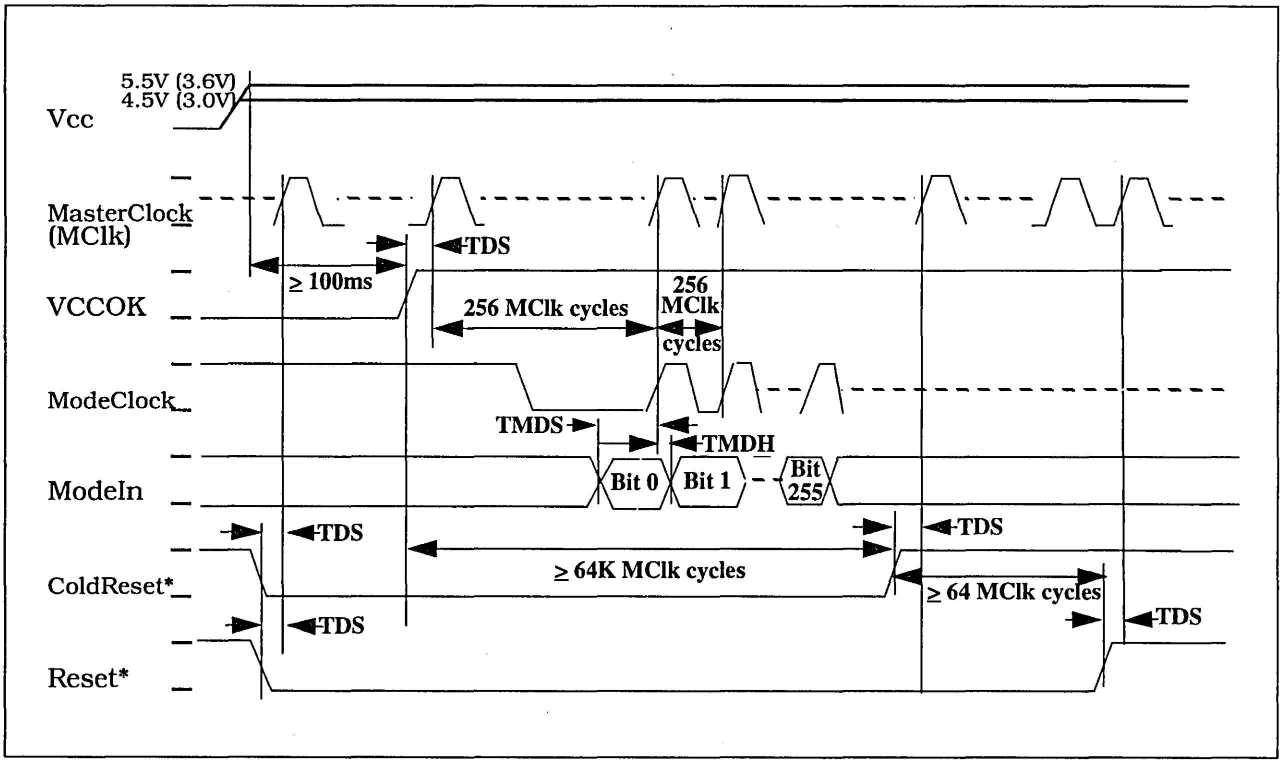


Figure 9.1 Power-on Reset

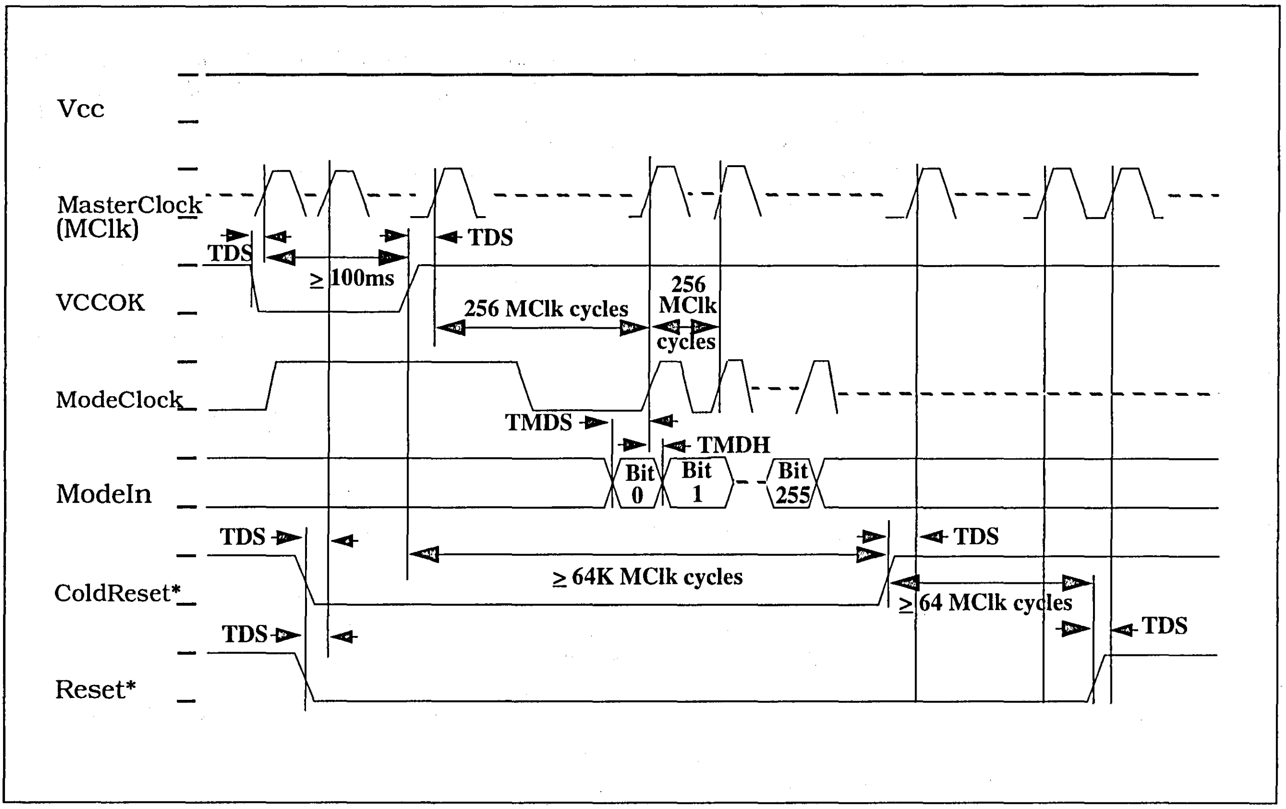


Figure 9.2 Cold Reset

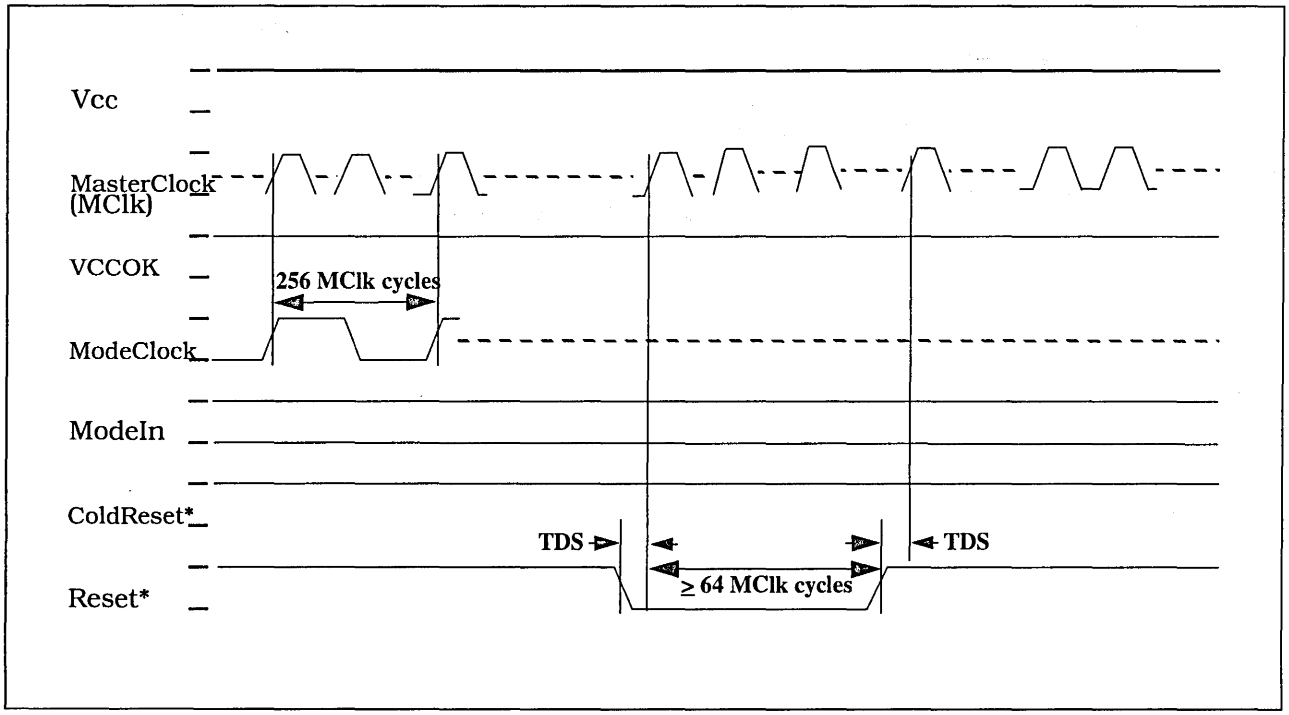


Figure 9.3 Warm Reset

Boot-Mode Settings

A number of processor operational parameters are determined statically at boot time. These include:

- Output driver slew rate
- Data writeback pattern
- System byte ordering
- **MasterClock** to **PClock** ratio
- Bus interface width.

Table 9.2 lists the processor boot-mode settings. The following rules apply to the settings in the table:

- Bit 0 of the stream is presented to the processor when **VCCOk** is first asserted.
- Selecting a reserved value results in undefined processor behavior.
- Bits 15 to 255 are reserved bits.
- Zeros must be scanned in for all reserved bits.

Serial Bit	Description	Value	Mode Setting
0	Reserved (must be zero)	0	
1:4	**Writeback data rate** System interface data rate for block writes only; bit 4 is most significant.	0	64-bit mode: DDDD 32-bit mode: WWWWWWWW
		1	64-bit mode: DDxDDx 32-bit mode: WWxWWxWWxWWx
		2	64-bit mode: DDxxDDxx 32-bit mode: WWxxWWxxWWxxWWxx
		3	64-bit mode: DxDxDxDx 32-bit mode: WxWxWxWxWxWxWxWx
		4	64-bit mode: DDxxxDDxxx 32-bit mode: WWxxxWWxxxWWxxxWWxxx
		5	64-bit mode: DDxxxxDDxxxx 32-bit mode: WWxxxxWWxxxxWWxxxxWWxxxx
		6	64-bit mode: DxxDxxDxxDxx 32-bit mode: WxxWxxWxxWxxWxxWxxWxxWxx
		7	64-bit mode: DDxxxxxxDDxxxxxx 32-bit mode: WWxxxxxxWWxxxxxxWWxxxxxxWWxxxxxx
		8	64-bit mode: DxxxDxxxDxxxDxxx 32-bit mode: WxxxWxxxWxxxWxxxWxxxWxxxWxxxWxxx
		9-15	Reserved
5:7	**Clock Multiplier** **MasterClock** is multiplied internally to generate PClock	0	Multiply by 2
		1	Multiply by 3
		2	Multiply by 4
		3	Multiply by 5
		4	Multiply by 6
		5	Multiply by 7
		6	Multiply by 8
		7	Reserved
8	**EndBit** Specifies byte ordering	0	Little-endian ordering
		1	Big-endian ordering
9:10	**Non-block write** Selects the manner in which non-block writes are handled; bit 10 is most significant	0	R4x00 compatible
		1	Reserved
		2	Pipelined Writes
		3	Write re-issue
11	**TmrIntEn** Disables the timer interrupt on Int*[5]	0	Enabled Timer Interrupt
		1	Disabled Timer Interrupt
12	**System interface bus width**	0	64-bit system interface
		1	32-bit system interface
13:14	**Drv_Out** Output driver slew rate control; bit 14 is most significant; affects only outputs that are not clocks.	10	100% strength (fastest)
		11	83% strength
		00	67% strength
		01	50% strength (slowest)
15:255	Reserved (must be zero)	0	

Key to Table:
D= Doubleword (64-bit data)
W= Word (32-bit data)			

Table 9.2 Boot-Mode Settings

Integrated Device Technology, Inc.

The Clock Interface

Chapter 10

Introduction

This chapter describes the clock signals ("clocks") used in the R4650 processor. The subject matter includes basic system clocks and system timing parameters.

Signal Terminology

The following terminology is used in this chapter (and throughout the book) when describing signals:

- *Rising edge* indicates a low-to-high transition.
- *Falling edge* indicates a high-to-low transition.
- *Clock-to-Q delay* is the amount of time it takes for a signal to move from the input of a device (*clock*) to the output of the device (*Q*).

Figure 10.1 and Figure 10.2 illustrate these terms.

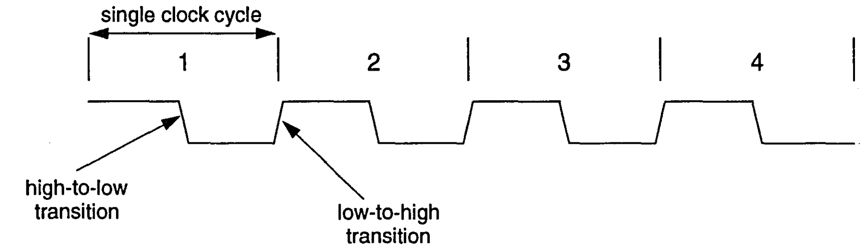


Figure 10.1 Signal Transitions

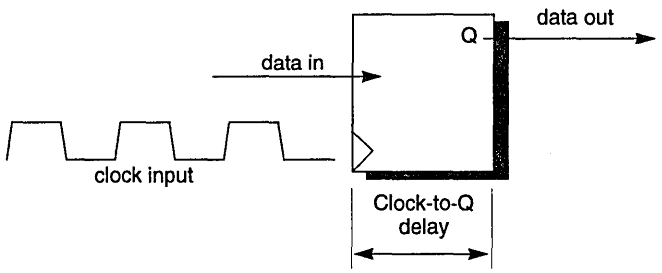


Figure 10.2 Clock-to-Q Delay

Basic System Clocks

The R4650 processor has a single input clock, **MasterClock**, and no output clocks.

MasterClock

The processor bases all internal and external clocking on the single **MasterClock** input signal. The R4650 uses **MasterClock** to sample data at the system interface and to clock data into the processor system interface output register. The external agent should use **MasterClock** for the global system clock and for clocking the output registers of an external agent.

PClock

The processor multiplies **MasterClock** by 2,3,4,5,6,7, or 8 to generate **PClock**. All internal registers and latches (except for **ModeClock**, which is part of the initialization interface) use **PClock**, which is the pipeline clock rate.

Figure 10.3 shows the clocks for a **MasterClock**-to-**PClock** multiply by 2.

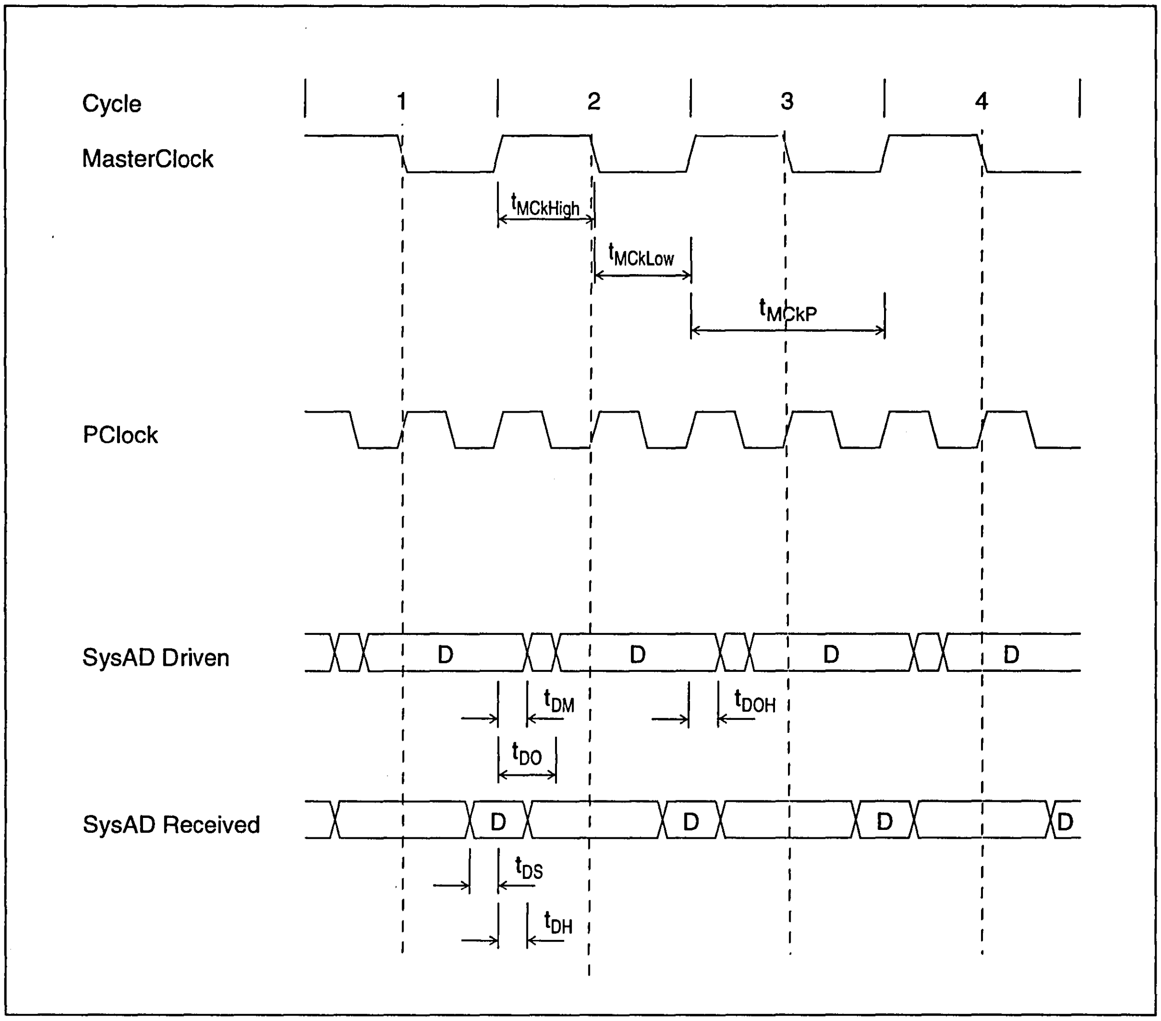


Figure 10.3 Processor Clocks, MasterClock- to-PClock Multiply by 2

System Timing Parameters

As shown in Figure 10.3, data provided to the processor must be stable a minimum of t_{DS} nanoseconds (ns) before the rising edge of **MasterClock** and be held valid for a minimum of t_{DH} ns after the rising edge of **MasterClock**.

Alignment to MasterClock

Processor data becomes stable a minimum of t_{DM} ns and a maximum of t_{DO} ns after the rising edge of **MasterClock**. This drive-time is the sum of the maximum delay through the processor output drivers together with the maximum clock-to-Q delay of the processor output registers. Processor data is held constant for a minimum of t_{DOH} ns after the rising edge of **MasterClock**. All processor inputs (including **VCCOk**, **ColdReset***, and **Reset***) are sampled based on **MasterClock**, and all outputs are based on **MasterClock**.

Phase-Locked Loop (PLL)

The processor aligns and generates **PClock** with internal phase-locked loop (PLL) circuits. By their nature, PLL circuits are only capable of generating aligned clocks for **MasterClock** frequencies within a limited range.

Clocks generated using PLL circuits contain some inherent inaccuracy, or *jitter*; a clock aligned with **MasterClock** by the PLL can lead or trail **MasterClock** by as much as the related maximum jitter specified in the data sheet.

PLL Components and Operation

The storage capacitor required for the Phase Locked Loop circuit is contained in the R4650. However, it is recommended that the system designer provide a filter network of passive components for the PLL power supply.

Passive Components

The Phase Locked Loop circuit requires several passive components for proper operation, which are connected to **Vcc**, **Vss**, **VccP**, and **VssP**, as illustrated in Figure 10.4.

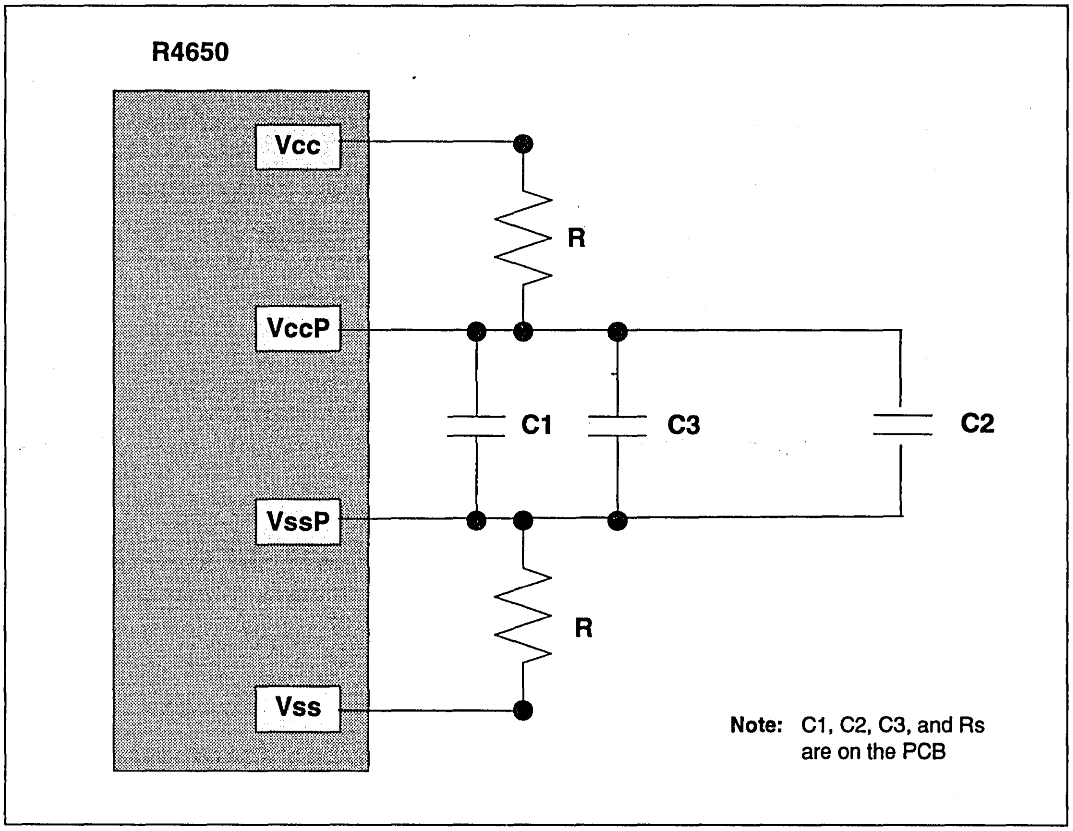


Figure 10.4 PLL Passive Components

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 = 1 nF
C2 = 82 nF
C3 = 10 μF
Cp = 470 pF
```

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 your specific application.

Connecting the R4650 to an External Agent

MasterClock is used to drive both the processor and the external agent. The R4650 uses **MasterClock** to drive its output buffer and to sample the input buffer. Similarly, the external agent should use **MasterClock** to sample its input buffers, drive its output buffer, and as the system clock.

In such a system, the delivery of data and data sampling have common characteristics, even if the processor and external agent have different delay values. For example, *transmission time* (the amount of time a signal takes to move from the processor to external agent to another along a trace on the board) can be calculated from the following equation:

Transmission Time = (MasterClock period)
– (t_{DO} for processor or external agent)
– (t_{DS} for external agent or processor)

Figure 10.5 shows a block-level diagram of a system using the R4650 processor.

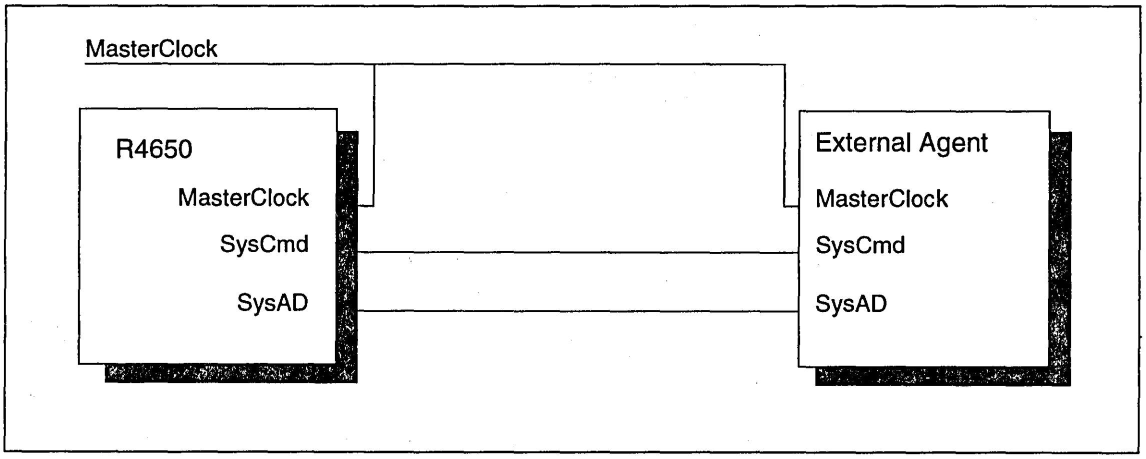


Figure 10.5 R4650 Processor System

Integrated Device Technology, Inc.

Cache Organization, Operation and Coherency

Chapter 11

Introduction

This chapter describes the on-chip cache memory, its place in the R4650 memory organization, and individual operations of the primary cache.

This chapter uses the following terminology:

- The primary cache may also be referred to as the P-cache.
- The primary data cache may also be referred to as the D-cache.
- The primary instruction cache may also be referred to as the I-cache.

These terms are used interchangeably throughout this book.

Memory Organization

Figure 11.1 shows the R4650 system memory hierarchy. In the logical memory hierarchy, caches lie between the CPU and main memory. They are designed to make the speedup of memory accesses transparent to the user.

Each functional block in Figure 11.1 has the capacity to hold more data than the block above it. For instance, physical main memory has a larger capacity than the primary cache.

At the same time, each functional block takes longer to access than any block above it. For instance, it takes longer to access data in main memory than in the CPU on-chip registers.

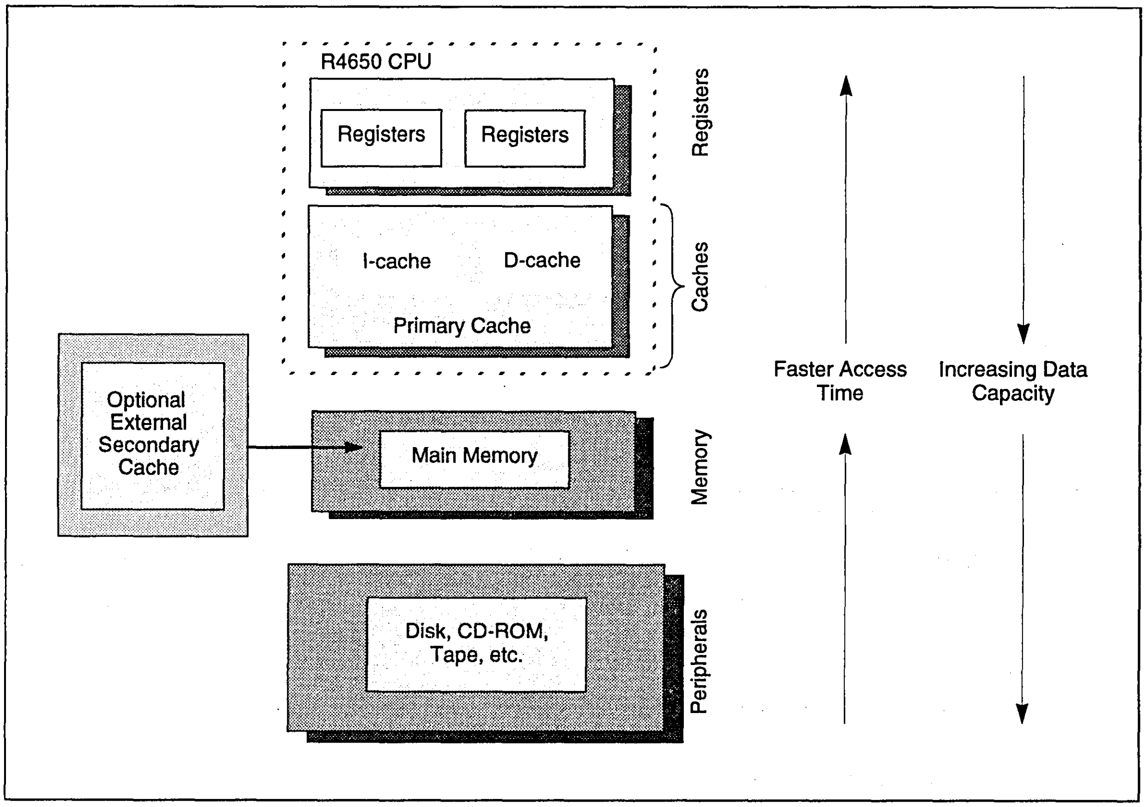


Figure 11.1 Logical Hierarchy of Memory

The R4650 processor has two on-chip primary caches. One holds instructions (the instruction cache), while the other holds data (the data cache).

Overview of Cache Operations

Caches provide fast temporary data storage, and they make the speedup of memory accesses transparent to the user. In general, the processor accesses cache-resident instructions or data through the following procedure:

1. The processor, through the on-chip cache controller, attempts to access the next instruction or data in the primary cache.
2. The cache controller checks to see if this instruction or data is present in the primary cache.
   - If the instruction/data is present, the processor retrieves it. This is called a primary-cache *hit.*
   - If the instruction/data is not present in the primary cache, it is retrieved as a cache line from memory and is written into the primary cache.
3. The processor retrieves the instruction/data from the primary cache and operation continues. For a data cache miss, the processor can restart the pipeline after the first doubleword (the one at the miss address) is retrieved and continues the cache line refill in parallel.

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

R4650 Cache Description

This section describes the organization of on-chip primary caches. As Figure 11.1 illustrates, the R4650 contains separate primary instruction and data caches.

Figure 11.2 provides a block diagram of the R4650 memory model.

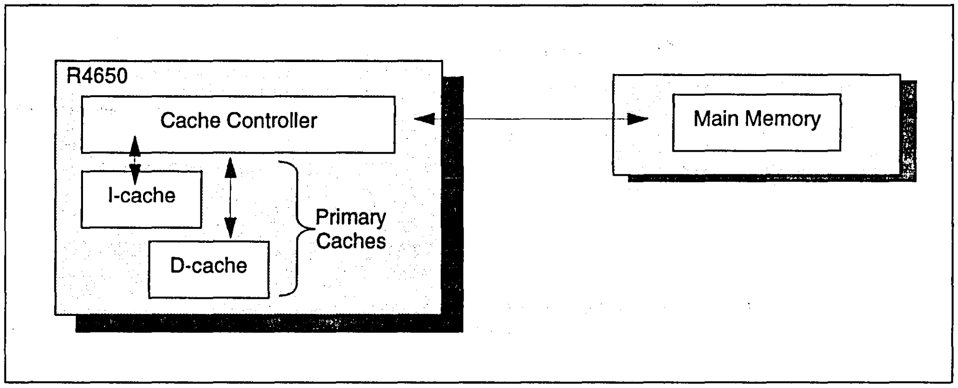


Figure 11.2 Cache Support in the R4650

Cache Line Size

A *cache line* is the smallest unit of information that can be fetched from memory to be filled into the cache. A primary cache line is 8 words in length and is represented by a single tag.

Upon a cache miss in the primary cache, the missing cache line is loaded from memory into the primary cache.

Cache Organization and Accessibility

This section describes the organization of the primary cache, including the manner in which it is mapped, the addressing used to index the cache, and composition of the cache lines. The primary instruction and data caches are indexed with a virtual address (VA).[1]

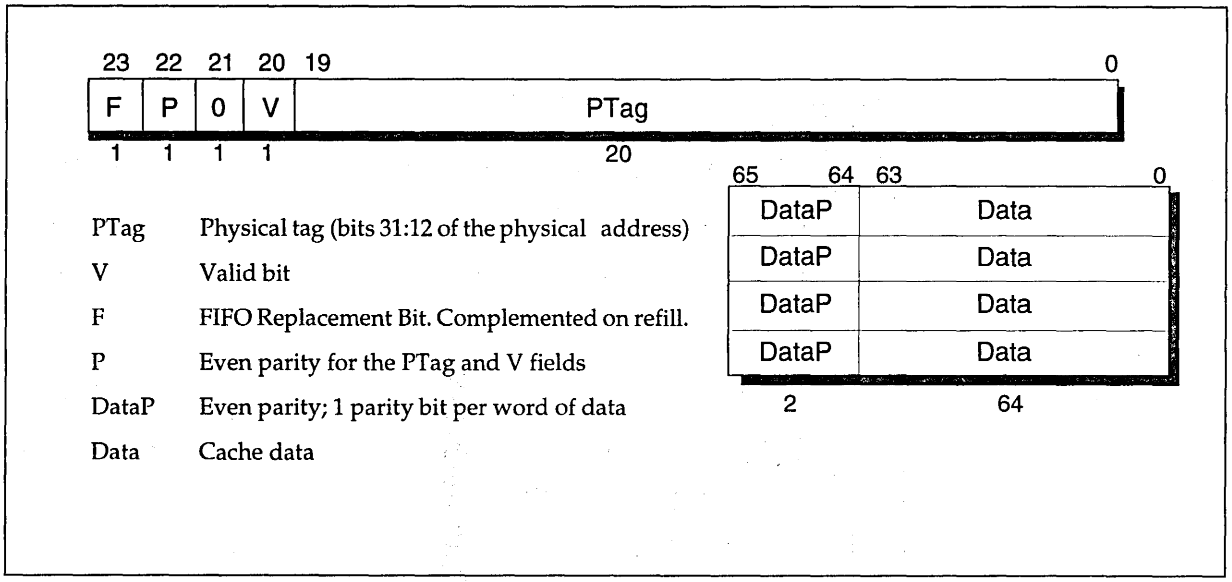
Organization of the Primary Instruction Cache (I-Cache)

Each line of primary I-cache data (although it is actually an instruction, it is referred to as data to distinguish it from its tag) has an associated 24-bit tag that contains a 20-bit physical address, a single valid bit, a reserved bit, a single parity bit and the FIFO replacement bit. Word parity is used on I-cache data.

The R4650 processor primary I-cache has the following characteristics:

- two-way set associative
- indexed with a virtual address
- checked with a physical tag
- organized with 8-word (32-byte) cache line
- lockable on a per-set basis

Figure 11.3 shows the format of a primary I-cache line.



PTag	Physical tag (bits 31:12 of the physical address)
V	Valid bit
F	FIFO Replacement Bit. Complemented on refill.
P	Even parity for the PTag and V fields
DataP	Even parity; 1 parity bit per word of data
Data	Cache data

Figure 11.3 R4650 Primary I-Cache Line Format

[1] Since the size of one set of primary caches is 4KB, the virtual offset equals the physical offset. Logically, however, the cache index is pre-translation, and thus considered virtual.

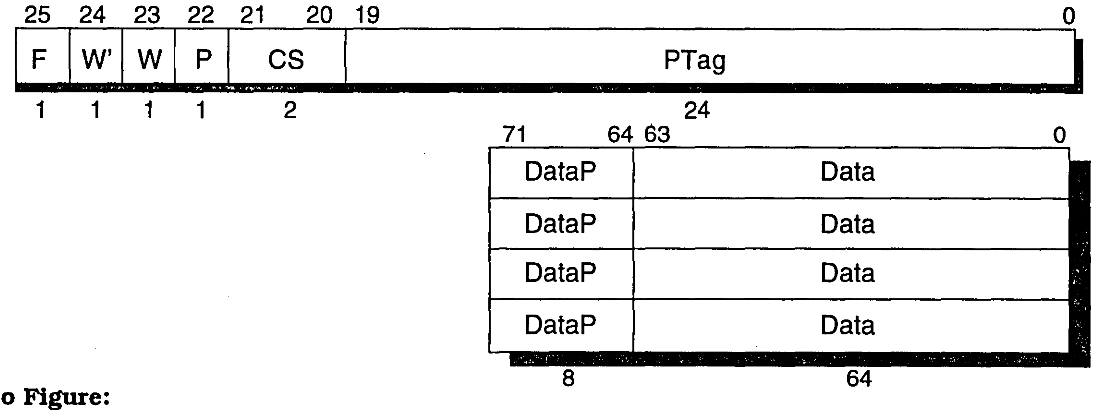
Organization of the Primary Data Cache (D-Cache)

Each line of primary D-cache data has an associated 26-bit tag that contains a 20-bit physical address, 2-bit cache line state, a write-back bit, a parity bit for the physical address and cache state fields, a parity bit for the write-back bit, and the FIFO replacement bit.

The R4650 processor primary D-cache has the following characteristics:

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

Figure 11.4 shows the format of a primary D-cache line.



Key to Figure:

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

Figure 11.4 R4650 8-Word Primary Data Cache Line Format

In the R4650, the *W* (write-back) bit, not the cache state, indicates whether or not the primary cache contains modified data that must be written back to memory

Note: There is no hardware support for cache coherency. The only cache states used are Dirty Exclusive and Invalid.

Accessing the Primary Caches

Figure .5 shows the virtual address (VA) index into the primary caches. Each instruction and data cache size is 8 Kbytes.

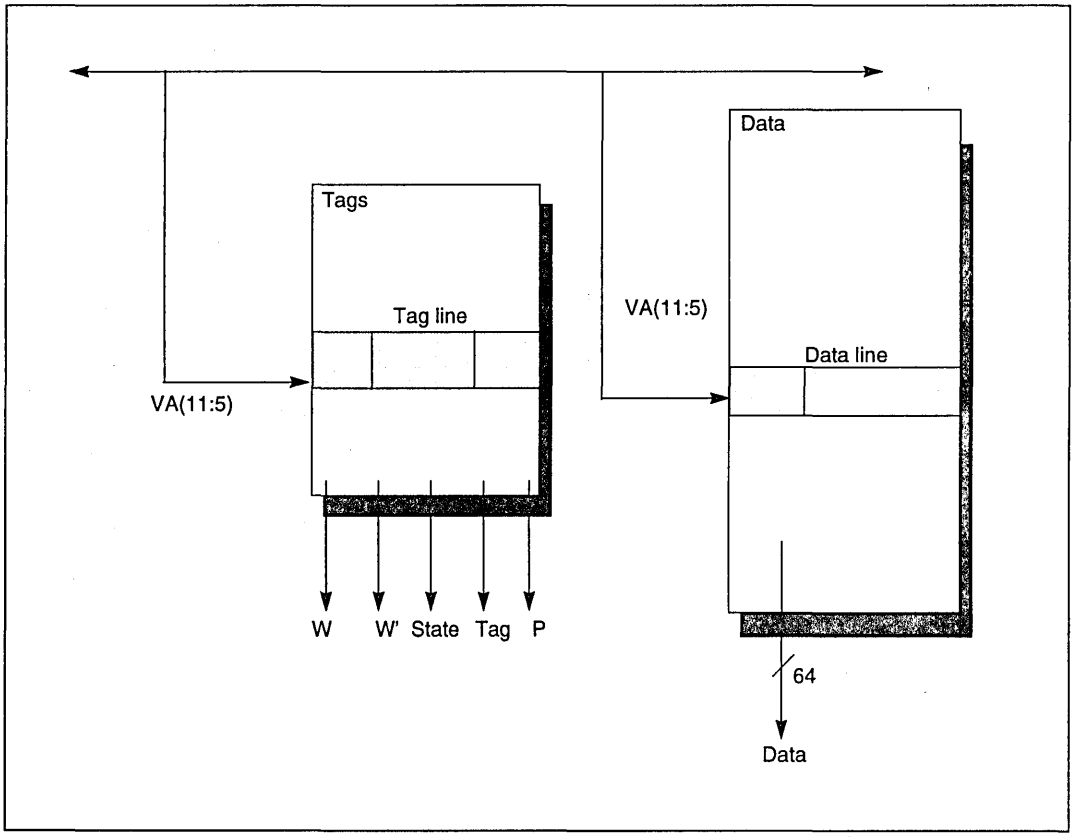


Figure 11.5 Primary Cache Data and Tag Organization

Cache States

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

- **Exclusive**: a cache line that is present in exactly one cache in the system is exclusive. This is always the case for the R4650. All cache lines are in an exclusive state.
- **Dirty**: a cache line that contains data that has changed since it was loaded from memory is dirty.
- **Clean**: a cache line that contains data that has not changed since it was loaded from memory is clean.
- **Shared**: a cache line that is present in more than one cache in the system. The R4650 does not provide for hardware cache coherency. This state will never happen in normal operations.

The R4650 only supports the four cache states as shown in Table 11.1 on page 11-6. The only states that will occur in the R4650, under normal operations are the Dirty Exclusive and Invalid states.

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

Each primary cache line in the R4650 system is in one of the states described in Table 11.1.

Cache Line State	Description
Invalid	A cache line that does not contain valid information must be marked invalid, and cannot be used. A cache line in any other state than invalid is assumed to contain valid information.
Shared	A cache line that is present in more than one cache in the system is shared. This state will not occur for normal operations.
Clean Exclusive	A clean exclusive cache line contains valid information and this cache line is not present in any other cache. The cache line is consistent with memory and is not owned by the processor (see "Cache Line Ownership" on page 6 in this chapter). This state will not occur for normal operations.
Dirty Exclusive	A dirty exclusive cache line contains valid information and is not present in any other cache. The cache line may or may not be consistent with memory and is owned by the processor (see "Cache Line Ownership" on page 6 in this chapter). Use the W bit to determine if the line must be written back on replacement.

Table 11.1 Cache States

Primary Cache States

Each primary data cache line is normally in one of the following states:

- invalid
- dirty exclusive

Each primary instruction cache line is in one of the following states:

- invalid
- valid

Cache Line Ownership

The processor is the owner of a cache line when it is in the dirty exclusive state, and is responsible for the contents of that line. There can only be one owner for each cache line.

The ownership of a cache line is set and maintained through the rules described below.

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

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

Cache Write Policy

The R4650 processor manages its primary data cache by using either a write-back or a write-through policy, determined by settings in the CP0 CAlg register. In a write-back cache, the data is not written back to memory until the cache line is replaced. A write-through policy means the store data is written to the cache and to memory. The write of the data to memory may not occur at the same time as the write to cache due to the write buffer.

For a write-back entry, if the cache line is valid and has been modified (the *W* bit is set), the processor writes this cache line back to memory when the line is replaced, either in the course of satisfying a cache miss or during the execution of a Write-back or Write-back Invalidate CACHE instruction.

For a write-through entry, whenever a store hits in the cache line, the data is also written to memory via the write buffer. The store will not set or clear the *W* bit for a write-through cache line. This allows a different virtual address that maps to the same physical address and with a write-back policy to set the *W* bit. For a miss to a write-through line, the action taken is determined by the write-allocation policy. For a write-allocate entry, the cache line is first retrieved from memory and the store continues. A no write-allocate entry posts the write to the system interface via the write buffer, in the same manner as an uncached write.

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, there are exceptions. For example, the processor retains a copy of the cache line if a cache line is written back by the Hit Write-back cache instruction. If the W bit is set, the cache line is written back and the W bit is cleared. The processor signals this line retention during a write by setting **SysCmd(2)** to a 1, as described in Chapters 12 and 14.

Cache State Transition Diagrams

The following sections describe the cache state diagrams that illustrate the cache state transitions for the primary cache. Figure .6 shows the state diagram of the primary cache.

When an external agent supplies a cache line, it need not return the initial state of the cache line, for normal operations (refer to Chapter 12 for a definition of an external agent). This is because the only read request the R4650 should issue are for non-coherent data and the lower three bits for the data identifier are reserved. The initial state will automatically be set to DE by the R4650. Otherwise, the processor changes the state of the cache line during one of the following events:

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

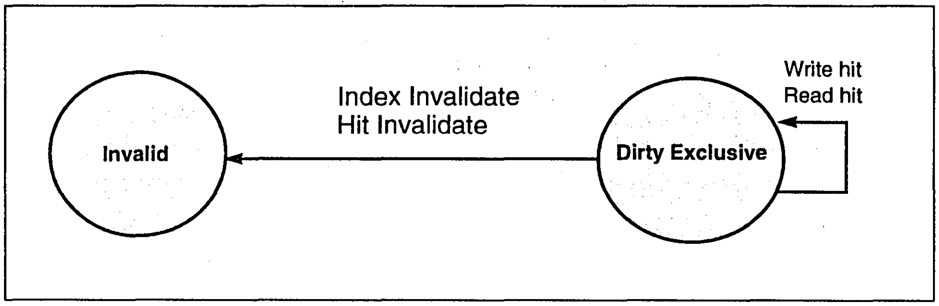


Figure 11.6 Primary Data Cache State Diagram

Cache Coherency Overview

Systems using more than one master must have a mechanism to maintain data consistency throughout the system. This mechanism is called a cache coherency protocol. The R4650 does not provide any hardware cache coherency. Cache coherency must be handled with software.

Cache Coherency Attributes

Cache coherency attributes are necessary to ensure the consistency of data throughout the system.

Bits in the CAlg register control coherency according to the virtual address. Specifically, the CAlg register contains 3 bits per entry that provide two possible coherency attribute types; they are listed below and described more fully in the following sections.

- uncached
- noncoherent (includes 3 attribute values)

Table 11.2 summarizes the behavior of the processor on load misses and store misses for each of the coherency attribute types listed above. The following sections describe in detail these coherency attribute types.

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

Table 11.2 Coherency Attributes and Processor Behavior

Uncached

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

Noncoherent

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

Cache Operation Modes

The R4650 processor only supports the no-secondary-cache mode (only uncached and noncoherent coherency attributes are applicable) of R4400 operation.

Cache Locking

The R4650 implements a feature referred to as "cache locking." That is, the kernel may set status register control bits that inhibit the cache refill process from displacing valid contents in set "A" of either cache. Note that these bits do not inhibit caches from being changed by any of the following operations or conditions:

- cache operations
- store operations to D-cache
- if they are invalid

Caches in the IDTR4650 RISC CPU are two-way set associative, just as they are in the Orion (R4600). Unlike the original R4600, they also support a cache-locking feature, which can be used to lock critical sections of code and/or data into on-chip caches for very fast access.

A cache is said to be *locked* when a particular piece of code or data is loaded into the cache and that cache location will not be selected later for refill by other data.

When To Use Cache Locking

Cache locking is useful in the following cases:

- a portion of code has to reside in cache permanently (*e.g.* time critical exception vectors) for real-time performance
- a given section of code is executed frequently and can fit inside the instruction cache
- a given section of data is accessed frequently and can fit inside the data cache (*e.g.* tables containing routing information in an embedded network application)

In the R4650, both Instruction cache and Data cache are 8KB. Each cache is two-way set associative with set A and set B. The size of each set is 4KB. On reset, both sets A and B are unlocked. By setting the DL or IL bit in the Status register of CP0, set A of the appropriate cache can be prevented from being chosen for refill on a cache miss, thus effectively locking the contents of the cache. The restriction on only set A being lockable is only for deterministic performance.

If both sets are invalid, the CPU always chooses set A. Similarly, data store operations to locked data update the D-cache contents; as above, locking merely prevents the cache line contents from being replaced by the contents of a different physical location. Otherwise, if a set is locked, its contents will not be changed.

An invalid line in a locked set will still be chosen for refill on a cache miss. Once refilled (and thus valid), this line will not be selected for refill until the appropriate lock bit is reset. This understanding, along with knowledge of Coprocessor 0 (CP0) hazards, can be used to develop a small and efficient algorithm for cache locking in the R4650.

The basic algorithm presented here consists of the following steps. Two examples follow the steps.

1. Invalidate the cache(s).
2. Set the appropriate cache lock bit(s).
3. Load the critical code/data into the cache(s).

Example of Data Cache Locking

Assume an example application in which there is a table that must always be kept in cache. In the startup code, after initialization of data structures, flushing of caches, etc., is done, the user can perform reads through cached addresses to load the data into the data cache, and then set the DL bit in the Status register to lock set A of the data cache.

Here is a sample code fragment for this example:

```
     .set noreorder
   jal      flush_cache          /* Flush caches */
   nop
   la       t0, critical_table   /* This table should always be in cache */
   li       t1, table_size       /* Size of table in bytes */
   li       t2, 0                /* Number of bytes read into cache */
1: lw       a0, 0(t0)
   addiu    t2, 4
   bneq     t2, t1, 1b           /* Loop back till done */
   addiu    t0, 4                /* bump read address */

   mfc0     a0, C0_SR            /* Get old SR value */
   li       a1, SR_DL            /* SR_DL = 0x00100000 */
   or       a0, a0, a1
   mtc0     a0, C0_SR            /* Set the Lock bit for data cache */
   nop
   nop
   nop                           /* 3 nops: safety against CP0 hazard */
```

Example of Instruction Cache Locking

Assume an example application in which there is a critical function that must always be kept in cache. Also assume that the size of the function is known. (If not known, you can find out the size by generating a disassembly of the object file.)

In the startup code, after initializing data structures, flushing of caches, etc., is done, you can perform the FILL operation in the CACHE instruction to fill the instruction cache with the critical function, and then set the IL bit in the Status register to lock set A of the instruction cache.

Here is a sample code fragment for this example:

```
    .set noreorder
    la      t0, 1f               /* Get address of label '1' */
    li      t1, 0xA0000000
    or      t0, t0, t0
    jr      t0                   /* Uncached execution from now onwards */
    nop
1:  jal     flush_cache
    nop
    la      t0, func_start_addr  /* Start address of critical code */
    li      t1, func_size        /* Critical code size */
    li      t2, 0                /* Number of words read into cache */
2:  cache   Fill_I, 0(t0)        /* Fill Operation */
    addiu   t2, 4
    bneq    t2, t1, 1b           /* Loop back till done */
    addiu   t0, 4                /* bump read address */

    mfc0    a0, C0_SR            /* Get old SR value */
    li      a1, SR_IL            /* SR_IL = 0x00080000 */
    or      a0, a0, a1
    mtc0    a0, C0_SR            /* Set Lock bit for instruction cache */
    nop
    nop
    nop
    nop
    nop                          /* 5 nops: safety against CP0 hazard */
la          v0, 3f
jr          v0
nop

3:                               /* Resume execution in mode as linked */
```

R4650 Processor Synchronization Support

In a multiprocessor system, it is essential that two or more processors working on a common task can execute without corrupting each other's subtasks. *Synchronization*, an operation that guarantees an orderly access to shared memory, must be implemented for a properly functioning multiprocessor system. Two of the more widely used methods are discussed in this section: test-and-set, and counter. Even though the R4650 does not support symmetric multi-processing (SMP), these are useful for multi-master and heterogenous multi-processing.

Test-and-Set

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

Figure 11.7 illustrates a test-and-set synchronization procedure that uses a semaphore; when the semaphore is set to 0, the shared data is unlocked, and when the semaphore is set to 1, the shared data is locked.

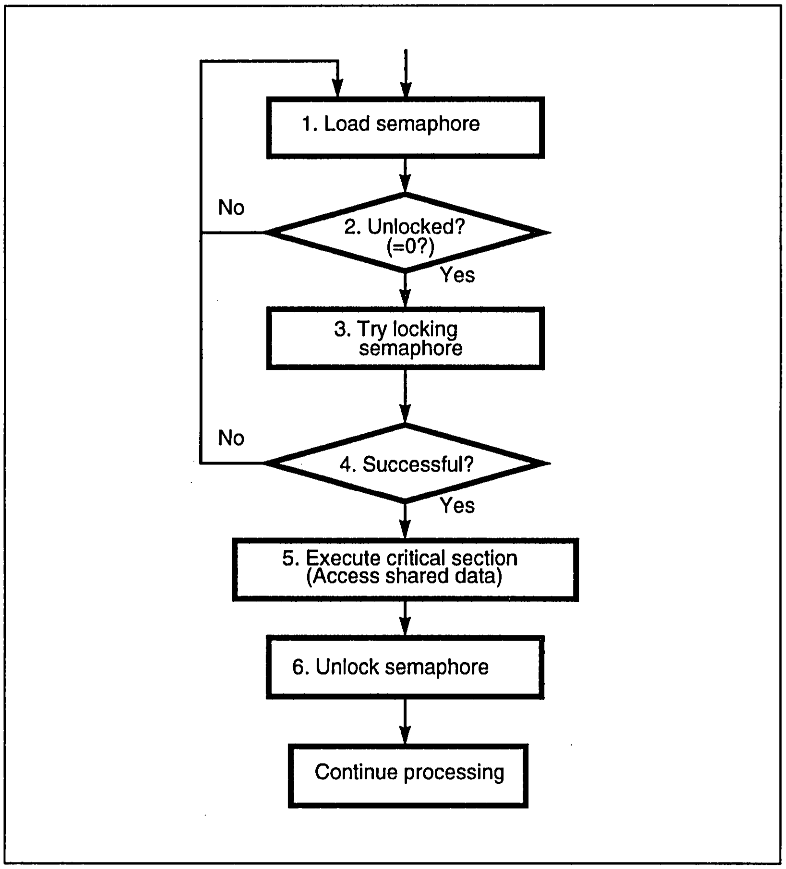


Figure 11.7 Synchronization with Test-and-Set

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

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

Counter

Another common synchronization technique uses a *counter. A counter is* a designated memory location that can be incremented or decremented.

In the test-and-set method, only one processor at a time is permitted to, enter the critical section. Using a counter, up to *N* processors are allowed to concurrently execute the critical section. All processors after the *N*th processor must wait until one of the *N* processors exits the critical section and a space becomes available.

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

Figure 11.8 shows this process.

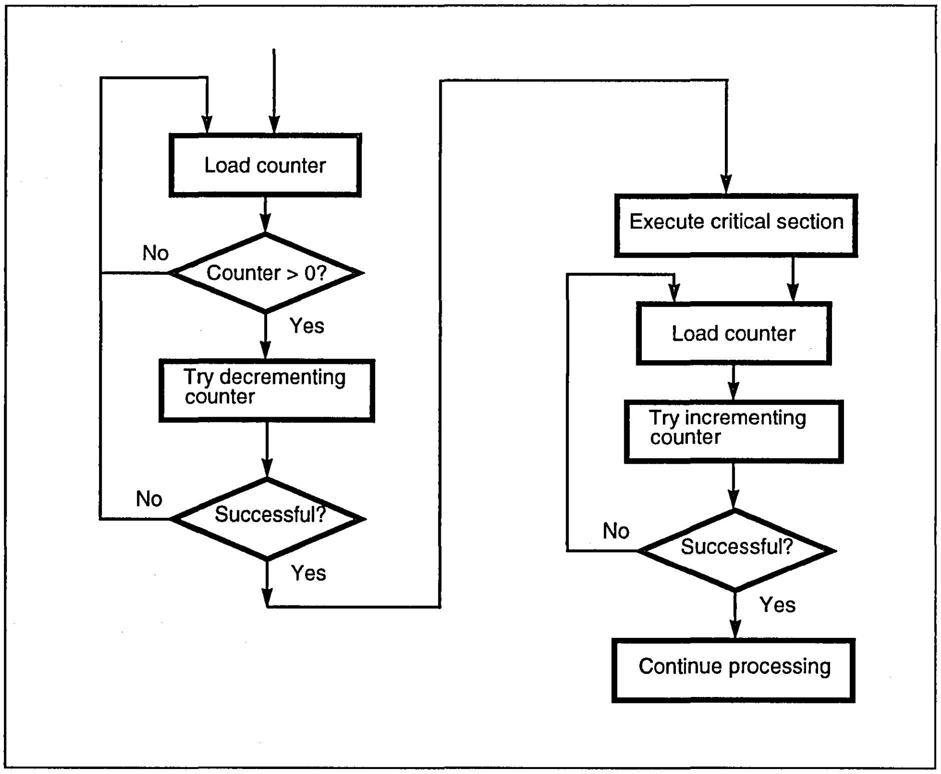


Figure 11.8 Synchronization Using a Counter

Load Linked and Store Conditional

The R4650 instructions *Load Linked* (LL) and *Store Conditional* (SC) provide support for processor synchronization. These two instructions work very much like their simpler counterparts, load and store. The LL instruction, in addition to doing a simple load, has the side effect of setting a bit called the *link bit.* This link bit forms a breakable link between the LL instruction and the subsequent SC instruction. The SC performs a simple store if the link bit is set when the store executes. If the link bit is not set, then the store fails to execute. The success or failure of the SC is indicated in the target register of the store.

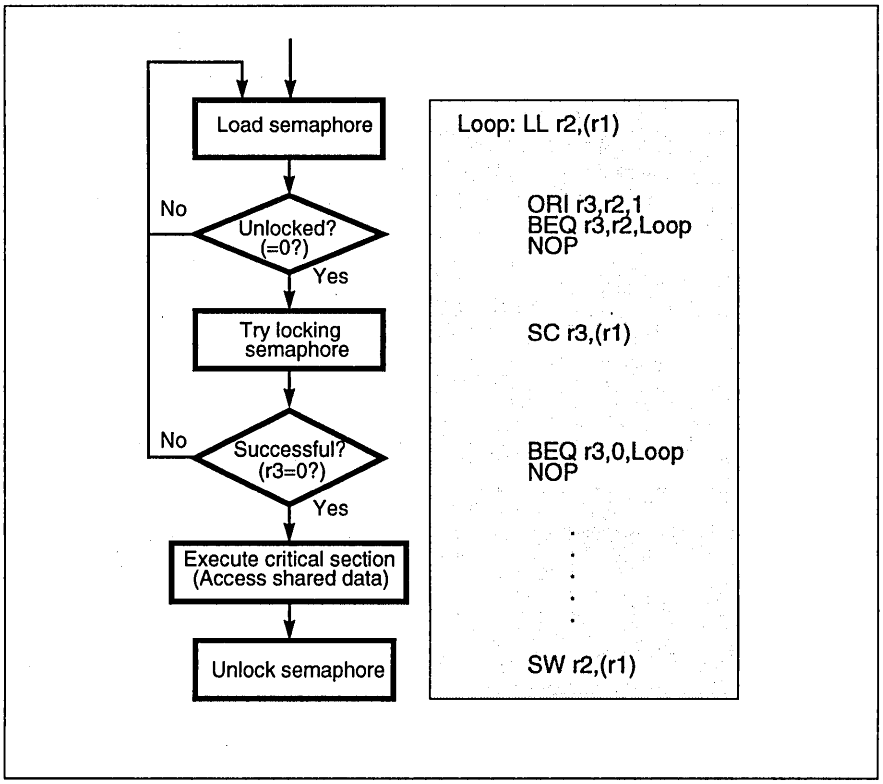
The link is broken upon completion of an ERET (return from exception) instruction.

The most important features of LL and SC are that:

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

Examples Using LL and SC

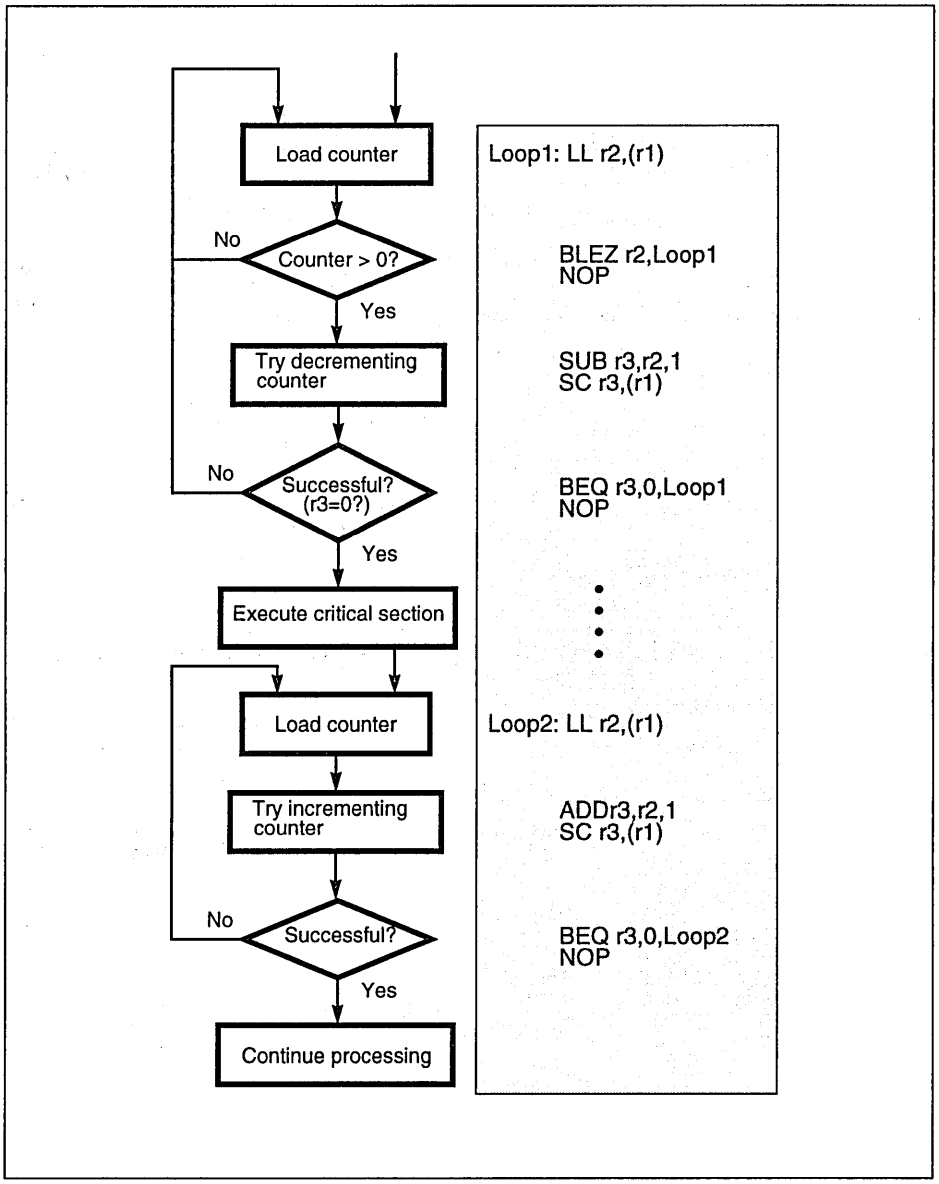
Figure 11.9 shows how to implement test-and-set using LL and SC instructions.



```
Loop: LL r2,(r1)

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

         SC r3,(r1)

         BEQ r3,0,Loop
         NOP

            .
            .
            .
            .
            .

         SW r2,(r1)
```

Figure 11.9 Test-and-Set using LL and SC

Figure 11.10 shows synchronization using a counter.



```
Loop1: LL r2,(r1)

            BLEZ r2,Loop1
            NOP

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

            BEQ r3,0,Loop1
            NOP

                 •
                 •
                 •
                 •

Loop2: LL r2,(r1)

            ADDr3,r2,1
            SC r3,(r1)

            BEQ r3,0,Loop2
            NOP
```

Figure 11.10 Counter Using LL and SC

Integrated Device Technology, Inc.

System Interface Overview

Chapter 12

Introduction

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

Terminology

The following terms are used in this chapter:

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

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

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

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

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

System Interface Description

The R4650 processor supports a 64-bit address/data interface that can construct a simple uniprocessor with main memory. The R4650 can be configured for a 32-bit external address/data interface as well.

The System interface consists of the following buses and signals:

- 64-bit address and data bus, **SysAD**
- 8-bit SysAD check bus, **SysADC** (even parity only)
- 9-bit command bus, **SysCmd**
- Six handshake signals:
  RdRdy*, **WrRdy***
  ExtRqst*, **Release***
  ValidIn*, **ValidOut***

The processor uses the system interface to access external resources in order to service processor requests such as cache misses, cache line write-backs, write-through stores and uncached operations.

Interface Buses

Figure 12.1 shows the primary communication paths for the system interface: a 64-bit address and data bus, **SysAD(63:0)**, and a 9-bit command bus, **SysCmd(8:0)**. These **SysAD** and the **SysCmd** buses are bidirectional; that is, they are driven by the processor to issue a processor request, and by the external agent to issue an external request.

A request through the system interface consists of:

- an address
- a System interface command that specifies the precise nature of the request
- a series of data elements if the request is for a write or read response.

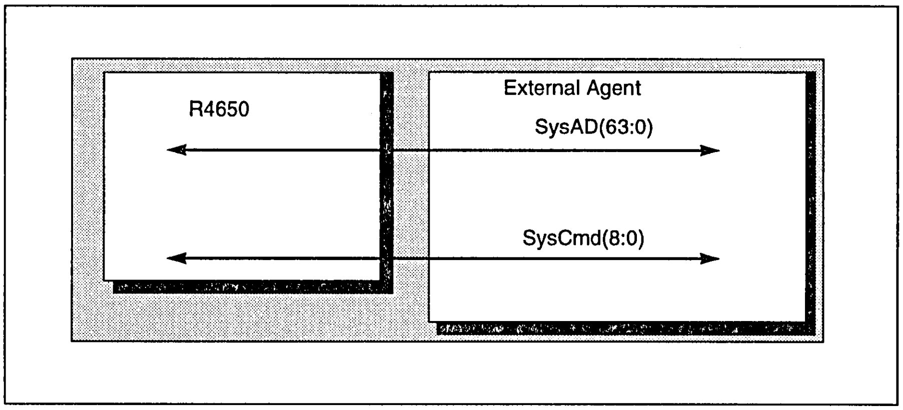


Figure 12.1 System Interface Buses

Address and Data Cycles

Cycles in which the **SysAD** bus contains a valid address are called *address cycles*. Cycles in which the **SysAD** bus contains valid data are called *data cycles*. Validity is determined by the state of the **ValidIn*** and **ValidOut*** signals.

The **SysCmd** bus identifies 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 [**SysCmd(8)** = 0], the remainder of the **SysCmd** bus, **SysCmd(7:0)**, contains a *System interface command*.
- During data cycles [**SysCmd(8)** = 1], the remainder of the **SysCmd** bus, **SysCmd(7:0)**, contains a *data identifier* .

Issue Cycles

The issue cycle is defined as the cycle when the external agent can accept the address issued from the processor. There are two types of processor issue cycles:

- processor read request issue cycles
- processor write request issue cycles.

The processor samples the signal **RdRdy*** to determine the *issue cycle* for a processor read request; the processor samples the signal **WrRdy*** to determine the *issue cycle* of a processor write request.

As shown in Figure 12.2, **RdRdy*** must be asserted for one clock cycle, two cycles prior to the address cycle of the processor read request to define the address cycle as the issue cycle (cycle 5 in Figure 12.2). **RdRdy*** does not need to be asserted during the issue cycle.

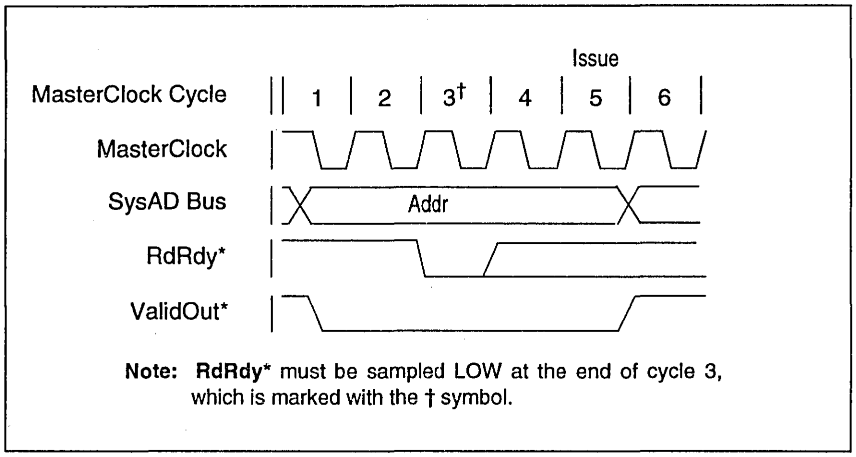

Issue
MasterClock Cycle
1
2
3†
4
5
6
MasterClock
SysAD Bus
Addr
RdRdy*
ValidOut*
Note: RdRdy* must be sampled LOW at the end of cycle 3, which is marked with the † symbol.

Figure 12.2 State of RdRdy* Signal for Read Requests

As shown in Figure 12.3, **WrRdy*** must be asserted for one clock cycle, two cycles prior to the first address cycle of the processor write request to define the address cycle as the issue cycle (cycle 5 in Figure 12.3). **WrRdy*** does not need to be asserted during the issue cycle.

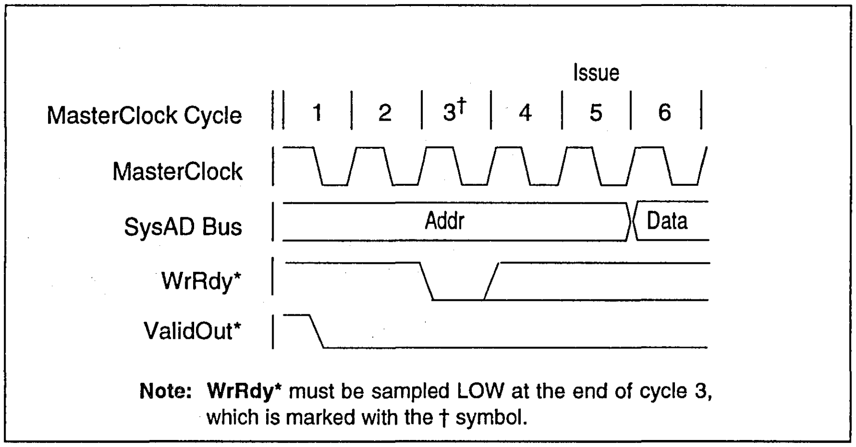

Issue
MasterClock Cycle
1
2
3†
4
5
6
MasterClock
SysAD Bus
Addr
Data
WrRdy*
ValidOut*
Note: WrRdy* must be sampled LOW at the end of cycle 3, which is marked with the † symbol.

Figure 12.3 State of WrRdy* Signal for Write Requests

The processor repeats the address cycle for the request (that is, asserts the valid address and the **ValidOut*** signal) until the conditions for a valid issue cycle are met. After the issue cycle, if the processor request requires data to be sent, the data transmission begins. There is only one issue cycle for any processor request.

The processor accepts external requests, even while attempting to issue a processor request, by releasing the system interface to slave state in response to an assertion of **ExtRqst*** by the external agent.

Note that the rules governing the issue cycle of a processor request are strictly applied to determine the action the processor takes. The processor either:

- completes the issuance of the processor request in its entirety before the external request is accepted, or
- releases the system interface to slave state without completing the issuance of the processor request.

In the latter case, the processor issues the processor request (provided the processor request is still necessary) after the external request is complete. The rules governing an issue cycle again apply to the processor request.

Handshake Signals

The processor manages the flow of requests through the following six control signals:

- **RdRdy***, **WrRdy*** are used by the external agent to indicate when it can accept a new read (**RdRdy***) or write (**WrRdy***) transaction.
- **ExtRqst***, **Release*** are used to transfer control of the **SysAD** and **SysCmd** buses. **ExtRqst*** is used by an external agent to indicate a need to control the interface. **Release*** is asserted by the processor when it transfers the mastership of the system interface to the external agent.
- The R4650 processor uses **ValidOut*** and the external agent uses **ValidIn*** to indicate valid command and data on the **SysCmd** and **SysAD** buses.

System Interface Protocols

Figure 12.4 shows the system interface operates from register to register. That is, processor outputs come directly from output registers and begin to change with the rising edge of **MasterClock.**[1]

Processor inputs are fed directly to input registers that latch these input signals with the rising edge of **MasterClock**. This allows the system interface to run at the highest possible clock frequency.

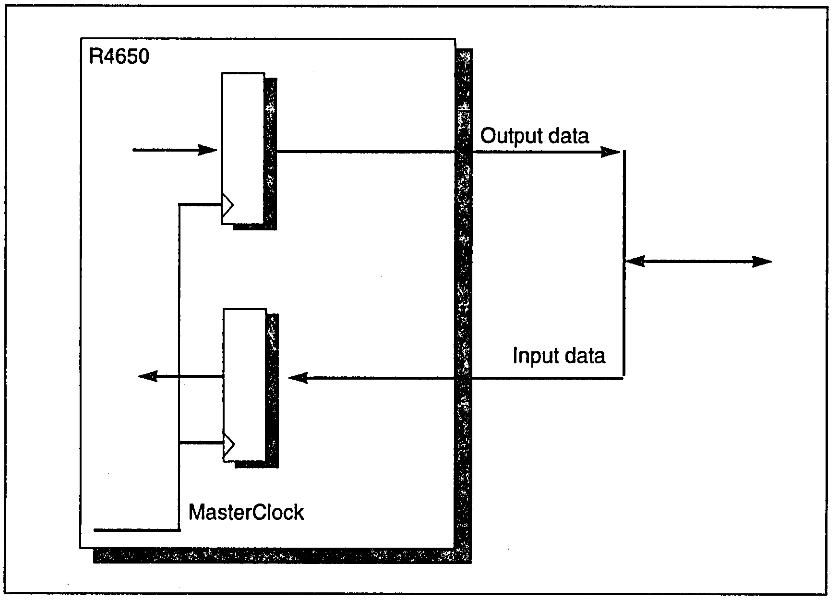


Figure 12.4 System Interface Register-to-Register Operation

Master and Slave States

When the R4650 processor is driving the **SysAD** and **SysCmd** buses, the system interface is in *master state.* When the external agent is driving the **SysAD** and **SysCmd** buses, the system interface is in *slave state.*

In master state, the processor drives the **SysAD** and **SysCmd** buses and will assert the signal **ValidOut*** whenever the information on these buses is valid.

In slave state, the external agent drives the **SysAD** and **SysCmd** buses and asserts the signal **ValidIn*** whenever the information on these buses is valid.

[1] **MasterClock** is the input clock to the processor.

Moving from Master to Slave State

The system interface remains in master state unless one of the following occurs:

- The external agent requests and is granted the system interface (external arbitration).
- The processor issues a read request and performs an uncompelled change to slave state.

External Arbitration

For the external agent to issue an external request through the system interface, the system interface must be in slave state. The transition from master state to slave state is arbitrated by the processor using the system interface handshake signals **ExtRqst*** and **Release***.

This transition is described by the following procedure:

1. An external agent signals that it wishes to issue an external request by asserting **ExtRqst***.
2. When the processor is ready to release bus mastership and accept an external request it asserts **Release*** for one cycle, which releases the system interface from master to slave state.
3. The system interface returns to master state as soon as the external request issue is complete.

This procedure is described in Chapter 15, "The External Request Interface."

Uncompelled Change to Slave State

An *uncompelled* change to slave state is the transition of the system interface from master state to slave state, initiated by the processor when a processor read request is pending. **Release*** is asserted automatically after a read request. An uncompelled change to slave state occurs during the issue cycle of a read request.

After an uncompelled change to slave state, the processor returns to master state at the end of the next external request. This can be a read response, or some other type of external request.

An external agent must note that the processor has performed an uncompelled change to slave state and begin driving the **SysAD** bus along with the **SysCmd** bus. As long as the system interface is in slave state, the external agent can begin a single external request without arbitrating for the system interface; that is, without asserting **ExtRqst***.

After the external request, the system interface returns to master state.

Whenever a processor read request is pending, after the issue of a read request, the processor automatically switches the system interface to slave state, even though the external agent is not arbitrating to issue an external request. This transition to slave state allows the external agent to quickly return read response data.

Processor and External Requests

There are two broad categories of requests: *processor requests* and *external requests*. These two categories are described in this section.

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

An external agent requesting access to a processor status register generates an *external request.* This access request passes through the system interface. System events and request cycles are shown in Figure 12.5.

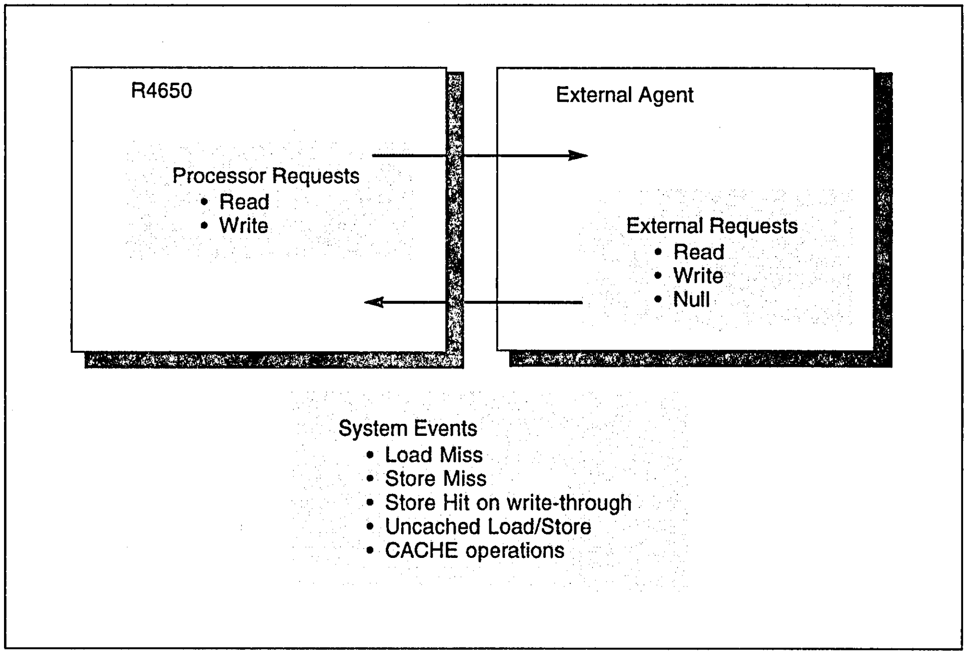


Figure 12.5 Requests and System Events

Rules for Processor Requests

The following rules apply to processor requests:

- After issuing a processor read request, the processor cannot issue a subsequent read request until it has received a read response.
- After the processor has issued a write request in R4x00 compatible write mode (set at boot time), the processor cannot issue a subsequent request until at least four cycles after the issue cycle of the write request. This means back-to-back write requests with a single data cycle are separated by two unused system cycles, as shown in Figure 12.6.

After the processor has issued a write request in either of the two new write modes, write reissue and pipelined writes, the processor can issue a subsequent write immediately provided the **WrRdy*** requirement is met. In Chapter 14, this is discussed in more detail.

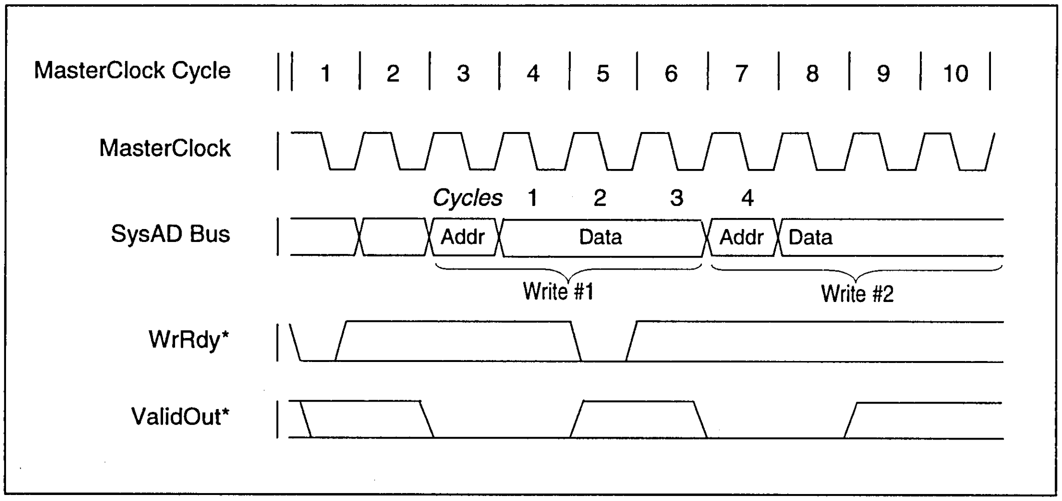


Figure 12.6 Back-to-Back Write Cycle Timing (R4000 compatible mode)

Processor Requests

A processor request is a request or a series of requests, through the system interface, to access some external resource. As shown in Figure 12.7, processor requests include only reads and writes.

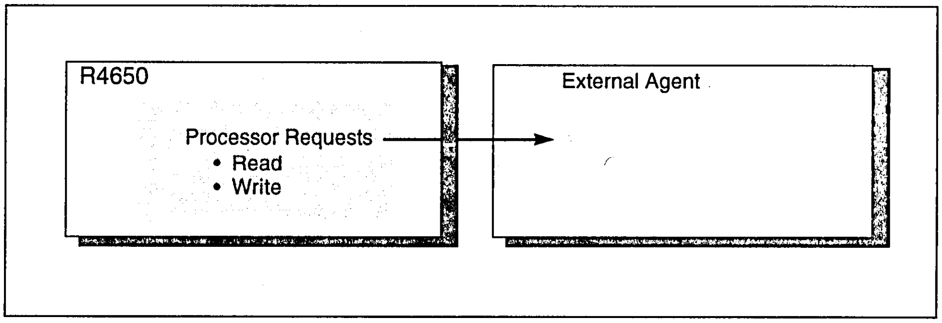


Figure 12.7 Processor Requests

Read request asks for a block, doubleword, partial doubleword, word, or partial word of data either from main memory or from another system resource.

Write request provides a block, doubleword, partial doubleword, word, or partial word of data to be written either to main memory or to another system resource.

Processor requests are managed by the processor in the equivalent of the R4400 *no-secondary-cache mode.*

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

The processor has the input signals **RdRdy*** and **WrRdy*** to allow an external agent to manage the flow of processor requests. **RdRdy*** controls the flow of processor read requests, while **WrRdy*** controls the flow of processor write requests.

The processor request cycle sequence is shown in Figure 12.8.

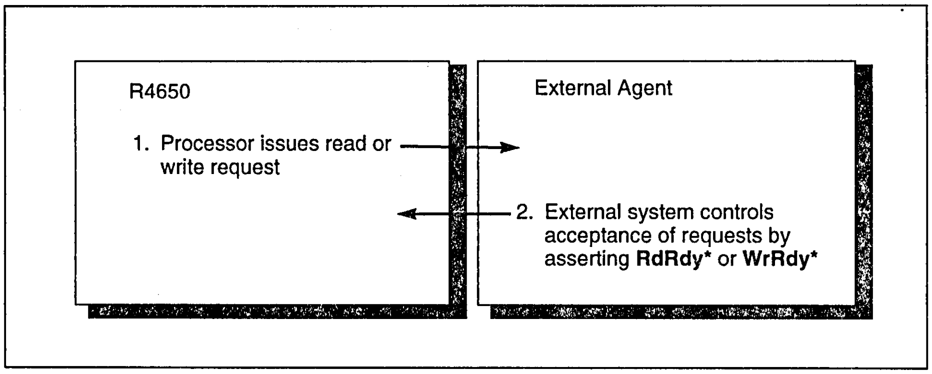

R4650
1. Processor issues read or write request
External Agent
2. External system controls acceptance of requests by asserting **RdRdy*** or **WrRdy***

Figure 12.8 Processor Request

Processor Read Request

When a processor issues a read request, the external agent must access the specified resource and return the requested data.

A processor read request can be split from the external agent's return of the requested data; in other words, the external agent can initiate an unrelated external request before it returns the response data for a processor read. A processor read request is completed after the last word of response data has been received from the external agent.

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

Processor read requests that have been issued, but for which data has not yet been returned, are said to be *pending.* A read remains pending until the requested read data is returned.

The external agent must be capable of accepting a processor read request any time the following two conditions are met:

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

Processor Write Request

When a processor issues a write request, the specified resource is accessed and the data is written to it.

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

The external agent must be capable of accepting a processor write request any time the following two conditions are met:

- No processor read request is pending.
- The signal **WrRdy*** has been asserted for one clock cycle, two cycles before the issue cycle.

The R4650 has added two new modes to enhance the throughput of non-block writes. These modes allow for 2 cycle throughput on back-to-back non-block writes. The actual protocol is discussed in Chapter 14, "The Write Interface." The external agent must be capable of accepting a processor write request in these modes under the same conditions as for the R4x00 compatibility mode (except as explained in Chapter 14, "The Write Interface").

External Requests

External requests include read, write and null requests, as shown in Figure 12.9. This section also includes a description of read response, a special case of an external request.

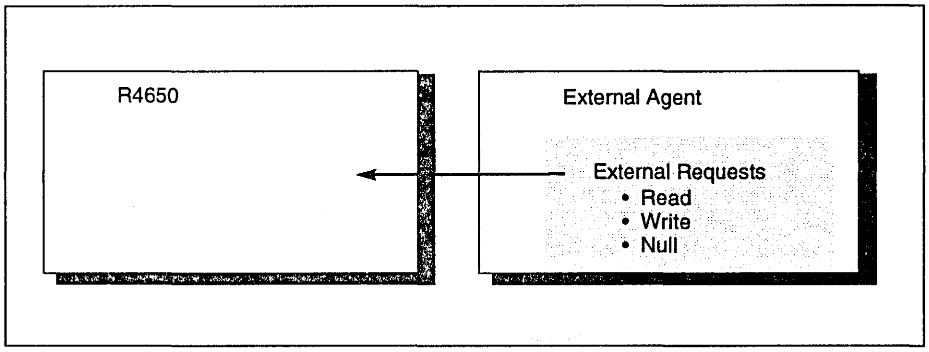


Figure 12.9 External Requests

Read request asks for a word of data from the processor's internal resource.

Write request provides a word of data to be written to the processor's internal resource.

Null request requires no action by the processor; it provides a mechanism for the external agent to return control of the system interface to the master state without affecting the processor.

The processor controls the flow of external requests through the arbitration signals **ExtRqst*** and **Release***, as shown in Figure 12.10. The external agent must acquire mastership of the system interface before it is allowed to issue an external request; the external agent arbitrates for mastership of the system interface by asserting **ExtRqst*** and then waiting for the processor to assert **Release*** for one cycle.

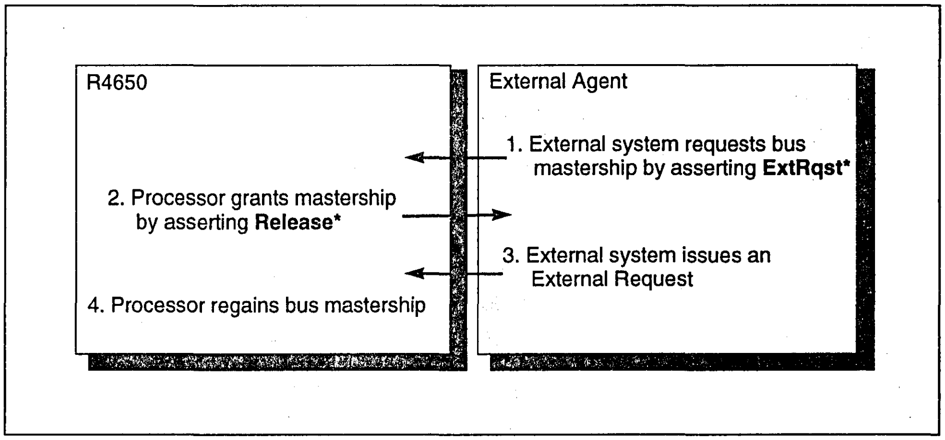

R4650
External Agent
1. External system requests bus mastership by asserting ExtRqst*
2. Processor grants mastership by asserting Release*
3. External system issues an External Request
4. Processor regains bus mastership

Figure 12.10 External Request

Mastership of the system interface always returns to the processor after an external request is issued. The processor does not accept a subsequent external request until it has completed the current request.

If there are no processor requests pending, the processor decides, based on its internal state, whether to accept the external request, or to issue a new processor request. The processor can issue a new processor request even if the external agent is requesting access to the system interface.

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

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

External Read Request

In contrast to a processor read request, data is returned directly in response to an external read request; no other requests can be issued until the processor returns the requested data. An external read request is complete after the processor returns the requested word of data.

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

Note: The R4650 does not contain any resources that are readable by an external read request; in response to an external read request the processor returns undefined data and a data identifier with its *Erroneous Data* bit, **SysCmd(5)**, set. Thus, the R4650 will take a bus error at the completion of the external read request.

External Write Request

When an external agent issues a write request, the specified resource is accessed and the data is written to it. An external write request is complete after the word of data has been transmitted to the processor.

The only processor resource available to an external write request is the IP field of the Cause register.

System Interface Endianness

The endianness of the system interface is programmed at boot time through the boot-time mode control interface (see Chapter 9, "Initialization Interface" for specifics), and remains fixed until the next time the processor boot-time mode bits are read. Software cannot change the endianness of the system interface and the external system; software can set the reverse endian bit to reverse the interpretation of endianness inside the processor, but the endianness of the system interface remains unchanged.

System Interface Cycle Time

The processor specifies minimum and maximum cycle counts for various processor transactions and for the processor response time to external requests. Processor requests themselves are constrained by the system interface request protocol, and request cycle counts can be determined by examining the protocol.

The following system interface interactions can vary within minimum and maximum cycle counts:

- waiting period for the processor to release the system interface to slave state in response to an external request (*release latency*)
- response time for an external request that requires a response (*external response latency*).

The remainder of this section describes and tabulates the minimum and maximum cycle counts for these system interface interactions.

Release Latency

Release latency is generally defined as the number of cycles the processor can wait to release the system interface to slave state for an external request. When no processor requests are in progress, internal activity can cause the processor to wait some number of cycles before releasing the system interface. Release latency is therefore more specifically defined as the number of cycles that occur between the assertion of **ExtRqst*** and the assertion of **Release***.

There are three categories of release latency:

- Category 1: When the external request signal is asserted two cycles before the last cycle of a processor request.
- Category 2: When the external request signal is not asserted during a processor request, or is asserted during the last cycle of a processor request.
- Category 3: When the processor makes an uncompelled change to slave state.

Table 12.1 summarizes the minimum and maximum release latencies for requests that fall into categories 1, 2 and 3. Note that the maximum and minimum cycle count values are subject to change.

Category	Minimum PCycles	Maximum PCycles
1	4	6
2	4	24
3	0	0

Table 12.1 Release Latency for External Requests

The differences in the minimum and maximum times are due to internal conditions not readily observable externally. The relationship between **PClock** and **MasterClock** will dictate when the **Release*** signal is seen externally.

64-Bit System Interface Addresses

System interface addresses are full 32-bit physical addresses presented on the least-significant 32 bits (bits 31 through 0) of the **SysAD** bus during address cycles; the remaining bits of the **SysAD** bus are unused during address cycles.

Addressing Conventions for 64-Bit Wide Interface

Addresses associated with doubleword, partial doubleword, word, or partial word transactions, are aligned for the size of the data element. The system uses the following address conventions:

- Addresses associated with block requests are aligned to double-word boundaries; that is, the low-order 3 bits of address are 0.
- Doubleword requests set the low-order 3 bits of address to 0.
- Word requests set the low-order 2 bits of address to 0.
- Halfword requests set the low-order bit of address to 0.
- Byte, tribyte, quintibyte, sextibyte, and septibyte requests use the byte address.

32-Bit System Interface Addresses

System interface addresses are 32-bit physical addresses presented on the least-significant 32 bits (bits 31 through 0) of the **SysAD** bus during address cycles; the remaining bits of the **SysAD** bus are unused during address cycles.

Addressing Conventions for 32-Bit Wide Interface

Addresses associated with doubleword, partial doubleword, word, or partial word transactions, are aligned for the size of the data element. The system uses the following address conventions:

- Addresses associated with block requests are aligned to word boundaries; that is, the low-order 2 bits of address are 0.
- Word requests set the low-order 2 bits of address to 0.
- Halfword requests set the low-order bit of address to 0.
- Byte and tribyte requests use the byte address.

Integrated Device Technology, Inc.

The Read Interface Chapter 13

Introduction

This chapter discusses specifics of the read interface and read operations.

When a processor issues a read request, the external agent must access the specified resource and return the requested data. A processor read request can be split from the external agent's return of the requested data; in other words, the external agent can initiate an unrelated external request before it returns the response data for a processor read. A processor read request is completed after the last word of response data has been received from the external agent.

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

Processor read requests that have been issued, but for which data has not yet been returned, are said to be *pending*. A read remains pending until the requested read data is returned.

The external agent must be capable of accepting a processor read request any time the following two conditions are met:

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

Read Response

A *read response* returns data in response to a processor read request, as shown in Figure 13.1. While a read response is technically an external request, it has one characteristic that differentiates it from all other external requests—it does not perform system interface arbitration. For this reason, read responses are handled separately from all other external requests, and are simply called read responses. When a read response comes back with bad parity for the first data, a cache error exception results.

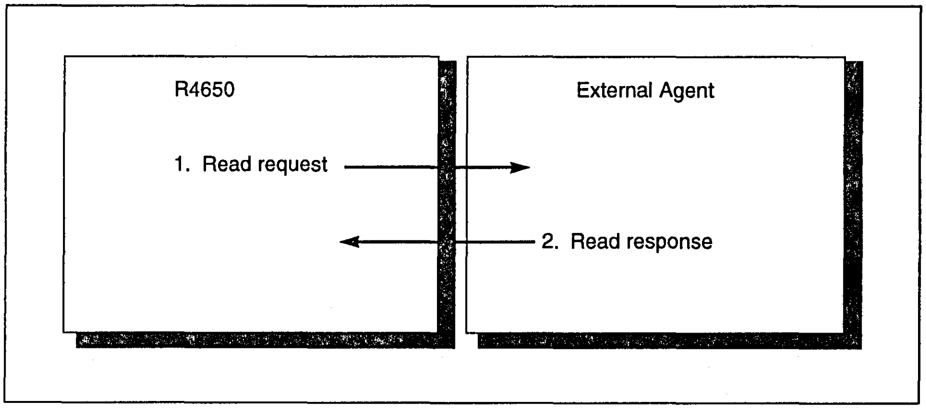


Figure 13.1 Read Response

Handling Requests

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

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

Load Miss

When a processor load misses in the primary cache, before the processor can proceed it must obtain the cache line that contains the data element to be loaded from the external agent.

If the new cache line replaces a current cache line with a W bit set, the current cache line must be written back.

The processor examines the coherency attribute in the CAlg register for the memory region that contains the requested cache line, and executes a noncoherent read request; the coherency attribute is *noncoherent* shows the actions taken on a load miss to primary cache.

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

Table 13.1 Load Miss to Primary Cache

If the cache line must be written back on a load miss, the read request is issued and completed before the write request is handled. The processor takes the following steps:

1. The processor issues a noncoherent read request for the cache line that contains the data element to be loaded.
2. The processor then waits for an external agent to provide the read response.
3. The processor will restart the pipeline after the first doubleword (the data that missed is fetched first). The rest of the data cache line will be placed into the cache in parallel.

If the current cache line must be written back, the processor issues a write request to save the dirty cache line in memory.

In 64-bit bus mode a block transfer (read or write) is equivalent to 4 data transfer to/from the memory. In 32-bit mode a block transfer (read or write) is equivalent to 8 data transfer to/from the memory.

Store Miss

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

The processor then executes one of the following requests:

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

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

Page Attribute	State of Data Cache Line Being Replaced	
	Clean/Invalid	**Dirty (W=1)**
Noncoherent, write-back or Noncoherent, write-through with write-allocate	NCR	NCR/W
Noncoherent, write-through with no write-allocate	NCW	NA

Table Legend:

NCR	Processor noncoherent block read request
NCR/W	Processor noncoherent block read request followed by processor block write request
NCW	Processor noncoherent write request

Table 13.2 Store Miss to Primary Cache

If the coherency attribute is write-back or write-through with write-allocate, the processor issues a read request for the cache line that contains the data element to be loaded, then waits for the external agent to provide read data in response to the read request. Then, if the current cache line must be written back, the processor issues a write request for the current cache line. For a write-through, no write-allocate store miss, the processor issues a write request only.

If the new cache line replaces a current cache line whose *Write back* (*W*) bit is set, the current cache line moves to an internal write buffer before the new cache line is loaded in the primary cache.

In 64-bit bus mode a block transfer (read or write) is equivalent to 4 data transfer to/from the memory. In 32-bit mode a block transfer (read or write) is equivalent to 8 data transfer to/from the memory.

Store Hit

This section describes store hits in no-secondary-cache mode for both write-back and write-through lines.

The action on the system interface will be determined by whether the line is write-back or write-through. All lines that use a write-back policy are set to the dirty exclusive cache state and there is no bus transaction generated. For lines with a write-through policy, the store will also generate a processor write request for the store data.

In 64-bit bus mode this is equivalent to 4 data transfer to the memory. In 32-bit mode this is equivalent to 8 data transfer to the memory.

Uncached Loads

When the processor performs an uncached load, it issues a noncoherent word read request (the actual access can be for a doubleword, word, partial word or byte, but the request is called a word read request to differentiate it from the block read request).

In 64-bit mode the CPU expects valid parity and data in the full **SysAD** bus (all 64 bits), even if it is looking for less than a double word. If a partial word is returned the correct parity for the full 64-bit must be returned, or the CPU must be informed not to check parity.

In 32-bit bus mode the CPU expects valid parity and data in the full **SysAD** bus (all 32 bits), even if it is looking for less than a word. If a partial word is returned the correct parity for the full 32-bit must be returned, or the CPU must be informed not to check parity.

All writes by the processor will be buffered from the system interface by the 4-deep write buffer. The write requests are sent to the system interface when there are no other requests in progress. If the write buffer contains any entries when a block request is needed, the write buffer is first flushed before any read request will occur (cache miss or uncached load).

Both a data cache miss and an uncached data load will flush the write buffer.

CACHE Operations

The processor provides a variety of CACHE operations to maintain the state and contents of the primary cache. During the execution of the CACHE operation instructions, the processor can issue write or read requests.

Load Linked/Store Conditional Operation

Generally, the execution of a Load Linked/Store Conditional instruction sequence is not visible at the system interface; that is, no special requests are generated due to the execution of this instruction sequence.

However, there is one situation in which the execution of a Load Linked/Store Conditional instruction sequence is visible, as indicated by the *link address retained* bit during a processor read request, as programmed by the **SysCmd(2)** bit. This occurs when the data location targeted by a Load-Linked-Store-Conditional instruction sequence maps to the same cache line to which the instruction area containing the Load Linked/Store Conditional code sequence is mapped. In this case, immediately after executing the Load Linked instruction, the cache line that contains the link location is replaced by the instruction line containing the code. The link address is kept in a register separate from the cache, and remains active as long as the *link* bit, set by the Load Linked instruction, is set.

The *link* bit, which is set by the load linked instruction, is cleared by a change of cache state for the line containing the link address, or by a Return From Exception.

For more information, refer to Chapter 11, or see the specific Load Linked and Store Conditional instructions described in Appendix A.

Processor Read Protocols

The following sections contain a cycle-by-cycle description of the bus arbitration protocols for the processor read request. Table 13.3 lists the abbreviations and definitions for each of the buses used in the timing diagrams that follow.

Scope	Abbreviation	Meaning
Global	Unsd	Unused
SysAD bus	Addr	Physical address
	Data<n>	Data element number n of a block of data
SysCmd bus	Cmd	An unspecified system interface command
	Read	A processor or external read request command
	Write	A processor or external write request command
	SINull	A system interface release external null request command
	NData	A noncoherent data identifier for a data element other than the last data element
	NEOD	A noncoherent data identifier for the last data element

Table 13.3 System Interface Requests

Processor Read Request

In the timing diagrams in this section note that the two closely spaced, wavy vertical lines (for example, MasterClock Cycle 2 in Figure 13.5 on page 13-12) indicate one or more identical cycles.

Processor Read Request Protocol Steps

The following sequence describes the protocol for a processor read request. This protocol is the same for either 32-bit bus mode or 64-bit bus mode. The numbered steps in this list correspond to the numbers in Figure 13.2.

1. **RdRdy*** is asserted low, indicating the external agent is ready to accept a read request.
2. With the system interface in master state, a processor read request is issued by driving a read command on the **SysCmd** bus and a read address on the **SysAD** bus.
3. At the same time, the processor asserts **ValidOut*** for one cycle, indicating valid data is present on the **SysCmd** and the **SysAD** buses.

   Note: Only one processor read request can be pending at a time. **ValidOut*** is asserted every time the CPU is driving valid information on **SysAD** and **SysCmd** bus. In the case of read request, this means as long as the address is driven and will be deasserted at the end of the bus cycle.
4. The processor makes an uncompelled change to slave state at the issue cycle of the read request by asserting the **Release*** signal for one cycle.

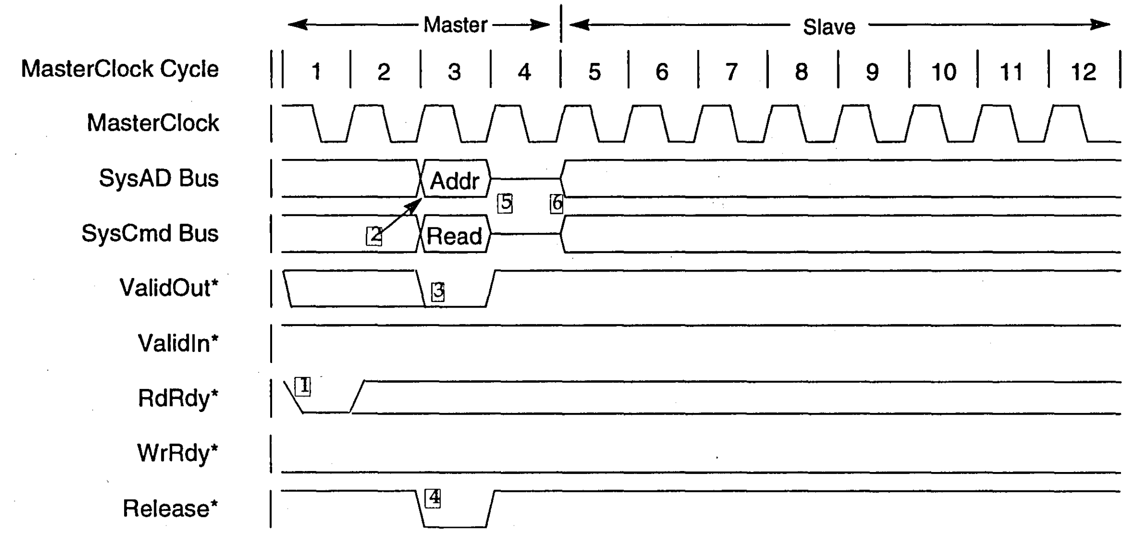
   Note: The external agent must not assert the signal **ExtRqst*** for the purposes of returning a read response, but rather must wait for the uncompelled change to slave state. The signal **ExtRqst*** can be asserted before or during a read response to perform an external request other than a read response.
5. The processor releases the **SysCmd** and the **SysAD** buses one **MasterClock** cycle after the assertion of **Release***.
6. The external agent drives the **SysCmd** and the **SysAD** buses within two cycles after the assertion of **Release***.

Once in slave state (starting at cycle 5 in Figure 13.2), the external agent can return the requested data through a read response. The read response can return the requested data or, if the requested data could not be successfully retrieved, an indication that the returned data is erroneous. If the returned data is erroneous, the processor takes a bus error exception.

Note: For read response data the R4650 only checks the error bits for the first doubleword in 64-bit bus mode, and the first word in 32-bit bus mode. All other error bits are ignored. **WrRdy*** is not checked during processor read requests.

Figure 13.2 illustrates a processor read request, coupled with an uncompelled change to slave state.

Note: Timings for the **SysADC** and **SysCmdP** buses are the same as those of the **SysAD** and **SysCmd** buses, respectively.



Note: Numbers in boxes correspond to numbered steps in preceding text.

Figure 13.2 Processor Read Request Protocol

The assertion of **Release*** indicates either an uncompelled change to slave state, or a response to the assertion of **ExtRqst***, whereupon the processor accepts either a read response, or any other external request. If any external request other than a read response is issued, the processor performs another uncompelled change to slave state after processing the external request by asserting release for one clock cycle.

The actual read response, where the external agent returns the requested data, is shown later in this chapter.

External Instruction Read Response Time

The R4650 accesses the external bus due to instruction cache miss or an uncached reference. The length of time for an external read is based on the overhead at the beginning and end of the read along with the time to drive the address and get the response data.

Instruction Read Latency Steps for System Clock

The read latency for a system clock in the multiply-by-two mode is as follows:

1. The startup overhead is one to two pipeline cycles (PCycle) for the CPU to transfer the address to the pads to be output. The second PCycle is needed if the miss is detected on a PCycle not aligned with the rising edge of **MasterClock**.

2. The CPU drives the address on the **SysAD** bus for two PCycles.
3. The CPU tri-states the **SysAD** bus for two PCycles.
4. The CPU waits for the main memory to return the data. This is expressed as *n* x 2 PCycles.
5. The first double word is driven in the **SysAD** from the main memory for two PCycles.
6. The remaining three double words of instruction are driven on **SysAD** for 3*2 PCycles.

Note that:

- For instruction misses, the pipeline starts after all the instructions are returned.
- *n* is the total number of idle cycles (even between double word instruction). For zero wait state systems, *n* = 0.

Example of Instruction Block Read With Zero Wait State

shows an instruction block read with a zero wait state (n=0):

Step	Description	PCycles
1	CPU overhead for cache miss detection	1-2
2	Address driven on **SysAD** bus	2
3	**SysAD** bus tri-stated	2
4	Memory latency to return the data (nx2)	0*2
5	First double word driven on SysAD bus	2
6	Remaining three instructions returned	2*3=6
	Total PCycles:	13-14

Table 13.4 Steps for Single Read With Zero Wait State

External Data Read Response Time

The R4650 accesses the external bus due to data cache miss or an uncached reference. The length of time for an external read is based on the overhead at the beginning and end of the read along with the time to drive the address and get the response data.

Data Read Latency Steps for System Clock

The read latency for a system clock that is in the multiply-by-two mode is as follows:

1. The startup overhead is one to two pipeline cycles (PCycle) for the CPU to generate the parity for the address to be output. The second PCycle is needed if the miss is detected or a PCycle not aligned with the rising edge of SClock.
2. The CPU drives the address on the **SysAD** bus for two PCycles.
3. The CPU tri-states the **SysAD** bus for two PCycles.
4. The CPU waits for the main memory to return the data. This is expressed as *n* x 2 PCycles where *n* is the number of MasterClock cycles for the first data to be returned in a block read, or the latency for the single read. For zero wait state memory system *n* should be zero.
5. The first double word is driven in the **SysAD** from the main memory for two PCycles.
6. The end of the overhead is two PCycles: one to transfer the data from the pads and generate the parity, and one to write to the register (or cache, if it is cacheable data).

Note the following:

- If *n*=0 and the line being replaced is dirty, the CPU takes one to two additional PCycles of overhead to move the dirty data into the write buffer.
- The additional latency for returning the remaining three data elements should be added in a manner similar to the instruction read latency.
- If cache line needs to be written back, the read request is posted first and then the write is completed.

Example of Data Single Read With Zero Wait State

Table 13.5 shows a data block read with a zero wait state (n=0):

Step	Description	PCycles
1	CPU overhead for cache miss detection	1-2
2	Address driven on **SysAD** bus	2
3	**SysAD** bus tri-stated	2
4	Memory latency to return the data (nx2)	0*2
5	First double word driven on **SysAD** bus	2
6	CPU overhead to write the data cache, do the fixup, and then restart	2
	Total PCycles:	9-10

Table 13.5 Steps for Data Block Read With Zero Wait State

External Cycles for Read Latency

The external cycles to get the response data will look similar to Figure 13.3. For a larger "multiply-by" it will take longer to get the response data.

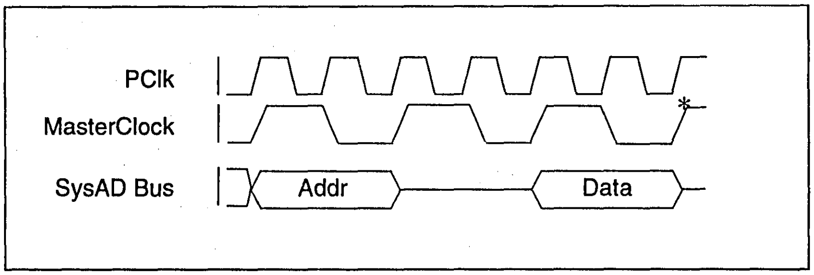


Figure 13.3 Uncached Read—External Cycles

The same operation is shown in greater detail in Figure 13.4. These figures assume the following:

- Data is returned immediately after **Release*** is asserted, and after the bus turnaround cycle (when the CPU tri-states the bus to allow the external agent to drive it).
- The data meets the setup and hold requirements for the rising edge of **MasterClock** that is identified in the·preceding and following figures with an asterisk.

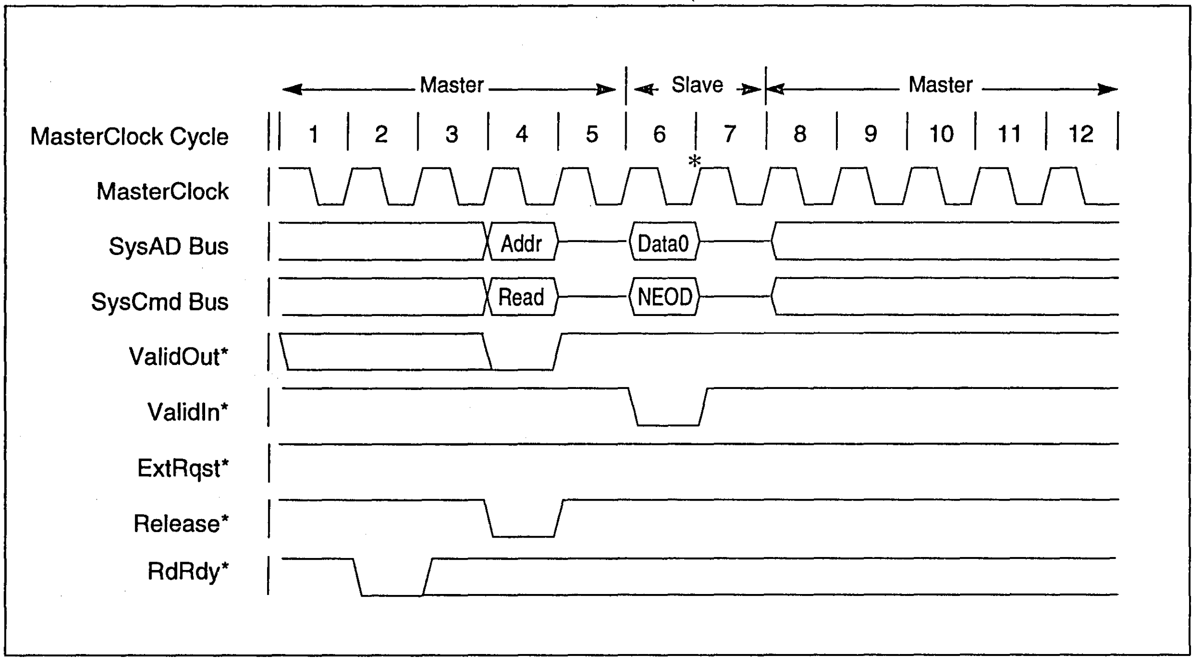


Figure 13.4 Processor Read Cycle

Read Response Protocol

An external agent must return data to the processor in response to a processor read request by using a read response protocol. A read response protocol consists of the following steps:

1. The external agent waits for the processor to perform an uncompelled change to slave state.
2. The external agent returns the data through a single data cycle or a series of data cycles.
3. After the last data cycle is issued, the read response is complete and the external agent sets the **SysCmd** and **SysAD** buses to a tri-state.
4. The system interface returns to master state.

   Note: The processor always performs an uncompelled change to slave state in the same cycle that it issues a read request.
5. The data identifier for data cycles must indicate the fact that this data is *response data.*
6. The data identifier associated with the last data cycle must contain a *last data cycle* indication.

For read responses to non-coherent block read requests (which is the only read request for normal operations of the R4650,) the response data will not need to identify an initial cache state. The cache state will automatically be assigned as dirty exclusive by the R4650.

The data identifier associated with a data cycle can indicate that the data transmitted during that cycle is erroneous; however, an external agent must return a data block of the correct size regardless of the fact that the data may be in error. The R4650 only checks the error bit for the first data of a block, while the other error bits for the block of data are ignored. If an initial erroneous data cycle is detected, the processor takes a bus error at the completion of the data transfer.

Read response data must only be delivered to the processor when a processor read request is pending. The behavior of the processor is undefined when a read response is presented to it and there is no processor read pending.

Figure 13.5 illustrates a processor word read request followed by a word read response. Figure 13.6 illustrates a read response for a processor block read with the system interface already in slave state. Figure 13.7 illustrates a block read transaction with one wait state.

Note: Timings for the **SysADC** and **SysCmdP** buses are the same as those of the **SysAD** and **SysCmd** buses, respectively.

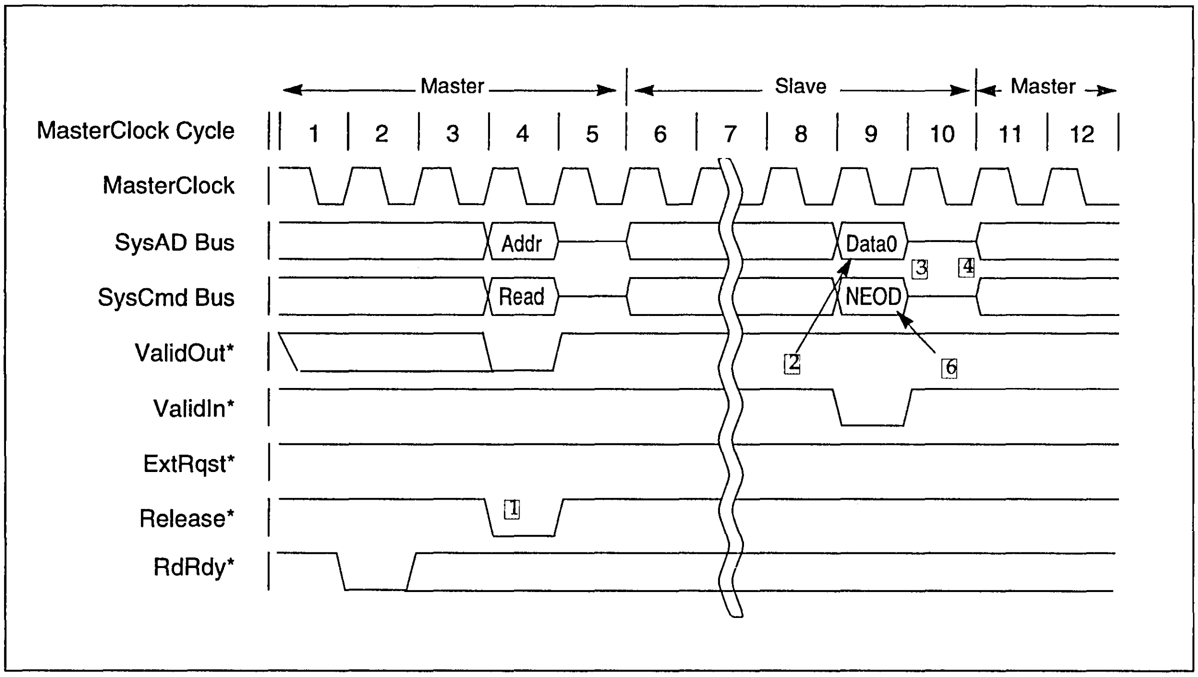


Figure 13.5 Processor Word Read Request Followed by a Word Read Response (64-bit bus interface)

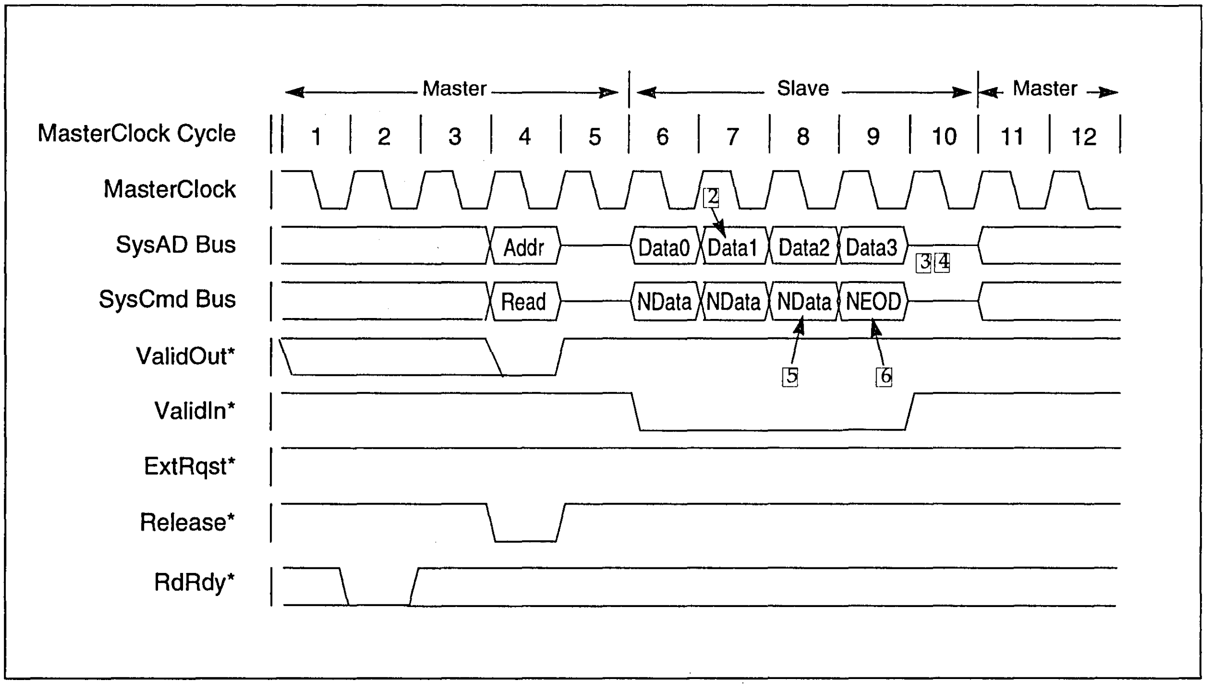


Figure 13.6 Block Read Response With Zero Wait State (64-bit bus interface)

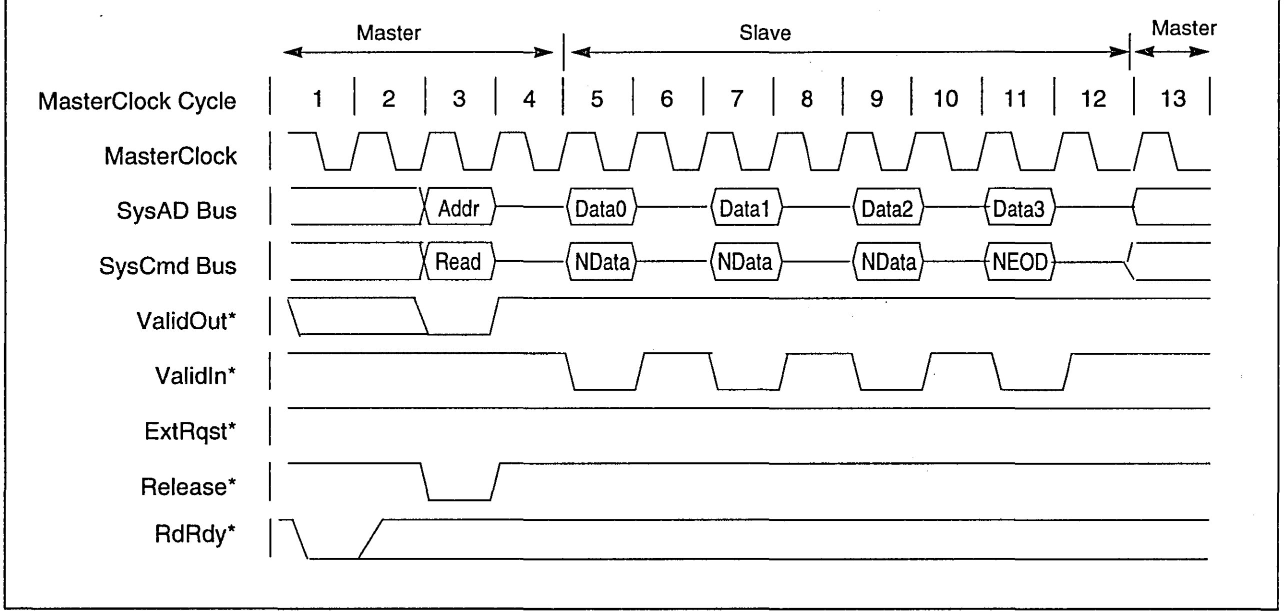


Figure 13.7 Block Read Transaction With One Wait State (64-bit bus interface)

Data Rate Control

The system interface supports a maximum data rate of one doubleword per cycle in 64-bit bus mode and one word per cycle in 32-bit bus mode. The data rate the processor can support is directly related to the rate at which the external agent can return data.

Read Data Pattern

The rate at which data is delivered to the processor can be determined by the external agent—for example, the external agent can drive data and assert **ValidIn*** every *n* cycles, instead of every cycle. An external agent can deliver data at any rate it chooses, but must not deliver data to the processor any faster than the processor is capable of receiving it.

The processor only accepts cycles as valid when **ValidIn*** is asserted and the **SysCmd** bus contains a data identifier. If the external agent sends more data items then requested (e.g., a fifth doubleword of read response data with **ValidIn*** asserted in 64-bit bus mode) or the last data (i.e., the fourth doubleword in 64-bit bus mode) of a block read is not tagged as the last data item, it is an error and the resulting actions of the processor for these cases will be undefined.

Figure 13.8 shows a read response with reduced data rate and with the system interface in slave state.

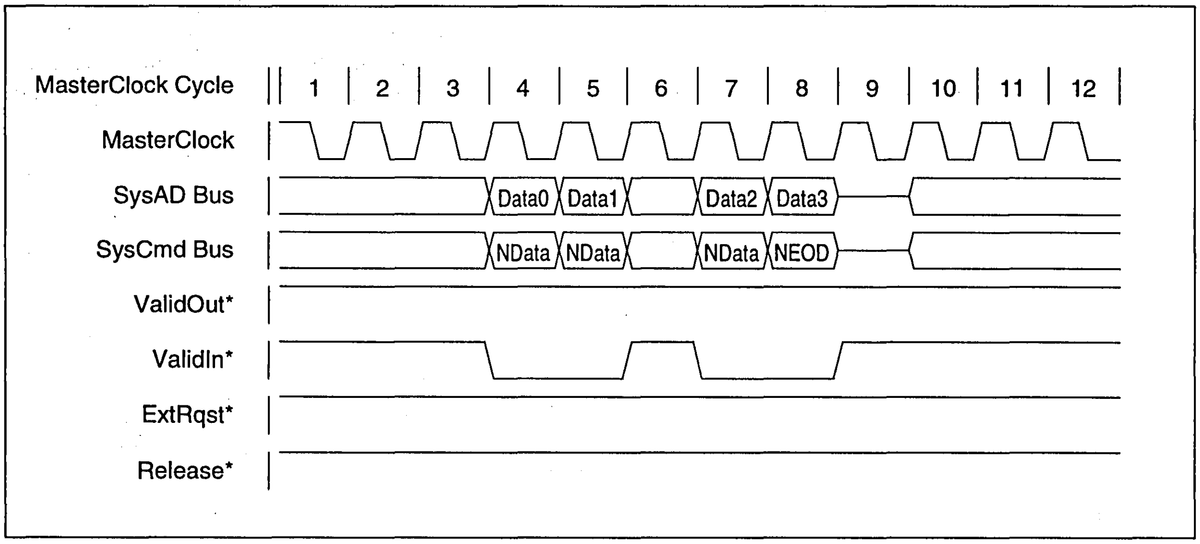


Figure 13.8 Read Response, Reduced Data Rate, System Interface in Slave State (64-bit bus interface)

64-Bit and 32-Bit Bus Modes

The bus interface of the R4650 can be configured during reset to be either 64-bit wide or 32-bit wide. The same bus protocol explained earlier in this chapter applies for both modes. In 32-bit bus mode, the internal execution core is still a full 64-bit engine. Only the bus interface unit can be configured as either 64-bit or 32-bit interface.

The bus width mode is a static feature of the device. This means that the bus width has to be configured once during reset. This feature should not be thought of as dynamic bus width interface where the bus width is 64-bit in one access and 32-bit wide in the other access.

64-Bit Bus Mode

In 64-bit bus mode, the R4650 supports 64-bit address/data system interface that consists of:

- 64-bit address and data, **SysAD(63:0)**
- 8-bit SysAD check bus, **SysADC(7:0)** (even parity)
- 9-bit command bus, **SysCmd(8:0)**
- Six handshake signals:
  RdRdy*, WrRdy*
  ExtReq*, Release*
  ValidIn*, ValidOut*

64-Bit Bus Mode Block Read Operation

In 64-bit bus mode, the R4650 issues a single block read request for the entire cache line (4 double words). The external agent should return all four double words as explained in the read protocol section earlier.

Figure 13.9 illustrates the timing diagram for a block read operation in 64-bit bus mode. The address issued by the R4650 is double word (64-bit) aligned.

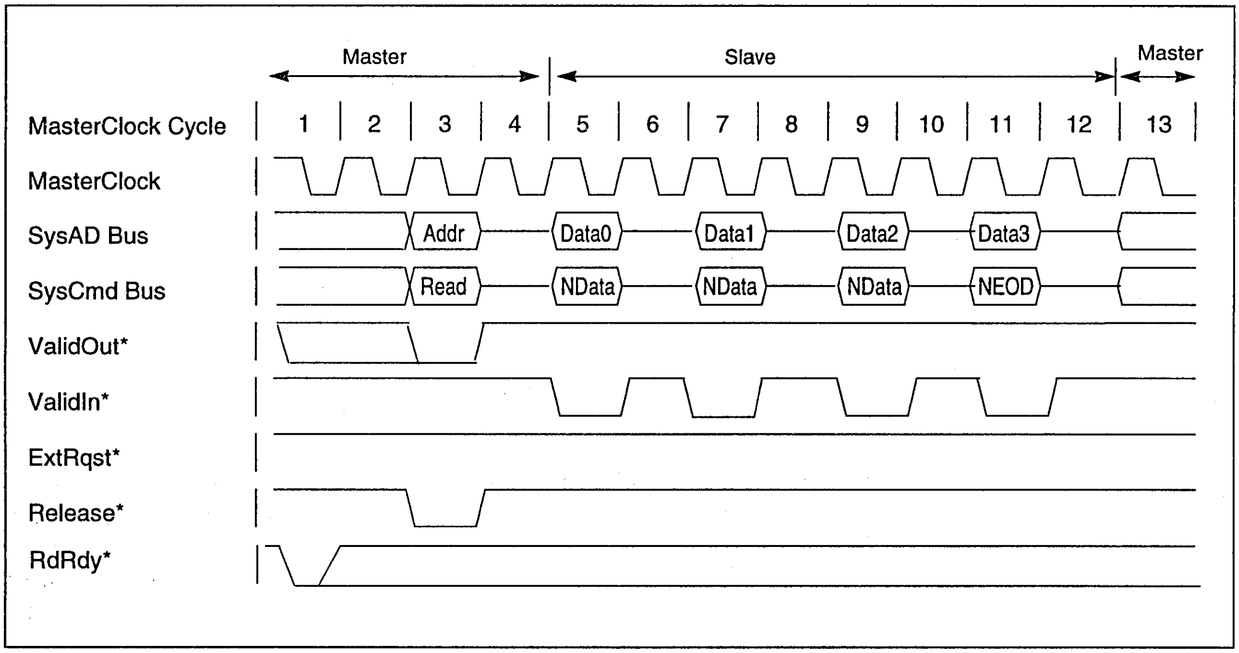


Figure 13.9 Block Read Transaction With One Wait State

64-Bit Bus Mode Single (Uncached) Read Operation

In 64-bit bus mode, the R4650 issues a single uncached read request using a doubleword (64-bit) aligned address. The actual access can be for a doubleword, word, partial word, or byte, but the request is called a *word read request* to differentiate it from the block read request.

Figure 13.10 illustrates the timing for an uncached read operation.

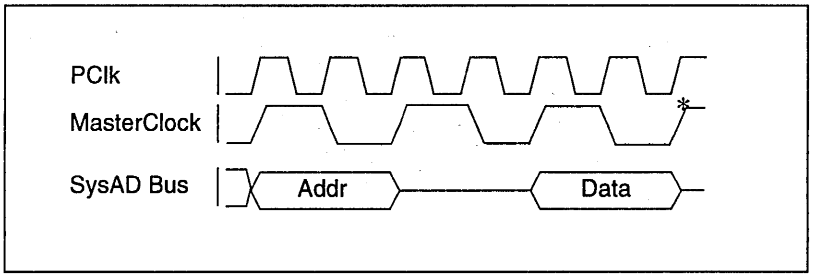


Figure 13.10 64-Bit Uncached Read—External Cycles

32-Bit Bus Mode

In 32-bit bus mode, the R4650 supports a 32-bit address/data system interface that consists of the following:

- The 32-bit address & data (**SysAD** (31:0)) and the 4-bit **SysAD** check bus (**SysADC** (3:0), even parity). **SysAD** (63:32) and **SysADC** (7:4) are undefined.
- 9-bit command bus, **SysCmd(8:0)**
- Six handshake signals:
  RdRdy*, **WrRdy***
  ExtReq*, **Release***
  ValidIn*, **ValidOut***

It is important to note that in the 32-bit bus mode **SysAd(31:0)** and **SysADC(3:0)** are always used regardless of the Endianness of the system.

It is also important to note that the encoding of **SysCmd(8:0)** is the same for both 64-bit and 32-bit bus modes. This means that the R4650 does not inform the external agent about the bus width mode. It is expected that this mode is programmed during reset and that the external agent is configured to interface to the R4650 in either 64-bit or 32-bit bus mode.

32-Bit Bus Mode Block Read Operation

In 32-bit bus mode, the R4650 issues a single block read request for the entire cache line (4 double words). since the bus interface is configured to be 32-bit wide, the R4650 issues a single address that is word (32-bit) aligned. The external agent should return 8 single words to the R4650 as explained in the read protocol section earlier.

Figure 13.11 illustrates the timing diagram for a block read operation in 32-bit bus mode. This means that a block read request is not divided into two requests. The external agent is responsible for returning all 8 single word to the R4650.

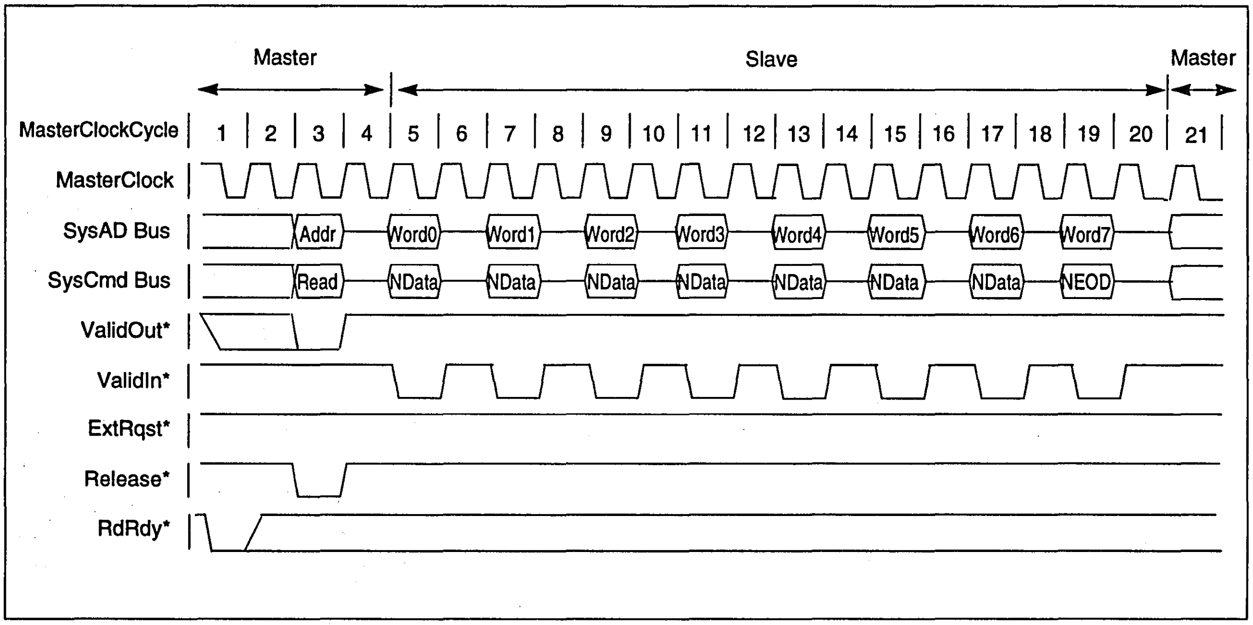


Figure 13.11 Block Read Transaction With One Wait State

The R4650 combines the word internally to generate a double word data to be used by the execution core. This implies that the order of the words in a double word data will be endian-dependent. On little-endian machines bits 31:0 will be transferred first and bits 63:32 transferred second; on a big- endian machine the order will be reversed.

32-Bit Bus Mode Single (Uncached) Read Operation

In 32-bit bus mode, the R4650 issues a single uncached read request using a word (32-bit) aligned address (the actual access could be for a word, partial word or a byte).

If the internal core requests an uncached data that is larger than a word, the external request is then broken into two external requests. The first request will transfer 4 bytes and the second will transfer up to 4 bytes.

Figure 13.12 illustrates the timing for an uncached read operation of one word.

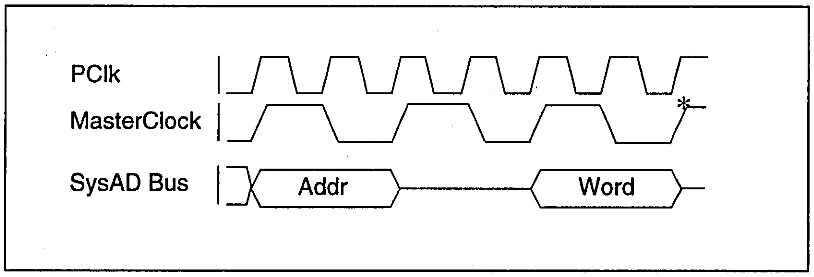


Figure 13.12 32-Bit Bus Mode Uncached Read for Single Word

Figure 13.13 illustrates the timing diagram for an uncached read operation of a double word value.

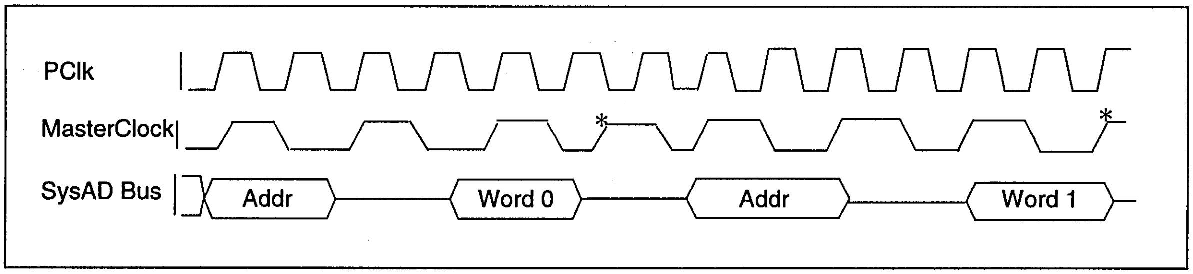


Figure 13.13 32-Bit Bus Mode Uncached Read for Double Word

The R4650 combines the word internally to generate a double word data to be used by the execution core. This implies that the order of the words in a double word data will be endian dependent. On little-endian machines, bits 31:0 will be transferred first, with bits 63:32 transferred second. On a big-endian machine, the order will be reversed.

Subblock Ordering

The order in which data is returned in response to a processor block read request is *subblock ordering*. In subblock ordering, the processor delivers the address of the requested doubleword (in 64-bit bus mode) or word (in 32-bit bus mode) within the block. An external agent must return the block of data using subblock ordering, starting with the addressed doubleword or word.

In general, a block of data elements (whether bytes, halfwords, words, or doublewords) can be retrieved from storage in two ways: in sequential order, or using a subblock order. This section describes these retrieval methods, with an emphasis on subblock ordering. Note that the R4650 uses only subblock ordering for block reads.

Example of Sequential Ordering

Sequential ordering retrieves the data elements of a block in serial, or sequential, order.

Figure 13.14 shows a sequential order in which doubleword 0 (DWO) is taken first and doubleword 3 (DW3) is taken last.

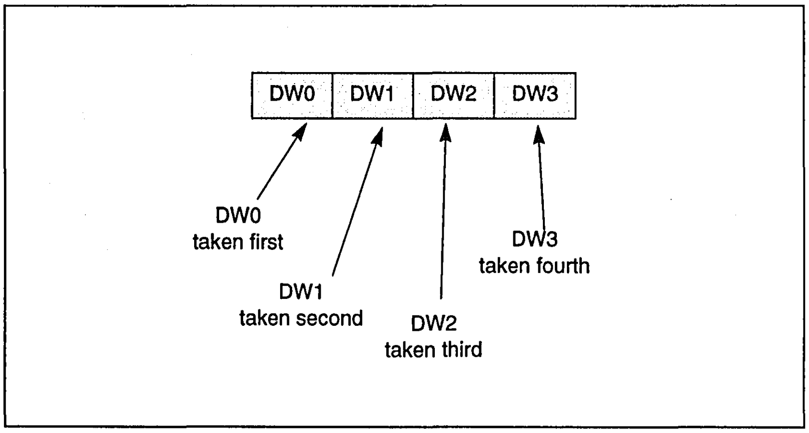


Figure 13.14 Retrieving a Data Block in Sequential Order

Examples of Subblock Ordering

Subblock ordering allows the system to define the order in which the data elements are retrieved. In 64-bit bus mode the smallest data element of a block transfer for the R4650 is a doubleword, and in 32-bit bus mode, a single word.

Figure 13.15 shows the retrieval of a block of data that consists of four doublewords in 64-bit bus mode, with doubleword 2 taken first. Cache line size is 8 words.

Using the subblock ordering shown in Figure 13.15, the doubleword at the target address is retrieved first (doubleword 2), followed by the remaining doubleword (doubleword 3) in this quadword. Next, the quadword that fills out the octalword are retrieved in the same order as the prior quadword (in this case doubleword 0 is followed by doubleword 1).

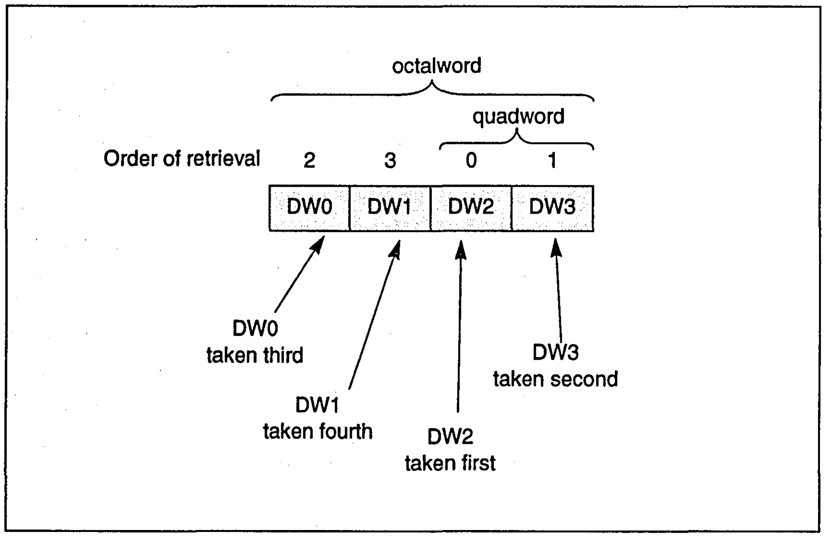


Figure 13.15 Retrieving Data in a Subblock Order

Figure 13.16 shows the retrieval of a block of data that consists of 8 words in 32-bit bus mode, with word 2 taken first. Cache line size is 8 words.

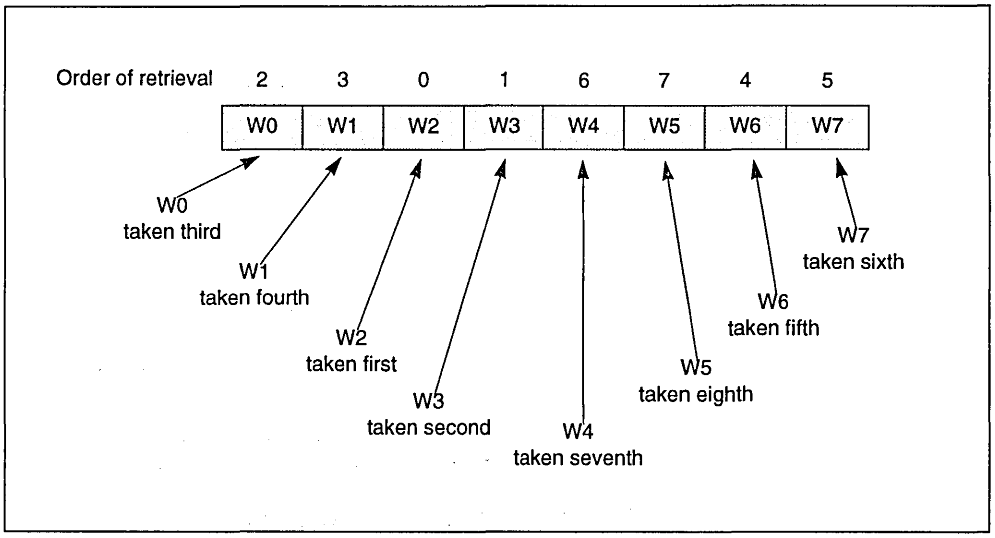


Figure 13.16 Retrieving Data in a Subblock Order

Using the subblock ordering shown in Figure 13.16, the word at the target address, in this case word 2, is retrieved first, followed by word 3. Next, word 6 is followed by word 7, then word 4, followed by word 5. Word 0 is then followed by word 1.

A simpler way to understand subblock ordering would be to take a look at the method used for generating the address of each doubleword or word as it is retrieved. The subblock ordering logic generates this address by executing a bit-wise exclusive-OR (XOR) of the starting block address with the output of a binary counter that increments with each doubleword or word, starting at doubleword 0 (00_2) or word 0 (000_2).

Generating Subblock Order of Doublewords

Using this scheme, Table 13.6, Table 13.7, and Table 13.8 list the subblock ordering of doublewords for an 8-word block, based on three different starting-block addresses: 10_2, 11_2, and 01_2. The subblock ordering is generated by an XOR of the subblock address (either 10_2, 11_2, or 01_2) with the binary count of the doubleword (00_2 through 11_2).

Thus, the third doubleword retrieved from a block of data with a starting address of 10_2 is determined by taking the XOR of address 10_2 with the binary count of doubleword 2, 10_2. The result is 00_2, or doubleword 0, as shown in Table 13.6).

Cycle	Starting Block Address	Binary Count	Double Word Retrieved
1	10	00	10
2	10	01	11
3	10	10	00
4	10	11	01

Table 13.6 Sequence of Doublewords Transferred Using Subblock Ordering: Address 10_2

Cycle	Starting Block Address	Binary Count	Double Word Retrieved
1	11	00	11
2	11	01	10
3	11	10	01
4	11	11	00

Table 13.7 Sequence of Doublewords Transferred Using Subblock Ordering: Address 11_2

Cycle	Starting Block Address	Binary Count	Double Word Retrieved
1	01	00	01
2	01	01	00
3	01	10	11
4	01	11	10

Table 13.8 Sequence of Doublewords Transferred Using Subblock Ordering: Address 01_2

Generating Subblock Order of Words

Using the same scheme, Table 13.9 and Table 13.10 list the subblock ordering of words for an 8-word block, based on two different starting-block addresses: 010_2 and 011_2. The subblock ordering is generated by an XOR of the subblock address (either 010_2 or 011_2) with the binary count of the word (000_2 through 111_2).

Therefore, the third word retrieved from a block of data with a starting address of 010_2 is determined by taking the XOR of address 010_2 with the binary count of word 2, 010_2. The result is 000_2, or word 0, as shown in Table 13.9.

Cycle	Starting Block Address	Binary Count	Word Retrieved
1	010	000	010
2	010	001	011
3	010	010	000
4	010	011	001
5	010	100	110
6	010	101	111
7	010	110	100
8	010	111	101

Table 13.9 Sequence of Words Transferred Using Subblock Ordering: Address 010_2

Cycle	Starting Block Address	Binary Count	Word Retrieved
1	011	000	011
2	011	001	010
3	011	010	001
4	011	011	000
5	011	100	111
6	011	101	110
7	011	110	101
8	011	111	100

Table 13.10 Sequence of Words Transferred Using Subblock Ordering: Address 011_2

System Interface Commands and Data Identifiers

System interface commands specify the nature and attributes of any system interface request; this specification is made during the address cycle for the request. System interface data identifiers specify the attributes of data transmitted during a system interface data cycle.

The following sections describe the syntax, that is, the bitwise encoding, of system interface commands and data identifiers. The same **SysCmd** encoding is used for both 32-bit and 64-bit bus mode. The selection of 64-bit versus 32-bit is not dynamic and should be done only once during Reset. The R4650 does not indicate externally whether the bus is configured as 32-bit or 64-bit.

Reserved bits and reserved fields in the command or data identifier should be set to 1 for system interface commands and data identifiers associated with external requests. For system interface commands and data identifiers associated with processor requests, reserved bits and reserved fields in the command and data identifier are undefined.

Command and Data Identifier Syntax

System interface commands and data identifiers are encoded in 9 bits and are transmitted on the **SysCmd** bus from the processor to an external agent, or from an external agent to the processor, during address and data cycles. Bit 8 (the most-significant bit) of the **SysCmd** bus determines whether the current content of the **SysCmd** bus is a command or a data identifier and, therefore, whether the current cycle is an address cycle or a data cycle. For system interface commands, **SysCmd(8)** must be set to 0. For system interface data identifiers, **SysCmd(8)** must be set to 1.

System Interface Command Syntax

This section describes the **SysCmd** bus encoding for system interface commands. Figure 13.17 shows a common encoding used for all system interface commands.

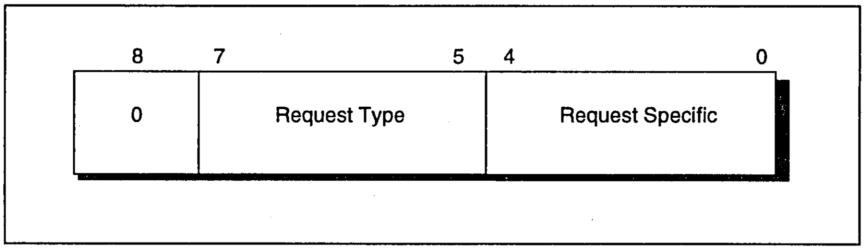


Figure 13.17 System Interface Command Syntax Bit Definition

SysCmd(8) must be set to 0 for all system interface commands.

SysCmd(7:5) specify the system interface request type which may be read, write or null; Table 13.11 illustrates the types of requests encoded by the **SysCmd(7:5)** bits.

SysCmd(7:5)	Command
0	Read Request
1	Reserved
2	Write Request
3	Null Request
4 - 7	Reserved

Table 13.11 Encoding of SysCmd(7:5) for System Interface Commands

SysCmd(4:0) are specific to each type of request and are defined in each of the following sections.

Read Requests

Figure 13.18 shows the format of a **SysCmd** read request.

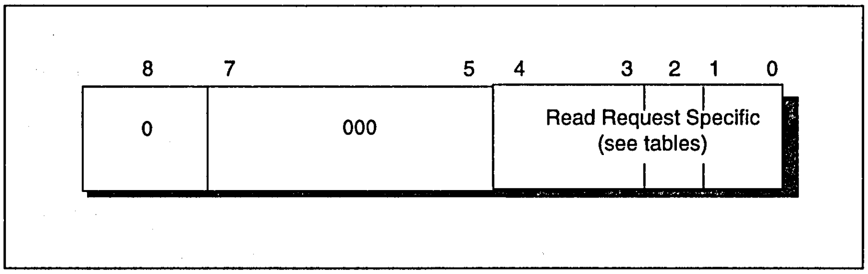


Figure 13.18 Read Request SysCmd Bus Bit Definition

Table 13.12, Table 13.13, and Table 13.14 list the encoding of **SysCmd(4:0)** for read requests.

SysCmd(4:3)	Read Attributes
0 - 1	Reserved
2	Noncoherent block read
3	64-bit mode: Doubleword, partial doubleword, word, or partial word 32-bit bus mode: Word or partial word.

Table 13.12 Encoding of SysCmd(4:3) for Read Requests

SysCmd(2)	Link Address Retained Indication
0	Link address not retained
1	Link address retained
SysCmd(1:0)	**Read Block Size**
0	Reserved
1	8 words (64-bit or 32-bit bus modes)
2 - 3	Reserved

Table 13.13 Encoding of SysCmd(2:0) for Block Read Request

SysCmd(2:0)	Read Data Size
	64-bit or 32-bit bus mode:
0 1 2 3	1 byte valid (Byte) 2 bytes valid (Halfword) 3 bytes valid (Tribyte) 4 bytes valid (Word)
	64-bit mode only:
4 5 6 7	5 bytes valid (Quintibyte) 6 bytes valid (Sextibyte) 7 bytes valid (Septibyte) 8 bytes valid (Doubleword)

Table 13.14 Doubleword, Word, or Partial-Word Read Request Data Size Encoding of SysCmd(2:0)

System Interface Data Identifier Syntax

This section defines the encoding of the **SysCmd** bus for system interface data identifiers. Figure 13.19 shows a common encoding scheme used for all system interface data identifiers.

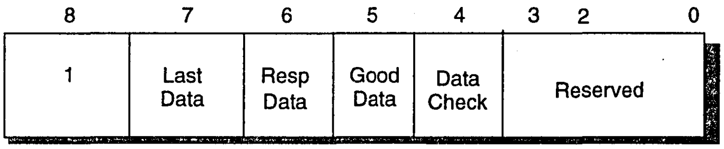


Figure 13.19 Data Identifier SysCmd Bus Bit Definition

SysCmd(8) must be set to 1 for all system interface data identifiers. system interface data identifiers use the format for noncoherent data.

Noncoherent Data

Noncoherent data is defined as follows:

- data that is associated with processor block write requests and processor doubleword, partial doubleword, word, or partial word write requests
- data that is returned in response to a processor noncoherent block read request or a processor doubleword, partial doubleword, word, or partial word read request
- data that is associated with external write requests
- data that is returned in response to an external read request

Data Identifier Bit Definitions

SysCmd(7) marks the last data element and **SysCmd(6)** indicates whether or not the data is response data, for both processor and external coherent and noncoherent data identifiers. Response data is data returned in response to a read request.

SysCmd(5) indicates whether or not the data element is error free. Erroneous data contains an uncorrectable error and is returned to the processor, forcing a bus error. The processor delivers data with the good data bit deasserted if a primary parity error is detected for a transmitted data item.

SysCmd(4) indicates to the processor whether to check the data and check bits for this data element.

SysCmd(3) is reserved for external data identifiers.

SysCmd(4:3) are reserved for noncoherent processor data identifiers.

SysCmd(2:0) are reserved for noncoherent data identifiers.

Table 13.15 lists the 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 13.15 Processor Data Identifier Encoding of SysCmd(7:3)

Table 13.16 lists the 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	Do not check the data and check bits
SysCmd(3)	Reserved

Table 13.16 External Data Identifier Encoding of SysCmd(7:3)

During data cycles in 64-bit bus mode, the valid byte lanes depend upon the position of the data with respect to the aligned doubleword (this may be a byte, halfword, tribyte, quadbyte/word, quintibyte, sextibyte, septibyte, or an octalbyte/doubleword). For example, in little-endian mode, on a byte request where the address modulo 8 is 0, **SysAD(7:0)** are valid during the data cycles.

Table 13.17 shows the byte lanes used for partial word transfers for both little and big endian in 64-bit bus mode.

# Bytes	Address	SysAD Byte Lanes Used (Big Endian)							
SysCmd(2:0)	Mod 8	63:56	55:48	47:40	39:32	31:24	23:16	15:8	7:0
1 (000)	0	•							
	1		•						
	2			•					
	3				•				
	4					•			
	5						•		
	6							•	
	7								•
2 (001)	0	•	•						
	2			•	•				
	4					•	•		
	6							•	•
3 (010)	0	•	•	•					
	1		•	•	•				
	4					•	•	•	
	5						•	•	•
4 (011)	0	•	•	•	•				
	4					•	•	•	•
5 (100)	0	•	•	•	•	•			
	3				•	•	•	•	•
6 (101)	0	•	•	•	•	•	•		
	2			•	•	•	•	•	•
7 (110)	0	•	•	•	•	•	•	•	
	1		•	•	•	•	•	•	•
8 (111)	0	•	•	•	•	•	•	•	•
		7:0	15:8	23:16	31:24	39:32	47:40	55:48	63:56
		SysAD Byte Lanes Used (Little Endian)							

Table 13.17 Partial Word Transfer Byte Lane Usage—64-Bit Mode

During data cycles in 32-bit bus mode, the valid byte lanes depend upon the position of the data with respect to the aligned word, which may be a byte, halfword, tribyte, or word. For example, in little-endian mode, on a byte request where the address modulo 4 is 0, **SysAD(7:0)** are valid during the data cycles.

Table 13.18 shows the byte lanes used for partial word transfers for both little and big endian in 32-bit bus mode.

# Bytes	Address	SysAD Byte Lanes Used (Big Endian)			
SysCmd(2:0)	Mod 4	31:24	23:16	15:8	7:0
1 (000)	0	•			
	1		•		
	2			•	
	3				•
2 (001)	0	•	•		
	2			•	•
3 (010)	0	•	•	•	
	1		•	•	•
4 (011)	0	•	•	•	•
		0:7	8:15	16:23	24:31
		SysAD Byte Lanes Used (Little Endian)			

Table 13.18 Partial Word Transfer Byte Lane Usage—32-Bit Mode

Integrated Device Technology, Inc.

The Write Interface — Chapter 14

Introduction

This chapter discusses the Write protocol and associated operations. When a processor issues a write request, the specified resource is accessed and the data is written to it. A processor write request is complete after the last word of data has been transmitted to the external agent. In no-secondary-cache mode, the external agent must be capable of accepting a processor write request any time **WrRdy*** has been asserted for one clock cycle, two cycles before the issue cycle.

The R4650 has added two new modes to enhance the throughput of non-block writes. These modes allow for 2 cycle throughput on back-to-back non-block writes. The external agent must be capable of accepting a processor write request in these modes under the same conditions as for the R4x00 compatibility mode (except as noted later in this chapter).

Processor Write Protocols

The following sections contain a cycle-by-cycle description of the bus arbitration protocols for the processor write request. Table 14.1 describes the buses that appear in the timing diagrams that follow.

Scope	Abbreviation	Description
Global	Unsd	Unused
SysAD bus	Addr	Physical address
	Data<n>	Data element number n of a block of data
SysCmd bus	Cmd	An unspecified system interface command
	Read	A processor or external read request command
	Write	A processor or external write request command
	SINull	A system interface release external null request command
	NData	A noncoherent data identifier for a data element other than the last data element
	NEOD	A noncoherent data identifier for the last data element

Table 14.1 System Interface Requests

The R4650 has three write protocols:

- R4xxx compatible
- Pipeline write
- Write reissue

These protocols apply to both single and block write and to 32-bit and 64-bit interface mode. This means, for example, that for pipeline write a single write can be followed immediately by a block write that the external agent must accept.

The write protocol is selected through the reset vector, along with the bus width interface. The selection of the write protocol is static, which means that it should be selected once during reset.

In R4xxx-compatible write a single write access takes four clock cycles, while in pipeline write or write reissue a single write access takes two clock cycles.

Processor Write Request Protocol

Processor write requests are issued using one of two protocols:

- Doubleword, partial doubleword, word, or partial word writes use a word[1] write request protocol.
- Block writes use a block write request protocol.

Processor word write requests are issued with the system interface in master state, as described in the following steps. These steps apply to both 64-bit and 32-bit bus interface modes.

1. 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.
2. The processor asserts **ValidOut***.
3. The processor drives a data identifier on the **SysCmd** bus and data on the **SysAD** bus.
4. The data identifier associated with the data cycle must contain a last data cycle indication. At the end of the cycle, **ValidOut*** is deasserted.

Timings for the **SysADC** and **SysCmdP** buses are the same as those of the **SysAD** and **SysCmd** buses, respectively. Figure 14.1 shows a processor noncoherent word write request cycle.

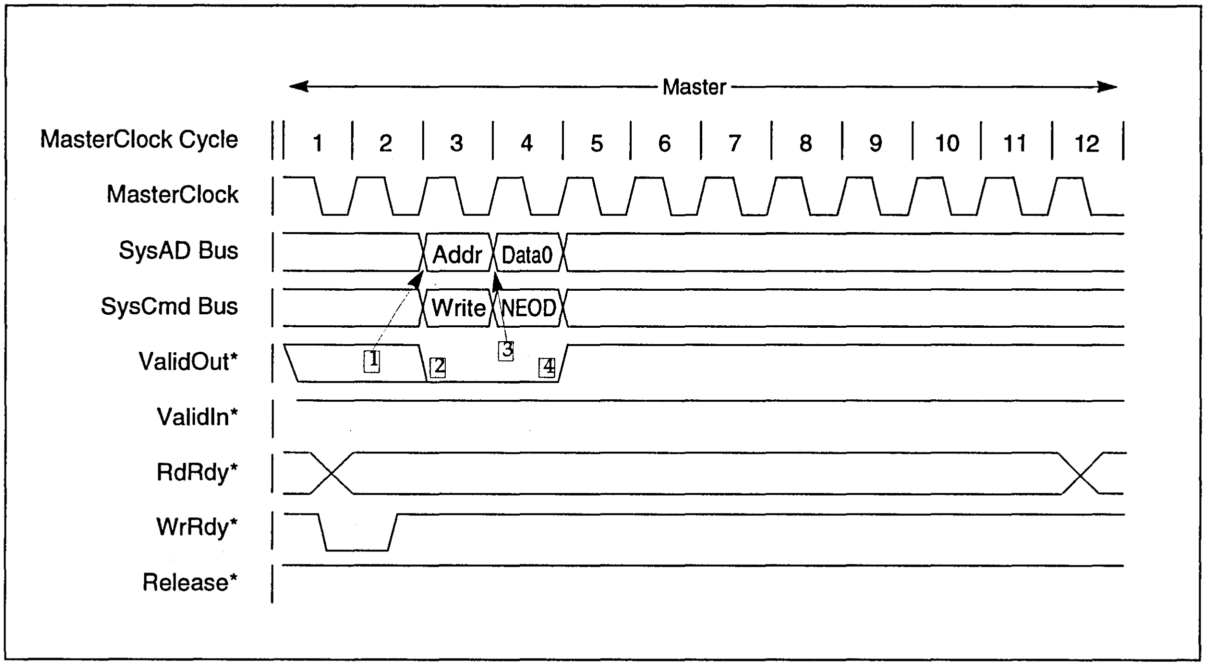


Figure 14.1 Processor Noncoherent Word Write Request Protocol

Processor Single Write Request

There are three types of processor single write requests, as follows:

- R4000-compatible writes
- Write reissue
- Pipelined writes

In this section, each one is discussed in detail.

[1] Called *word* to distinguish it from *block* request protocol. Data transferred can actually be doubleword, partial doubleword, word, or partial word.

R4000-Compatible Write Mode

In R4000-compatible write mode a single write operation takes four clock cycles. The address is asserted for one clock cycle, followed by one clock cycle of data and then two unused clock cycles. This applies to both 64-bit and 32-bit bus modes, and is illustrated in Figure 14.2

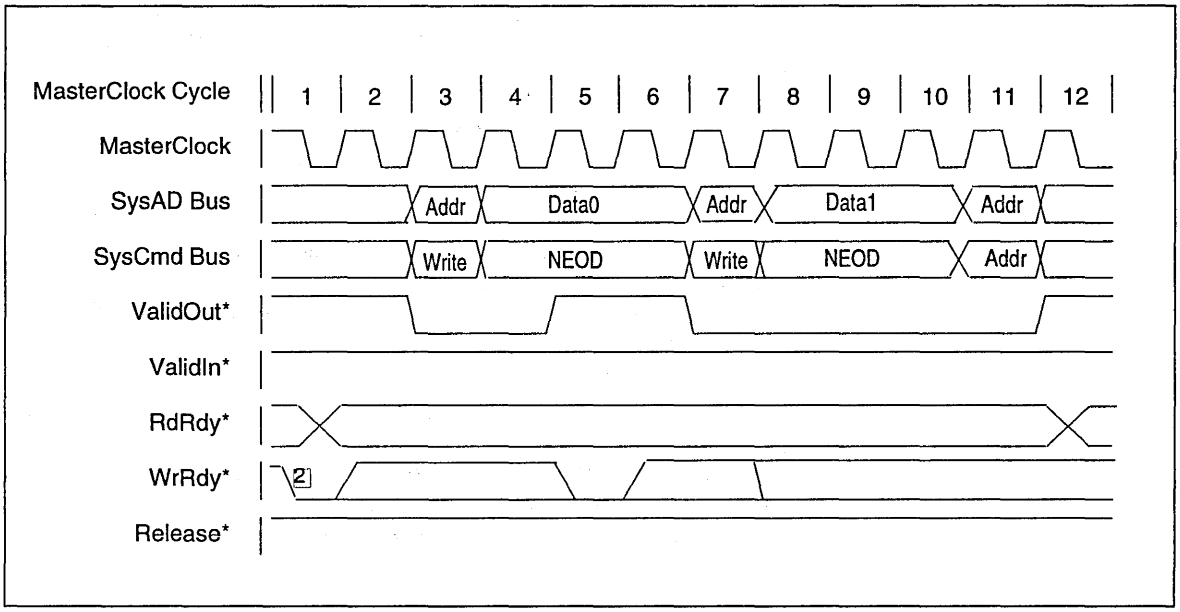


Figure 14.2 R4000 Compatible Write Mode

The R4650 interface requires that **WrRdy*** be asserted two system cycles prior to the issue of a write, for one clock cycle. An external agent that deasserts **WrRdy*** immediately upon receiving the write that fills its buffer will stop a subsequent write for four system cycles in R4000 non-block write compatible mode. This leaves two null system cycles after a write address/data pair to give the external agent time to stop the next write.

An Address/data pair every four system cycles is not sufficiently high performance for all applications. For this reason, the R4650 provides two new protocol options that modify the R4000 back-to-back write protocol to allow an address/data pair every two system cycles. The first protocol, called write reissue, allows **WrRdy*** to be deasserted during the address cycle and forces a write to be reissued. The second, called pipelined writes, leaves the sample point of **WrRdy*** unchanged and requires that the external agent accept one more write than the R4000 protocol.

Write Reissue

In Write Reissue mode, writes issue when **WrRdy*** is asserted both for 1 clock cycle, two cycles prior to the address cycle and during the address cycle. The write reissue protocol is shown in Figure 14.3. For this figure, note the following:

- For Addr0/Data0 the write will issue because **WrRdy*** is sampled LOW at *0 and at *1, which is the issue cycle.
- Addr1/Data1 will not issue because **WrRdy*** is sampled HIGH at *2, which is the possible issue cycle.
- This address/data pair will then be reissued to the system interface, and will issue as indicated in Figure 12.3 because **WrRdy*** is sampled LOW at *3 and at *4.

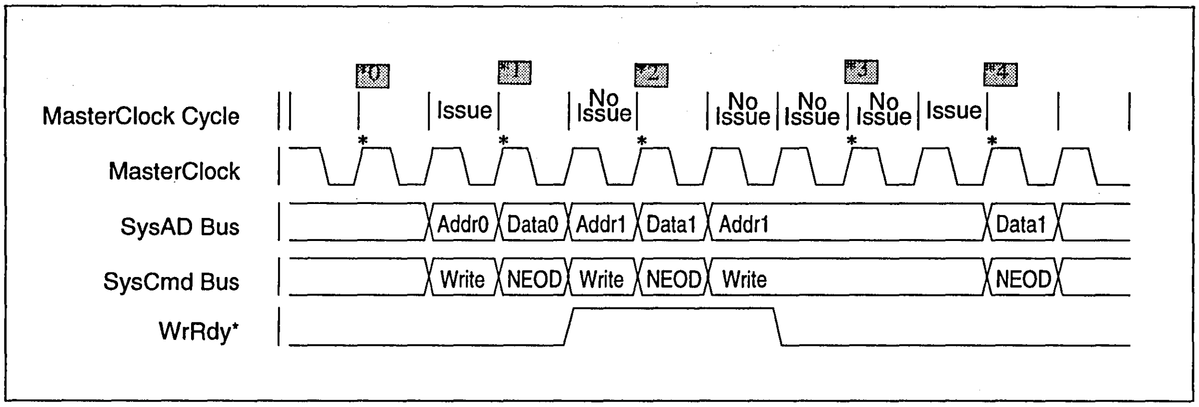


Figure 14.3 Write Reissue

Pipelined Write

The pipelined write protocol maintains the R4000 write issue rule (which is, issue if **WrRdy*** is asserted two cycles prior to the address cycle, for one clock cycle), and eliminates the two null cycles between writes. The external agent may be required to accept one more write after it deasserts **WrRdy***.

This protocol is shown in Figure 14.4. For this figure note the following:

- Addr0/Data0 issues because **WrRdy*** was asserted at *0.
- Addr1/Data1 will be issued because **WrRdy*** was asserted at *1.
- Addr2/Data2 will not issue at first because **WrRdy*** is sampled HIGH at *2. It will issue as indicated in the figure because **WrRdy*** was sampled LOW at *3.

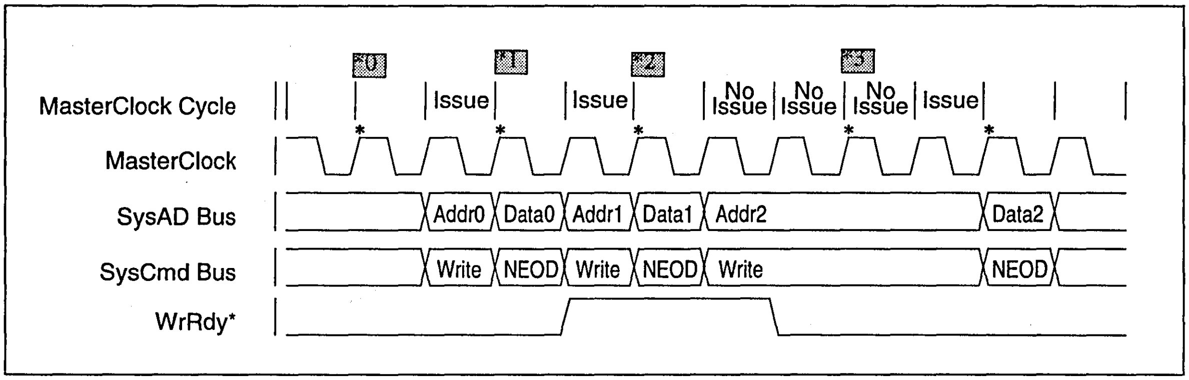


Figure 14.4 Pipelined Writes

All three write protocols apply for both single write and block writes. This means that in pipeline write, for example, a single write can be followed immediately by a block write that the external agent must accept.

Processor Block Write Request

Processor block write requests are issued with the system interface in master state, as described below. The protocol is the same for either 64-bit or 32-bit bus mode. A processor noncoherent block request for eight words of data in 64-bit bus mode is illustrated in Figure 14.5.

1. The processor issues a write command on the **SysCmd** bus and a write address on the **SysAD** bus
2. The processor asserts **ValidOut***.
3. The processor drives a data identifier on the **SysCmd** bus and data on the **SysAD** bus.
4. The processor asserts **ValidOut*** for a number of cycles sufficient to transmit the block of data.
5. The data identifier associated with the last data cycle must contain a last data cycle indication.

Figure 14.5 illustrates a processor noncoherent block request for eight words of data with a data pattern of DDDD in 64-bit bus mode.

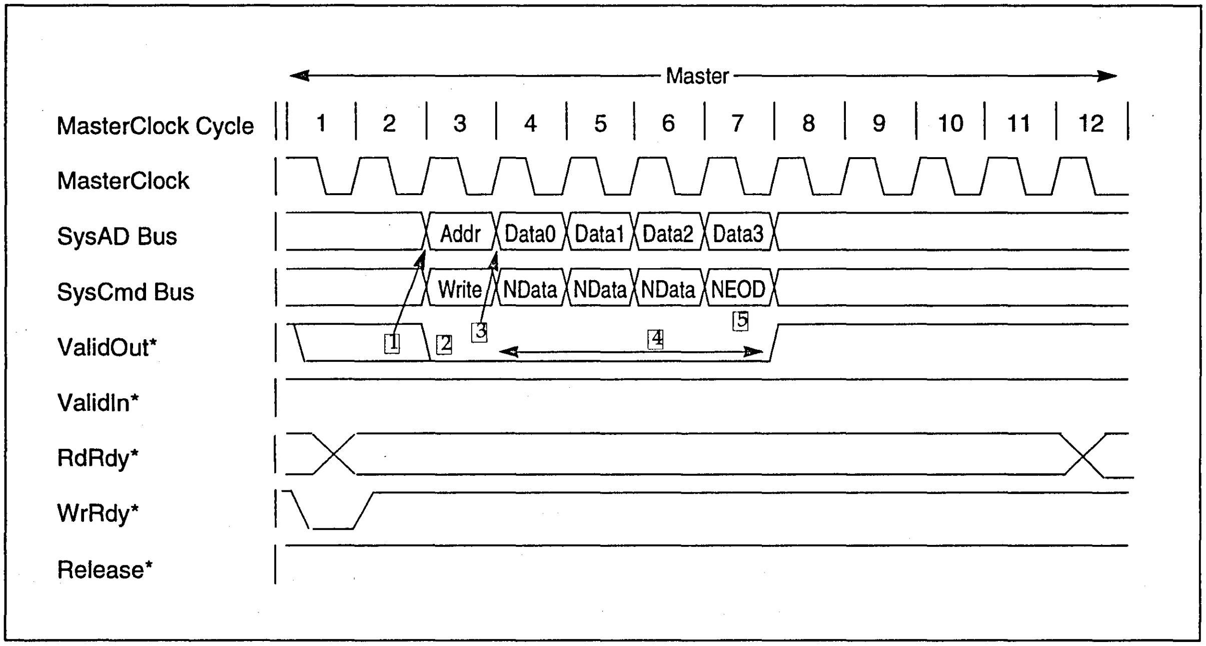


Figure 14.5 Processor Noncoherent Block Write Request Protocol

Write Data Transfer Patterns

The write data pattern specifies the pattern the R4650 uses when writing a block to the external agent. This pattern is specified once through the mode bits during reset.

A data pattern is a sequence of letters indicating the *data* and *unused* cycles that repeat to provide the appropriate data rate. For example, the data pattern **DDxx** specifies a repeatable data rate of two doublewords every four cycles, with the last two cycles unused.

Table 14.2 lists the maximum processor data rate and the data pattern for each data rate in 64-bit mode. Data patterns are specified using the characters *D* and *x; D* indicates a doubleword data cycle and *x* indicates an unused cycle. During the unused cycles, the data bus will maintain the last doubleword data value (D).

Maximum Data Transmit Rate Block Writes	Data Pattern
1 Double/1 MasterClock Cycle	DDDD
2 Doubles/3 MasterClock Cycles	DDxDDx
1 Double/2 MasterClock Cycles	DDxxDDxx
1 Double/2 MasterClock Cycles	DxDxDxDx
2 Doubles/5 MasterClock Cycles	DDxxxDDxxx
1 Double/3 MasterClock Cycles	DDxxxxDDxxxx
1 Double/3 MasterClock Cycles	DxxDxxDxxDxx
1 Double/4 MasterClock Cycles	DDxxxxxxDDxxxxxx
1 Double/4 MasterClock Cycles	DxxxDxxxDxxxDxxx

Table 14.2 Transmit Data Rates and Patterns in 64-Bit Mode

Table 14.3 lists the maximum processor data rate and the data pattern for each data rate in 32-bit mode. Data patterns are specified using the characters *W* and *x*; W indicates a word data cycle and *x* indicates an unused cycle. During the unused cycles, the data bus will maintain the last word data value (D).

Maximum Data Transmit Rate Block Writes	Data Pattern
1 Double/1 MasterClock Cycle	WWWWWWWW
2 Doubles/3 MasterClock Cycles	WWxWWxWWxWWx
1 Double/2 MasterClock Cycles	WWxxWWxxWWxxWWxx
1 Double/2 MasterClock Cycles	WxWxWxWxWxWxWxWx
2 Doubles/5 MasterClock Cycles	WWxxxWWxxxWWxxxWWxxx
1 Double/3 MasterClock Cycles	WWxxxxWWxxxxWWxxxxWWxxxx
1 Double/3 MasterClock Cycles	WxxWxxWxxWxxWxxWxxWxxWxx
1 Double/4 MasterClock Cycles	WWxxxxxxWWxxxxxxWWxxxxxxWWxxxxxx
1 Double/4 MasterClock Cycles	WxxxWxxxWxxxWxxxWxxxWxxxWxxxWxxx

Table 14.3 Transmit Data Rates and Patterns in 32-Bit Mode

Processor Request and Flow Control

To control the flow of processor write requests, the external agent uses **WrRdy***. These are the steps that occur:

1. The processor samples the signal **WrRdy*** to determine if the external agent is capable of accepting a read request.
2. The processor does not complete the issue of a read request, until it issues an address cycle in response to the request for which the signal **RdRdy*** was asserted two cycles earlier.
3. The processor does not complete the issue of a write request until it issues an address cycle in response to the write request for which the signal **WrRdy*** was asserted two cycles earlier.

Figure 14.6 illustrates two processor write requests in which the issue of the second is delayed for the assertion of **WrRdy***. These steps apply for both 64-bit and 32-bit bus modes.

Note: Timings for the **SysADC** and **SysCmdP** buses are the same as for the **SysAD** and **SysCmd** buses, respectively.

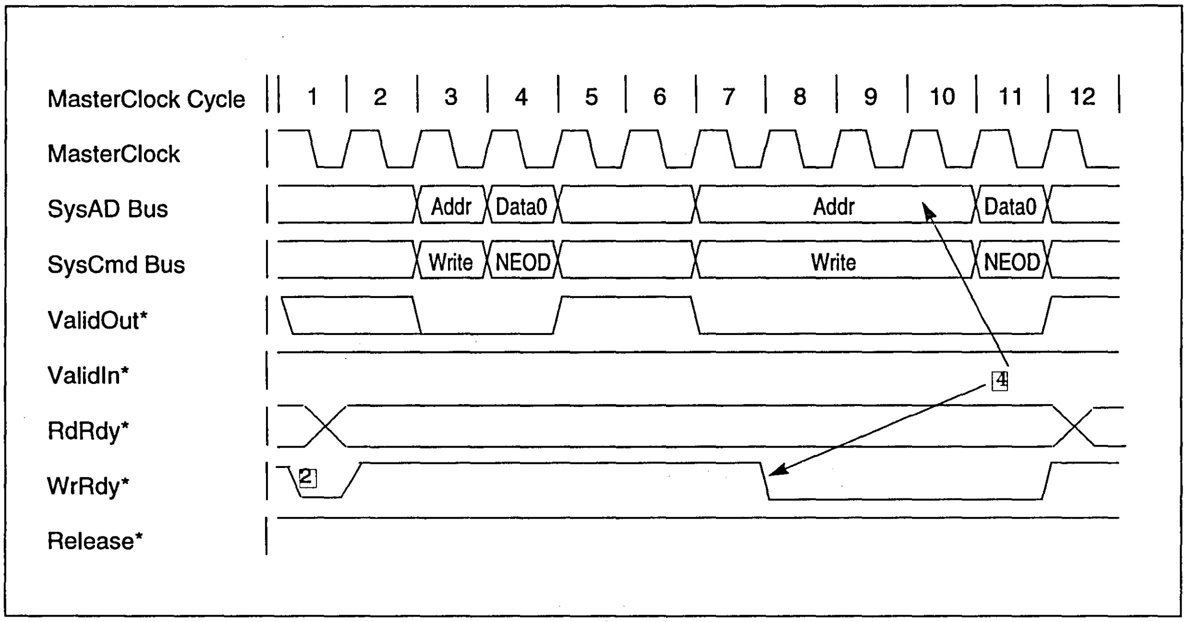


Figure 14.6 Two Processor Write Requests, Second Write Delayed for the Assertion of WrRdy*

64-Bit and 32-Bit Bus Modes

The bus interface of the R4650 can be configured during reset to be either 64-bit wide or 32-bit wide. The same bus protocol explained earlier in this chapter applies for both modes. In 32-bit bus mode, the internal execution core is still a full 64-bit engine. Only the bus interface unit can be configured as either 64-bit or 32-bit interface.

The bus width mode is a static feature of the device. This means that the bus width has to be configured once during reset. This feature should not be thought of as dynamic bus width interface where the bus width is 64-bit in one access and 32-bit wide in the other access.

64-Bit Bus Mode

In 64-bit bus mode, the R4650 supports 64-bit address/data system interface that consist of:

- 64-bit address and data, **SysAD(63:0)**
- 8-bit SysAD check bus, **SysADC(7:0)** (even parity)
- 9-bit command bus, **SysCmd(8:0)**
- Six handshake signals:
  RdRdy*, WrRdy*
  ExtReq*, Release*
  ValidIn*, ValidOut*

64-Bit Bus Mode Block Write Operation

In 64-bit bus mode, the R4650 issues a single block write request for the entire cache line (4 double words). The external agent should return all four double words as explained in the write protocol section earlier. Figure 14.7 illustrates the timing diagram for a block write operation in 64-bit bus mode. The address issued by the R4650 is double word (64-bit) aligned.

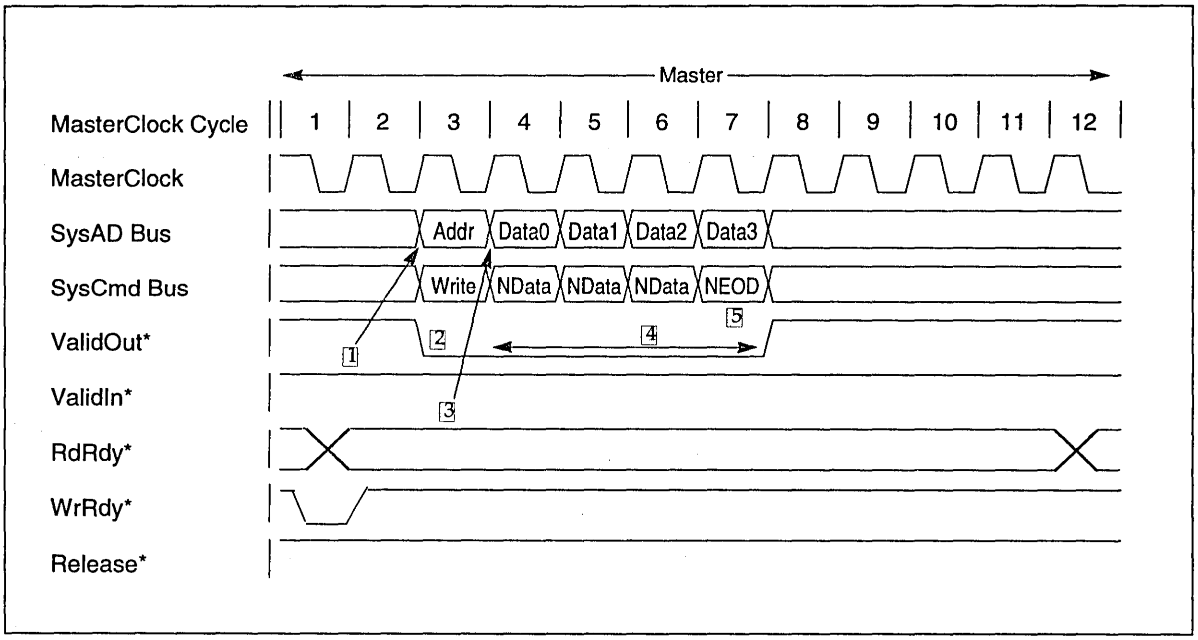


Figure 14.7 Processor Noncoherent Block Write Request Protocol

64-Bit Bus Mode Single (Uncached) Write Operation

In 64-bit bus mode, the R4650 issues a single uncached write request using a doubleword (64-bit) aligned address. The actual access can be for a doubleword, word, partial word, or byte, but the request is called a *word write request* to differentiate it from the block write request.

R4000-Compatible Write Mode

In R4000-compatible write mode, a single write operation takes four clock cycles. The address is asserted for one clock cycle, followed by one clock cycle of data and then two unused clock cycles. This is illustrated in Figure 14.8.

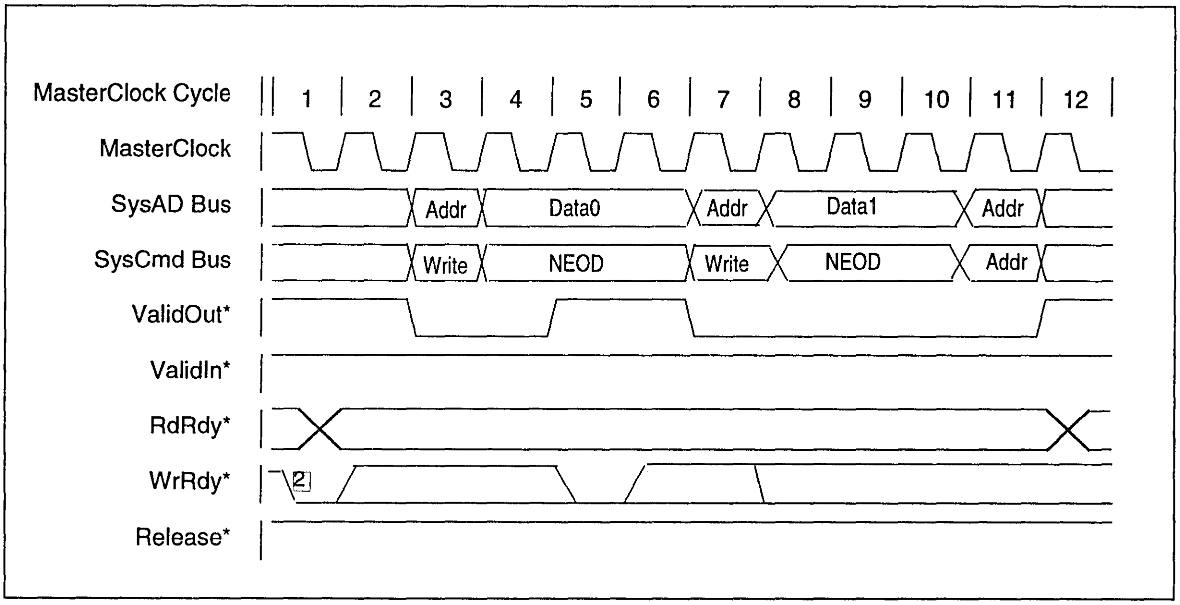


Figure 14.8 R4000 Compatible Write Mode

The R4650 interface requires that **WrRdy*** be asserted two system cycles prior to the issue of a write, for one clock cycle. An external agent that deasserts **WrRdy*** immediately upon receiving the write that fills its buffer will stop a subsequent write for four system cycles in R4000 non-block write compatible mode. This leaves two null system cycles after a write address/data pair to give the external agent time to stop the next write.

Write Reissue

Writes issue when **WrRdy*** is asserted both for 1 clock cycle, two cycles prior to the address cycle and during the address cycle. The write reissue protocol is shown in Figure 14.9. For this figure note the following:

- For Addr0/Data0 the write will issue because **WrRdy*** is sampled LOW at *0 and at *1, which is the issue cycle.
- Addr1/Data1 will not issue because **WrRdy*** is sampled HIGH at *2, which is the possible issue cycle.
- This address/data pair will then be reissued to the system interface, and will issue as indicated in Figure 14.9 because **WrRdy*** is sampled LOW at *3 and at *4.

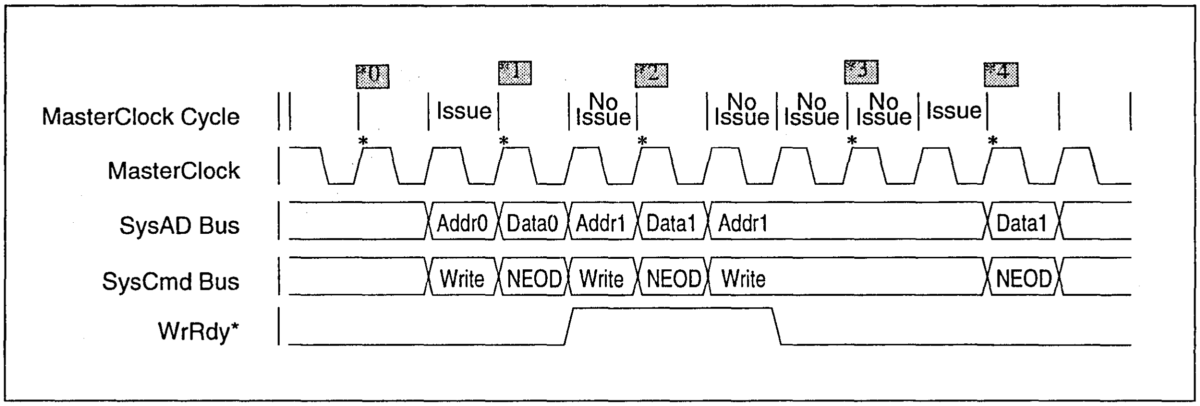


Figure 14.9 Write Reissue

Pipelined Write

The pipelined write protocol maintains the R4000 write issue rule (which is, issue if **WrRdy*** is asserted two cycles prior to the address cycle, for one clock cycle), and eliminates the two null cycles between writes. The external agent may be required to accept one more write after it deasserts **WrRdy***.

This protocol is shown in Figure 14.10. For this figure note the following:

- Addr0/Data0 issues because **WrRdy*** was asserted at *0.
- Addr1/Data1 will be issued because **WrRdy*** was asserted at *1.
- Addr2/Data2 will not issue at first because **WrRdy*** is sampled HIGH at *2. It will issue as indicated in the figure because **WrRdy*** was sampled LOW at *3.

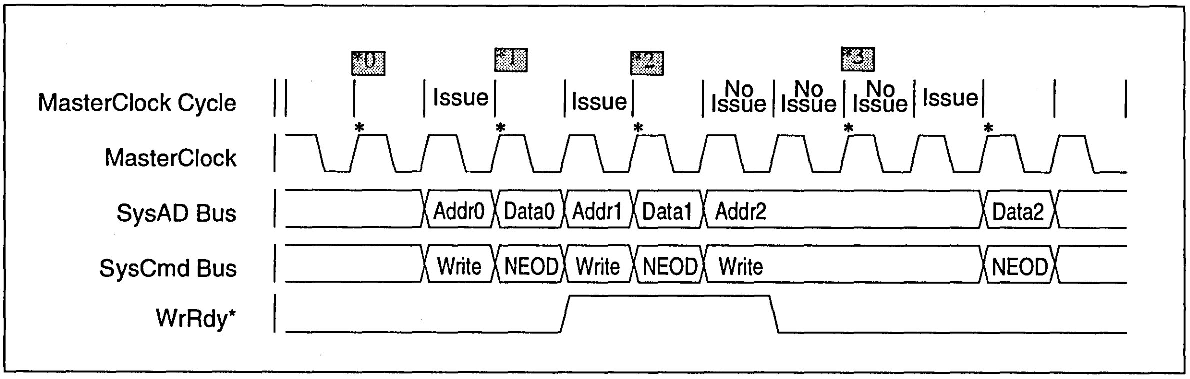


Figure 14.10 Pipelined Writes

All three write protocols apply for both single write and block writes. For example,this means that in pipeline write a single write can be followed immediately by a block write that the external agent must accept.

32-Bit Bus Mode

In 32-bit bus mode, the R4650 supports a 32-bit address/data system interface that consists of the following:

- The 32-bit address & data (**SysAD (31:0)**) and the 4-bit **SysAD** check bus (**SysADC(3:0)**, even parity). **SysAD(63:31)** and **SysADC(7:4)** are undefined.
- 9-bit command bus, **SysCmd(8:0)**
- Six handshake signals:
  RdRdy*, **WrRdy***
  ExtReq*, **Release***
  ValidIn*, **ValidOut***

It is important to note that in the 32-bit bus mode **SysAd(31:0)** and **SysADC(3:0)** are always used regardless of the Endianness of the system.

It is also important to note that the encoding of **SysCmd(8:0)** is the same for both 64-bit and 32-bit bus modes. This means that the R4650 does not inform the external agent about the bus width mode. It is expected that this mode is programmed during reset and that the external agent is configured to interface to the R4650 in either 64-bit or 32-bit bus mode.

32-Bit Bus Mode Block Write Operation

In 32-bit bus mode, the R4650 issues a single block write request for the entire cache line (4 double words). since the bus interface is configured to be 32-bit wide, the R4650 issues a single address that is word (32-bit) aligned, followed by 8 single words to the R4650.

Figure 14.11 illustrates the timing diagram for a block write operation in 32-bit bus mode. This means that a block write request is not divided into two requests. The external agent is responsible for accepting all 8 single word from the R4650.

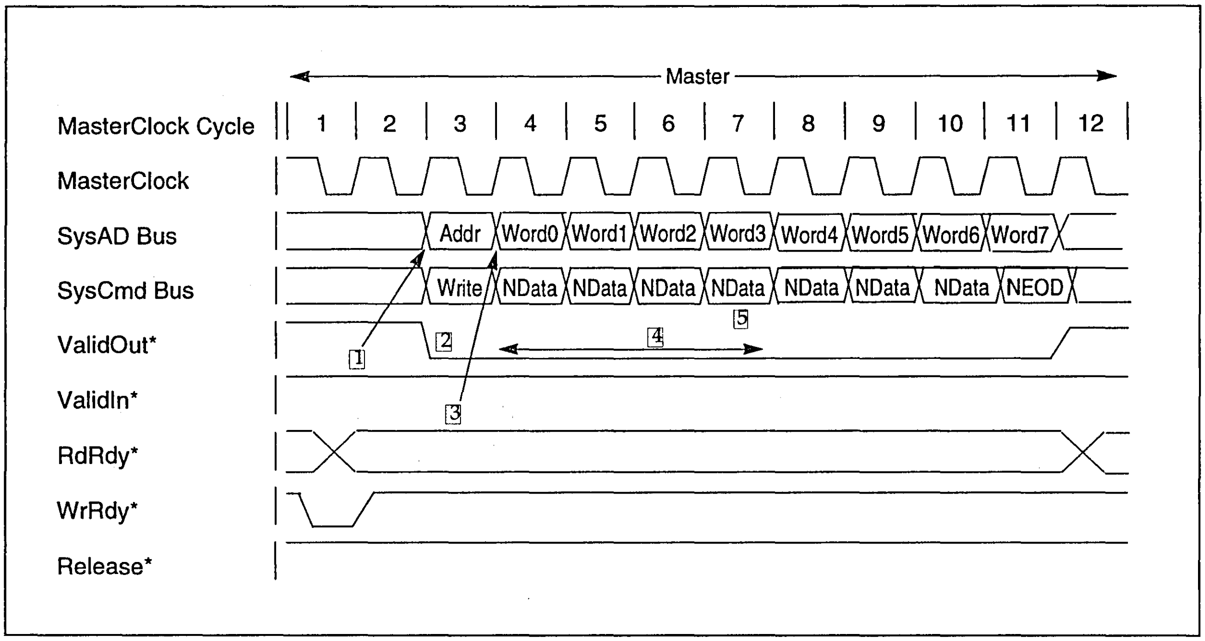


Figure 14.11 Processor Noncoherent Block Write Request Protocol

The order of the words in a double word datum will be endian-dependent. On little-endian machines bits 31:0 will be transferred first and bits 63:32 transferred second, while on a big-endian machine the order will be reversed.

32-Bit Bus Mode Single (Uncached) Write Operation

In 32-bit bus mode, the R4650 issues a single uncached write request using a word (32-bit) aligned address (the actual access could be for a word, partial word or a byte).

If the internal core writes an uncached datum that is larger than a word, the external request is then broken into two external requests. The first request will transfer 4 bytes and the second will transfer up to 4 bytes.

The order of the words in a double word datum will be endian dependent. On little-endian machines, bits 31:0 will be transferred first, with bits 63:32 transferred second. On a big-endian machine, the order will be reversed.

R4000-Compatible Write Mode

In R4000-compatible write mode, a single write operation takes four clock cycles. The address is asserted for one clock cycle, followed by one clock cycle of data and then two unused clock cycles. This is illustrated in Figure 14.12.

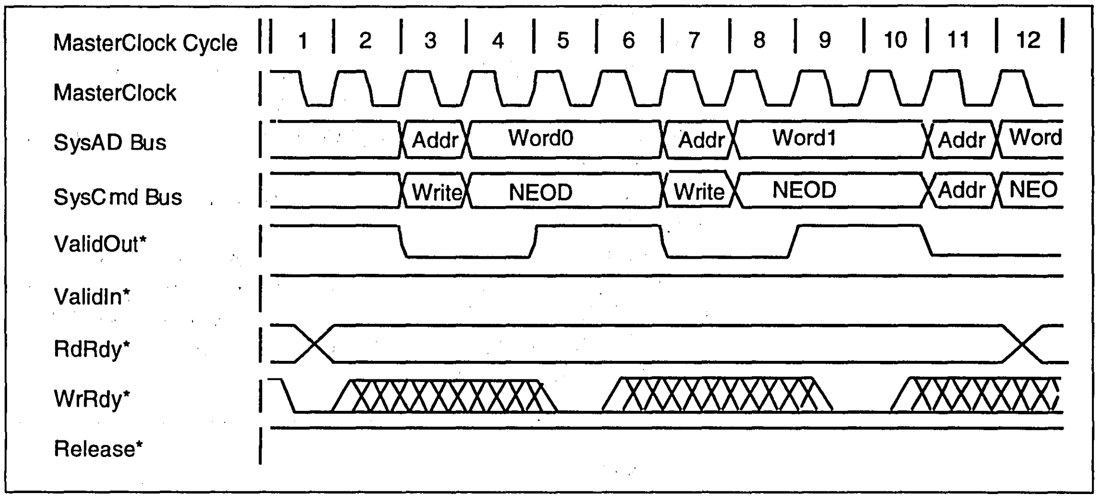


Figure 14.12 R4000 Compatible Write Protocol

The R4650 interface requires that **WrRdy*** be asserted two system cycles prior to the issue of a write, for one clock cycle. An external agent that deasserts **WrRdy*** immediately upon receiving the write that fills its buffer will stop a subsequent write for four system cycles in R4000 non-block write compatible mode. This leaves two null system cycles after a write address/data pair to give the external agent time to stop the next write.

Write Reissue

Writes issue when **WrRdy*** is asserted both for 1 clock cycle, two cycles prior to the address cycle and during the address cycle. A 64-bit transfer is broken into 2 word transfers. The write reissue protocol is shown in Figure 14.13. For this figure, note the following:

- For Addr0/Word0 the write will issue because **WrRdy*** is sampled LOW at *0 and at *1, which is the issue cycle.
- Addr1/Word1 will not issue because **WrRdy*** is sampled HIGH at *2, which is the possible issue cycle.
- This address/word pair will then be reissued to the system interface, and will issue as indicated in Figure 14.13 because **WrRdy*** is sampled LOW at *3 and at *4.

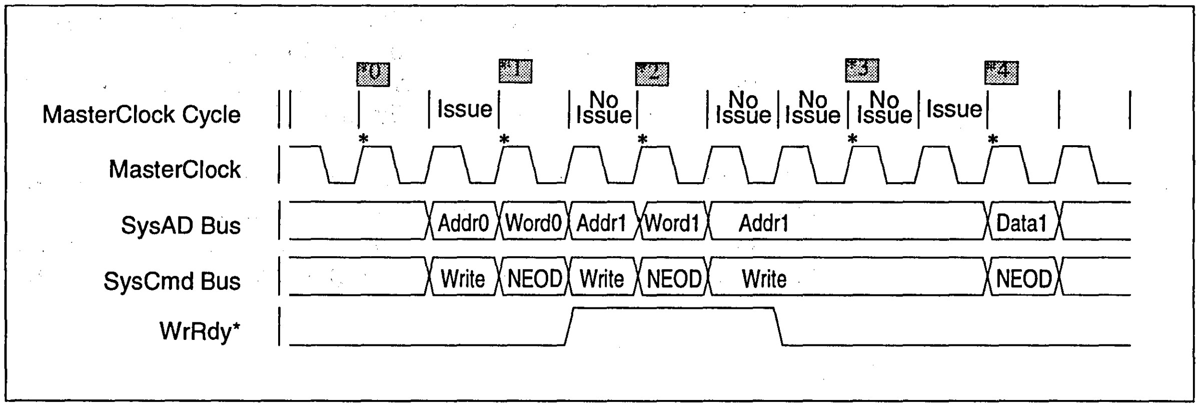


Figure 14.13 Write Reissue

Pipelined Write

The pipelined write protocol maintains the R4000 write issue rule (which is, issue if **WrRdy*** is asserted two cycles prior to the address cycle, for one clock cycle), and eliminates the two null cycles between writes. The external agent may be required to accept one more write after it deasserts **WrRdy***.

The pipeline write protocol is shown in Figure 14.14. For this figure, note the following:

- Addr0/Word0 issues because **WrRdy*** was asserted at *0.
- Addr1/Word1 will be issued because **WrRdy*** was asserted at *1.
- Addr2/Word2 will not issue at first because **WrRdy*** is sampled HIGH at *2. It will issue as indicated in the figure because **WrRdy*** was sampled LOW at *3.

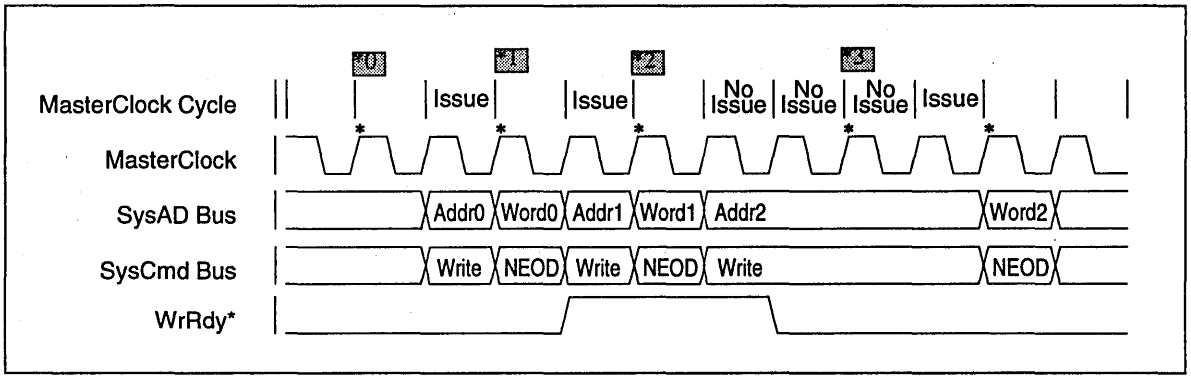


Figure 14.14 Pipelined Writes

All three write protocols apply for both single write and block writes. This means that in pipeline write, for example, a single write can be followed immediately by a block write that the external agent must accept.

Note: In 32-bit bus mode and pipeline write mode a single write can be followed by a block write of eight words. This means that the external agent must be capable of accepting all nine words both: a) in a sequential fashion, and b) at the speed of the data transmission pattern selected during reset.

Sequential Ordering

For block write requests in 64-bit bus mode, the processor always delivers the address of the doubleword at the beginning of the block. The processor delivers data beginning with the doubleword at the beginning of the block and progresses sequentially through the doublewords that form the block.

For block write requests in 32-bit bus mode, the processor always delivers the address of the word at the beginning of the block. The processor delivers data beginning with the word at the beginning of the block and progresses sequentially through the words that form the block.

Example of Sequential Ordering

Sequential ordering transfers the data elements of a block in serial, or sequential, order.

Figure 14.15 shows a sequential order in which doubleword 0 (DWO) is transferred first and doubleword 3 (DW3) is transferred last.

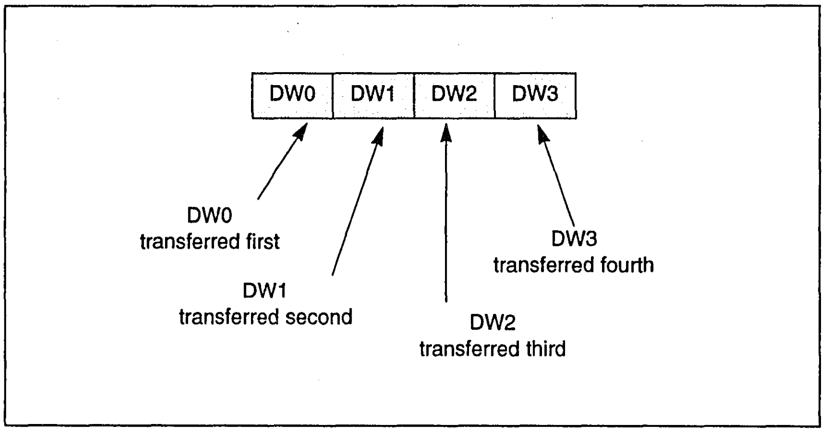


Figure 14.15 Transferring a Data Block in Sequential Order

Figure 14.16 shows a sequential order in which WordO (WO) is transferred first and Word 7 (W7) is transferred last.

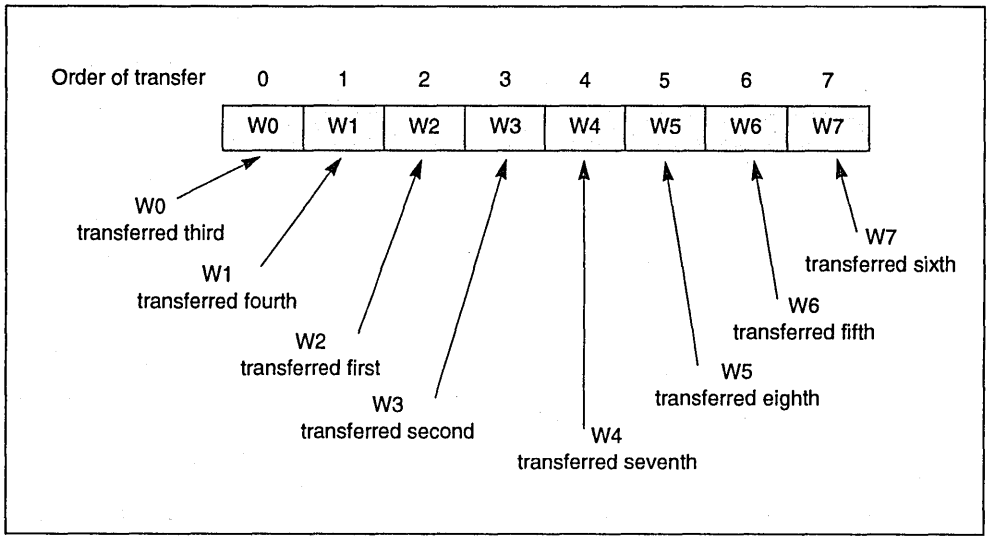


Figure 14.16 Transferring Data in a Subblock Order

Table 14.4 shows the byte lanes used for 64-bit bus mode partial word transfers for both little and big endian.

# Bytes	Address	SysAD byte lanes used (big endian)							
SysCmd(2:0)	Mod 8	63:56	55:48	47:40	39:32	31:24	23:16	15:8	7:0
1 (000)	0	•							
	1		•						
	2			•					
	3				•				
	4					•			
	5						•		
	6							•	
	7								•
2 (001)	0	•	•						
	2			•	•				
	4					•	•		
	6							•	•
3 (010)	0	•	•	•					
	1		•	•	•				
	4					•	•	•	
	5						•	•	•
4 (011)	0	•	•	•	•				
	4					•	•	•	•
5 (100)	0	•	•	•	•	•			
	3				•	•	•	•	•
6 (101)	0	•	•	•	•	•	•		
	2			•	•	•	•	•	•
7 (110)	0	•	•	•	•	•	•	•	
	1		•	•	•	•	•	•	•
8 (111)	0	•	•	•	•	•	•	•	•
		7:0	15:8	23:16	31:24	39:32	47:40	55:48	63:56
		SysAD byte lanes used (little endian)							

Table 14.4 Partial Word Transfer Byte Lane Usage

Table 14.5 shows the byte lanes used for 32-bit bus mode partial word transfers for both little and big endian.

# Bytes	Address	SysAD Byte Lanes Used (Big Endian)			
SysCmd(2:0)	Mod 4	31:24	23:16	15:8	7:0
1 (000)	0	•			
	1		•		
	2			•	
	3				•
2 (001)	0	•	•		
	2			•	•
3 (010)	0	•	•	•	
	1		•	•	•
4 (011)	0	•	•	•	•
		0:7	8:15	16:23	24:31
		SysAD Byte Lanes Used (Little Endian)			

Table 14.5 Partial Word Transfer Byte Lane Usage—32-Bit Mode

During data cycles, the valid byte lines depend upon the position of the data with respect to the aligned doubleword (this may be a byte, halfword, tribyte, quadbyte/word, quintibyte, sextibyte, septibyte, or an octalbyte/ doubleword). For example, in little-endian mode, on a byte request where the address modulo 8 is 0, **SysAD(7:0)** are valid during the data cycles.

System Interface Commands and Data Identifiers

System interface commands specify the nature and attributes of any system interface request; this specification is made during the address cycle for the request. System interface data identifiers specify the attributes of data transmitted during a system interface data cycle.

The following sections describe the syntax, that is, the bitwise encoding, of system interface commands and data identifiers. The same **SysCmd** encoding is used for both 32-bit and 64-bit bus mode. The selection of 64-bit versus 32-bit is not dynamic and should be done only once during **Reset**. The R4650 does not indicate externally whether the bus is configured as 32-bit or 64-bit.

Reserved bits and reserved fields in the command or data identifier should be set to 1 for system interface commands and data identifiers associated with external requests. For system interface commands and data identifiers associated with processor requests, reserved bits and reserved fields in the command and data identifier are undefined.

Command and Data Identifier Syntax

System interface commands and data identifiers are encoded in 9 bits and are transmitted on the **SysCmd** bus from the processor to an external agent, or from an external agent to the processor, during address and data cycles. Bit 8 (the most-significant bit) of the **SysCmd** bus determines whether the current content of the **SysCmd** bus is a command or a data identifier and, therefore, whether the current cycle is an address cycle or a data cycle. For system interface commands, **SysCmd(8)** must be set to 0. For system interface data identifiers, **SysCmd(8)** must be set to 1.

System Interface Command Syntax

This section describes the **SysCmd** bus encoding for system interface commands. Figure 14.17 shows a common encoding used for all system interface commands.

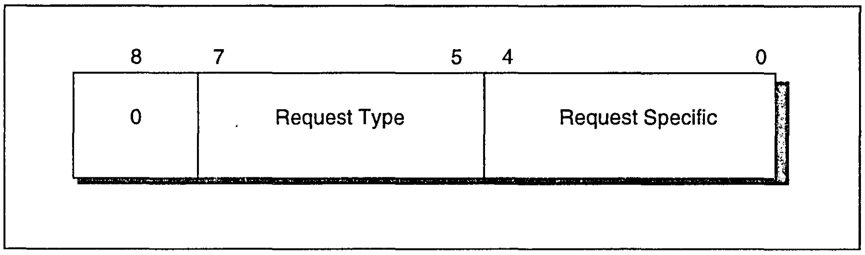


Figure 14.17 System Interface Command Syntax Bit Definition

SysCmd(8) must be set to 0 for all system interface commands. **SysCmd(7:5)** specify the system interface request type which may be read, write or null.

Table 14.6 shows the types of requests encoded by the **SysCmd(7:5)** bits. **SysCmd(4:0)** are specific to each type of request.

SysCmd(7:5)	Command
0	Read Request
1	Reserved
2	Write Request
3	Null Request
4 - 7	Reserved

Table 14.6 Encoding of SysCmd(7:5) for System Interface Commands

Write Requests

Figure 14.18 shows the format of a **SysCmd** write request.

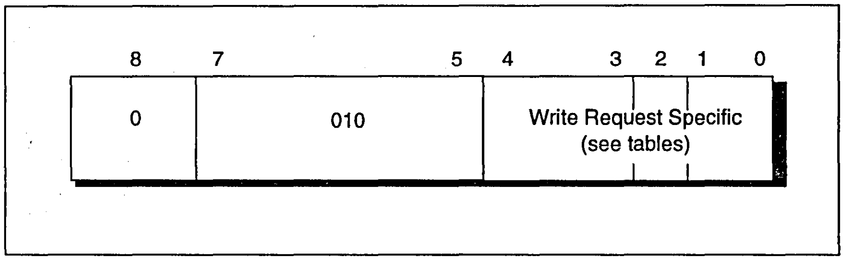


Figure 14.18 Write Request SysCmd Bus Bit Definition

Table 14.7 lists the write attributes encoded in bits **SysCmd(4:3)**.

SysCmd(4:3)	Write Attributes	
0	Reserved	
1	Reserved	
2	Block write	
3	64-bit mode: 32-bit bus mode:	Doubleword, partial doubleword, word, or partial word Word or partial word.

Table 14.7 Write Request Encoding of SysCmd(4:3)

Table 14.8 lists the block write replacement attributes encoded in bits **SysCmd(2:0)**.

SysCmd(2)	Cache Line Replacement Attributes
0	Cache line replaced
1	Cache line retained
SysCmd(1:0)	**Write Block Size**
0	Reserved
1	8 words
2 - 3	Reserved

Table 14.8 Block Write Request Encoding of SysCmd(2:0)

Table 14.9 lists the write request bit encoding in **SysCmd(2:0)**.

SysCmd(2:0)	Read Data Size
	64-bit or 32-bit bus mode:
0	1 byte valid (Byte)
1	2 bytes valid (Halfword)
2	3 bytes valid (Tribyte)
3	4 bytes valid (Word)
	64-bit mode only:
4	5 bytes valid (Quintibyte)
5	6 bytes valid (Sextibyte)
6	7 bytes valid (Septibyte)
7	8 bytes valid (Doubleword)

Table 14.9 Doubleword, Word, or Partial-Word Write Request Data Size Encoding of SysCmd(2:0)

The External Request Interface

Chapter 15

Introduction

This chapter discusses the External Request protocol and associated operations.

External requests include read, write and null requests, as shown in Figure 15.1. This section also includes a description of processor read response, a special case of an external request.

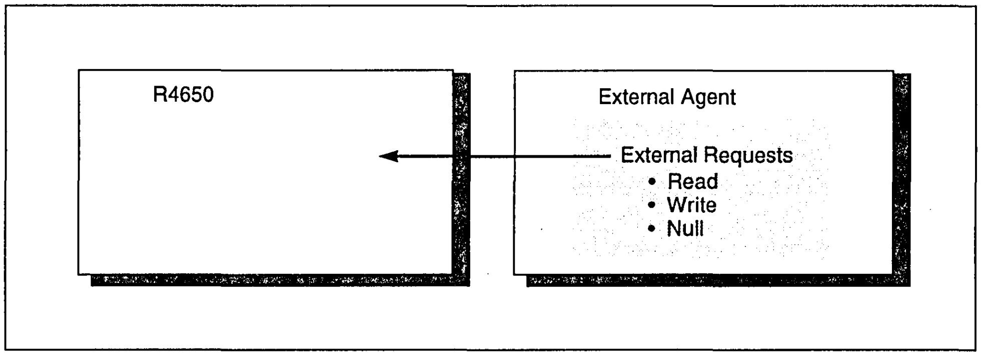


Figure 15.1 External Requests

Read request asks for a word of data from the processor's internal resource.

Write request provides a word of data to be written to the processor's internal resource.

Null request requires no action by the processor; it provides a mechanism for the external agent to return control of the system interface to the master state without affecting the processor.

The processor controls the flow of external requests through the arbitration signals **ExtRqst*** and **Release***, as shown in Figure 15.2. The external agent must acquire mastership of the system interface before it is allowed to issue an external request; the external agent arbitrates for mastership of the system interface by asserting **ExtRqst*** and then waiting for the processor to assert **Release*** for one cycle.

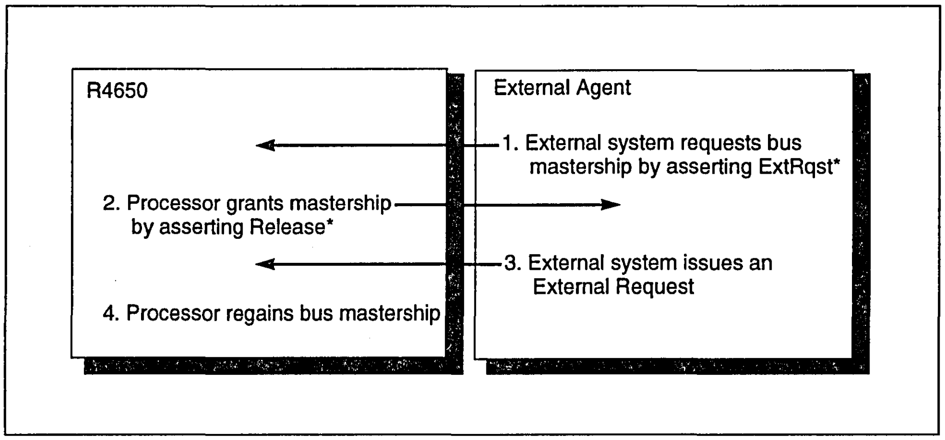

R4650
External Agent
1. External system requests bus mastership by asserting ExtRqst*
2. Processor grants mastership by asserting Release*
3. External system issues an External Request
4. Processor regains bus mastership

Figure 15.2 External Request

Mastership of the system interface always returns to the processor after an external request is issued. The processor does not accept a subsequent external request until it has completed the current request.

If there are no processor requests pending, the processor decides, based on its internal state, whether to accept the external request, or to issue a new processor request. The processor can issue a new processor request even if the external agent is requesting access to the system interface.

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

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

External Read Request

In contrast to a processor read request, data is returned directly in response to an external read request; no other requests can be issued until the processor returns the requested data. An external read request is complete after the processor returns the requested word of data.

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

Note: The R4650 does not contain any resources that are readable by an external read request; in response to an external read request the processor returns undefined data and a data identifier with its *Erroneous Data* bit, **SysCmd(5)**, set.

External Write Request

When an external agent issues a write request, the specified resource is accessed and the data is written to it. An external write request is complete after the word of data has been transmitted to the processor.

The only processor resource available to an external write request is the IP field of the Cause register.

Read Response

A *read response* returns data in response to a processor read request, as shown in Figure 15.3. While a read response is technically an external request, it has one characteristic that differentiates it from all other external requests—it does not perform system interface arbitration. For this reason, read responses are handled separately from all other external requests, and are simply called read responses. When a read response comes back with bad parity for the first datum, a cache error exception results.

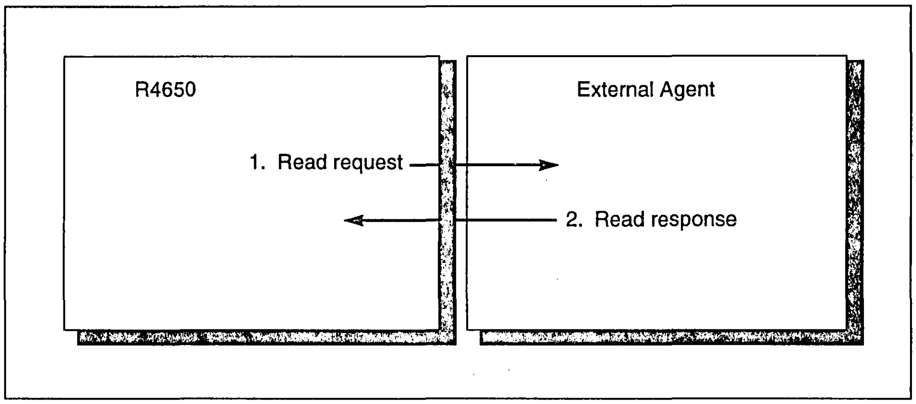


Figure 15.3 Read Response

Processor and External Request Protocols

The following sections contain a cycle-by-cycle description of the bus arbitration protocols for each type of processor and external request. lists the abbreviations and definitions for each of the buses that are used in the timing diagrams that follow.

Scope	Abbreviation	Meaning
Global	Unsd	Unused
SysAD bus	Addr	Physical address
	Data<n>	Data element number n of a block of data
SysCmd bus	Cmd	An unspecified system interface command
	Read	A processor or external read request command
	Write	A processor or external write request command
	SINull	A system interface release external null request command
	NData	A noncoherent data identifier for a data element other than the last data element
	NEOD	A noncoherent data identifier for the last data element

Table 15.1 System Interface Requests

External Request Protocols

This section describes the following external request protocols:

- read
- null
- write
- read response

External requests can only be issued with the system interface in slave state. An external agent asserts **ExtRqst*** to arbitrate (see the "External Arbitration Protocol" subsection) for the system interface, then waits for the processor to release the system interface to slave state by asserting **Release*** before the external agent issues an external request. If the system interface is already in slave state (that is, the processor has previously performed an uncompelled change to slave state due to a read operation) the external agent can begin an external request immediately.

After issuing an external request, the external agent must return the system interface to master state. If the external agent does not have any additional external requests to perform, **ExtRqst*** must be deasserted two cycles after the cycle in which **Release*** was asserted. For a string of external requests, the **ExtRqst*** signal is asserted until the last request cycle, whereupon it is deasserted two cycles after the cycle in which **Release*** was asserted.

The processor continues to handle external requests as long as **ExtRqst*** is asserted; however, the processor cannot release the system interface to slave state for a subsequent external request until it has completed the current request. As long as **ExtRqst*** is asserted, the string of external requests is not interrupted by a processor request. The protocol is the same for either 64-bit or 32-bit bus interface mode.

External Arbitration Protocol

System interface arbitration uses the signals **ExtRqst*** and **Release*** as described above. Figure 15.4 is a timing diagram of the arbitration protocol, in which slave and master states are shown.

The arbitration cycle consists of the following steps:

1. The external agent asserts **ExtRqst*** when it wishes to submit an external request.
2. The processor waits until it is ready to handle an external request, whereupon it asserts **Release*** for one cycle.
3. The processor sets the **SysAD** and **SysCmd** buses to tri-state.
4. The external agent must begin driving the **SysAD** bus and the **SysCmd** bus two cycles after the assertion of **Release***.
5. The external agent deasserts **ExtRqst*** two cycles after the assertion of **Release***, unless the external agent wishes to perform an additional external request.
6. The external agent sets the **SysAD** and the **SysCmd** buses to tri-state at the completion of an external request.

The processor can start issuing a processor request one cycle after the external agent sets the bus to tri-state.

Note: Timings for the **SysADC** and **SysCmdP** buses are the same as those for the **SysAD** and **SysCmd** buses, respectively. The protocol is the same for 64-bit and 32-bit bus interface mode.

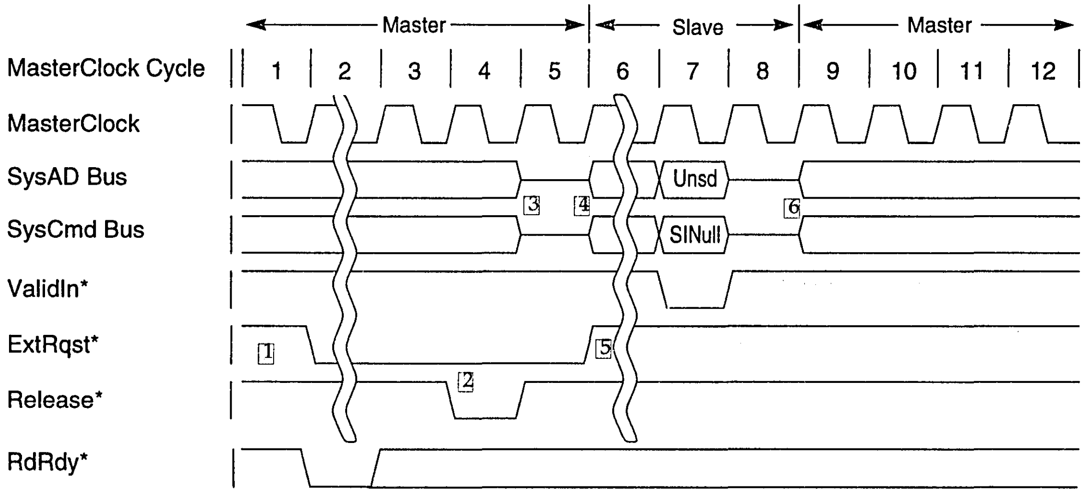


Figure 15.4 Arbitration Protocol for External Requests

External Read Request Protocol

External reads are requests for a word of data from a processor internal resource, such as a register. External read requests cannot be split; that is, no other request can occur between the external read request and its read response.

Figure 15.5 shows a timing diagram of an external read request, which consists of the following steps:

1. An external agent asserts **ExtRqst*** to arbitrate for the system interface.
2. The processor releases the system interface to slave state by asserting **Release*** for one cycle and then deasserting **Release*.**
3. After **Release*** is deasserted, the **SysAD** and **SysCmd** buses are set to a tri-state for one cycle.
4. The external agent drives a read request command on the **SysCmd** bus and a read request address on the **SysAD** bus and asserts **ValidIn*** for one cycle.
5. After the address and command are sent, the external agent releases the **SysCmd** and **SysAD** buses by setting them to tri-state and allowing the processor to drive them. The processor, having accessed the data that is the target of the read, returns this 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 last-data-cycle response data.
6. The system interface is in master state. The processor continues driving the **SysCmd** and **SysAD** buses after the read response is returned.

Note: Timings for the **SysADC** and **SysCmdP** buses are the same as those of the **SysAD** and **SysCmd** buses, respectively.

External read requests are only allowed to read a (32-bit) word of data from the processor. The processor response to external read requests is undefined for any data element other than a word. In 64-bit or 32-bit bus mode this operation is only a single external read request to the processor. In both modes **SysAD(31:0)** provides the address of the internal resource that is to be read.

Note: The processor does not contain any resources that are readable by an external read request. In response to an external read request the processor returns undefined data and a data identifier that has its *erroneous data bit*, **SysCmd(5)**, set. This will also cause the CPU to take an error data exception.

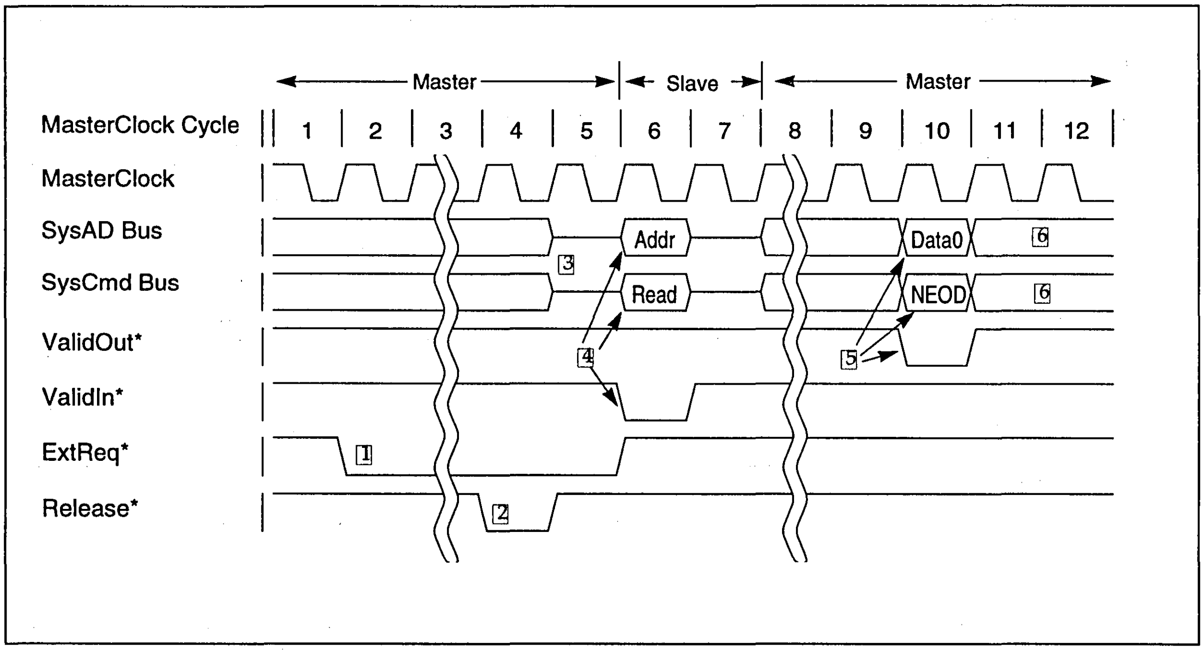


Figure 15.5 External Read Request, System Interface in Master State

External Null Request Protocol

The R4650 only supports one external null request. A *system interface release external null request* returns the system interface to master state from slave state without otherwise affecting the processor.

External null requests require no action from the processor other than to return the system interface to master state.

Figure 15.6 show timing diagram of the external null request cycle, which consist of the following steps:

1. The external agent asserts **ExtRqst*** to arbitrate for the system interface.
2. The processor releases the system interface to slave state by asserting **Release***.
3. The external agent drives a system interface release external null request command on the **SysCmd** bus, and asserts **ValidIn*** for one cycle to return the system interface back to master state.
4. The **SysAD** bus is unused (does not contain valid data) during the address cycle associated with an external null request.
5. 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 expects the system interface to return to master state. This protocol is the same for both 64-bit and 32-bit bus modes.

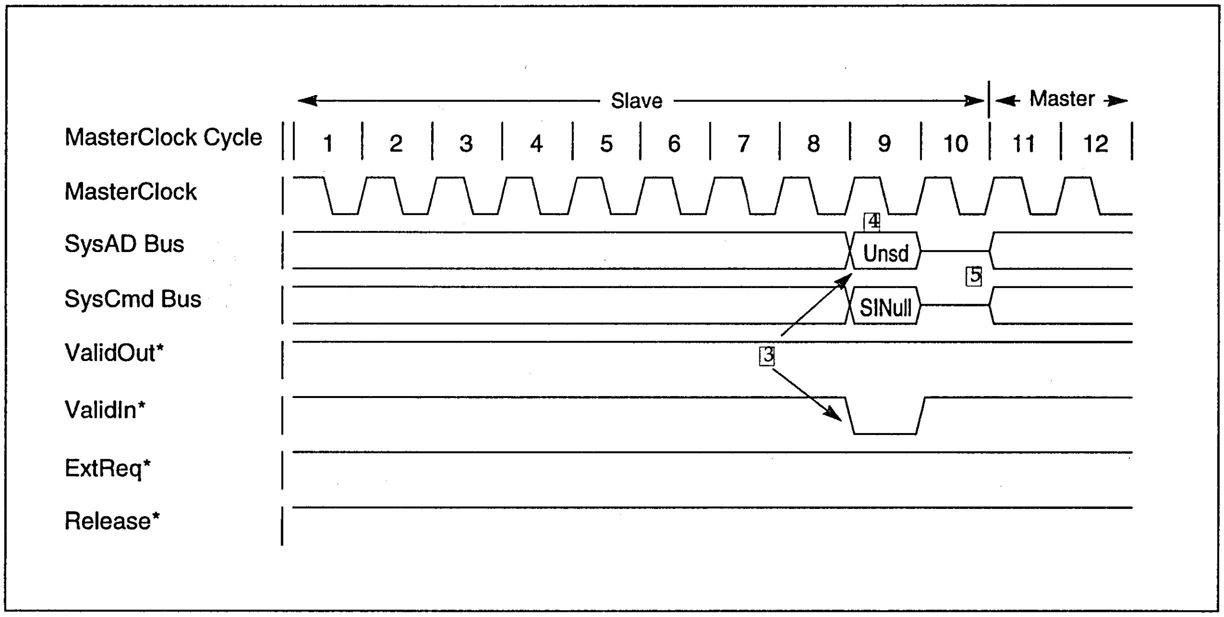


Figure 15.6 System Interface Release External Null Request

External Write Request Protocol

External write requests use a protocol identical to the processor single word write protocol except the **ValidIn*** signal is asserted instead of **ValidOut***. Figure 15.7 on page 8 shows a timing diagram of an external write request, which consists of the following steps:

1. The external agent asserts **ExtRqst*** to arbitrate for the system interface.
2. The processor releases the system interface to slave state by asserting **Release***.
3. The external agent drives a write command on the **SysCmd** bus, a write address on the **SysAD** bus, and asserts **ValidIn***.
4. The external agent drives a data identifier on the **SysCmd** bus, data on the **SysAD** bus, and asserts **ValidIn***.
5. The data identifier associated with the data cycle must contain a coherent or noncoherent last data cycle indication.
6. After the data cycle is issued, the write request is complete and the external agent sets the **SysCmd** and **SysAD** buses to a tri-state, allowing the system interface to return to master state. Timings for the **SysADC** and **SysCmdP** buses are the same as those of the **SysAD** and **SysCmd** buses, respectively.

External write requests are only allowed to write a (32-bit) word of data to the processor. Processor behavior in response to an external write request for any data element other than a word is undefined. In 64-bit and 32-bit bus mode **SysAD(31:0)** is used for both the address and the data portions of the external write request, regardless of the “endianness” of the system.

Note: The interrupt register is the only processor internal resource available for write access by an external request.

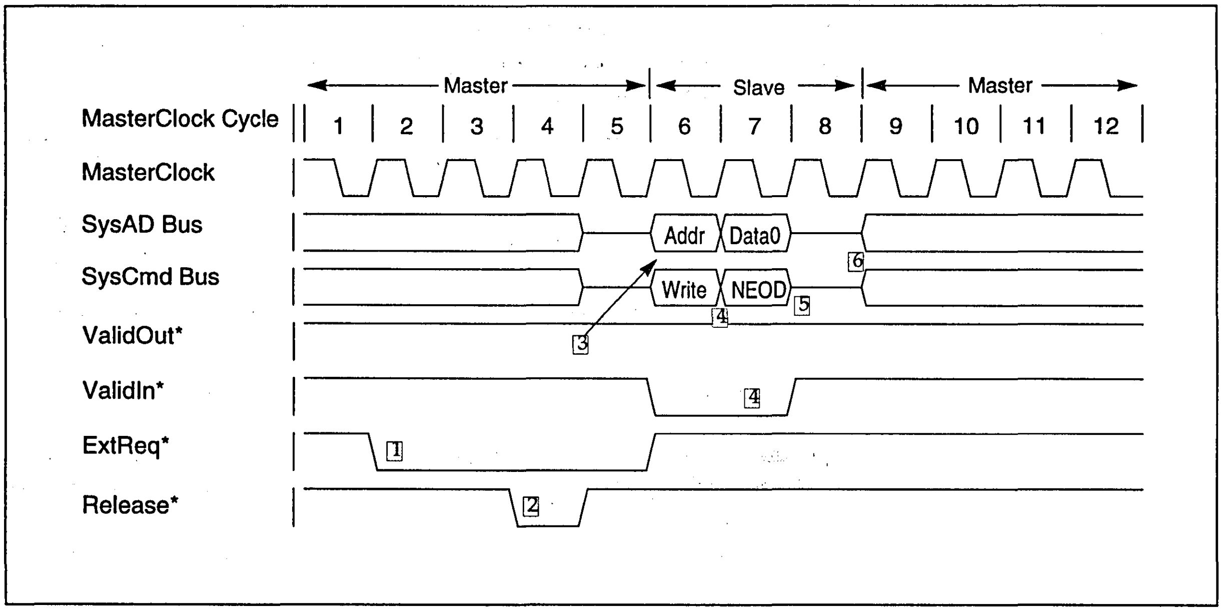


Figure 15.7 External Write Request, with System Interface Initially in Master State

Read Response Protocol

An external agent must return data to the processor in response to a processor read request by using a read response protocol. The read response protocol is discussed in detail in Chapter 13, "The Read Interface."

System Interface Commands and Data Identifiers

System interface commands specify the nature and attributes of any system interface request; this specification is made during the address cycle for the request. System interface data identifiers specify the attributes of data transmitted during a system interface data cycle.

The following sections describe the syntax, that is, the bitwise encoding, of system interface commands and data identifiers. The same **SysCmd** encoding is used for both 32-bit and 64-bit bus mode. The selection of 64-bit versus 32-bit is not dynamic and should be done only once during **Reset**. The R4650 does not indicate externally whether the bus is configured as 32-bit or 64-bit.

Reserved bits and reserved fields in the command or data identifier should be set to 1 for system interface commands and data identifiers associated with external requests. For system interface commands and data identifiers associated with processor requests, reserved bits and reserved fields in the command and data identifier are undefined.

Command and Data Identifier Syntax

System interface commands and data identifiers are encoded in 9 bits and are transmitted on the **SysCmd** bus from the processor to an external agent, or from an external agent to the processor, during address and data cycles. Bit 8 (the most-significant bit) of the **SysCmd** bus determines whether the current content of the **SysCmd** bus is a command or a data identifier and, therefore, whether the current cycle is an address cycle or a data cycle. For system interface commands, **SysCmd(8)** must be set to 0. For system interface data identifiers, **SysCmd(8)** must be set to 1.

System Interface Command Syntax

This section describes the **SysCmd** bus encoding for system interface commands. Figure 15.8 shows a common encoding used for all system interface commands.

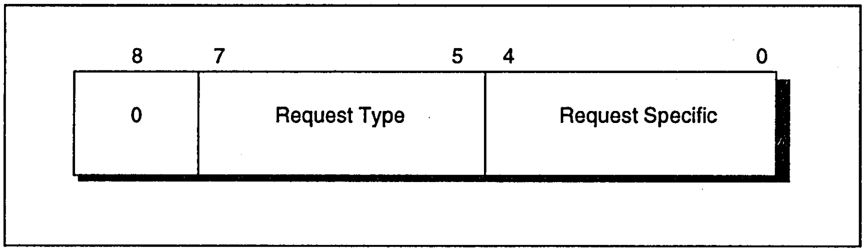


Figure 15.8 System Interface Command Syntax Bit Definition

SysCmd(8) must be set to 0 for all system interface commands.

SysCmd(7:5) specify the system interface request type which may be read, write or null; lists the encoding of **SysCmd(7:5)**.

shows the types of requests encoded by the **SysCmd(7:5)** bits.

SysCmd(7:5)	Command
0	Read Request
1	Reserved
2	Write Request
3	Null Request
4 - 7	Reserved

Table 15.2 Encoding of SysCmd(7:5) for System Interface Commands

SysCmd(4:0) are specific to each type of request and are defined in each of the following sections.

Null Requests

Figure 15.9 shows the format of a **SysCmd** null request.

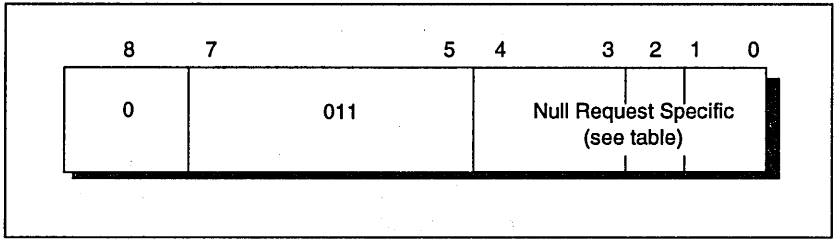


Figure 15.9 Null Request SysCmd Bus Bit Definition

System interface release external null requests use the null request command. lists the encoding of **SysCmd(4:3)** for external null requests. **SysCmd(2:0)** are reserved for both instances of null requests.

SysCmd(4:3)	Null Attributes
0	System Interface release
1 - 3	Reserved

Table 15.3 External Null Request Encoding of SysCmd(4:3)

System Interface Data Identifier Syntax

This section defines the encoding of the **SysCmd** bus for system interface data identifiers. Figure 15.10 shows a common encoding scheme used for all system interface data identifiers.

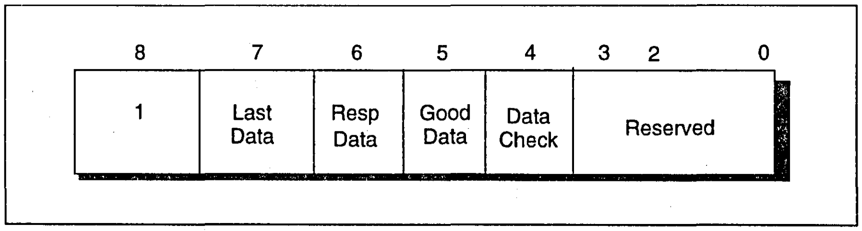


Figure 15.10 Data Identifier SysCmd Bus Bit Definition

SysCmd(8) must be set to 1 for all system interface data identifiers. system interface data identifiers use the format for noncoherent data.

Noncoherent Data

Noncoherent data is defined as follows:

- data that is associated with processor block write requests and processor doubleword, partial doubleword, word, or partial word write requests
- data that is returned in response to a processor noncoherent block read request or a processor doubleword, partial doubleword, word, or partial word read request
- data that is associated with external write requests
- data that is returned in response to an external read request

Data Identifier Bit Definitions

SysCmd(7) marks the last data element and **SysCmd(6)** indicates whether or not the data is response data, for both processor and external coherent and noncoherent data identifiers. Response data is data returned in response to a read request.

SysCmd(5) indicates whether or not the data element is error free. Erroneous data contains an uncorrectable error and is returned to the processor, forcing a bus error. The processor delivers data with the good data bit deasserted if a primary parity error is detected for a transmitted data item.

SysCmd(4) indicates to the processor whether to check the data and check bits for this data element.

SysCmd(3) is reserved for external data identifiers.

SysCmd(4:3) are reserved for noncoherent processor data identifiers.

SysCmd(2:0) are reserved for noncoherent data identifiers.

Table 15.4 lists the 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 15.4 Processor Data Identifier Encoding of SysCmd(7:3)

lists the 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	Do not check the data and check bits
SysCmd(3)	Reserved

Table 15.5 External Data Identifier Encoding of SysCmd(7:3)

System Interface Addresses

System interface addresses are full 32-bit physical addresses presented on the least-significant 32 bits (bits 31 through 0) of the **SysAD** bus during address cycles; the remaining bits of the **SysAD** bus are unused during address cycles.

Addressing Conventions

Addresses associated with doubleword, partial doubleword, word, or partial word transactions, are aligned for the size of the data element. The system uses the following address conventions:

- Addresses associated with block requests are aligned to double-word boundaries; that is, the low-order 3 bits of address are 0.
- Doubleword requests set the low-order 3 bits of address to 0.
- Word requests set the low-order 2 bits of address to 0.
- Halfword requests set the low-order bit of address to 0.
- Byte, tribyte, quintibyte, sextibyte, and septibyte requests use the byte address.

Processor Internal Address Map

External reads and writes provide access to processor internal resources that may be of interest to an external agent. The processor decodes bits **SysAD(6:0)** of the address associated with an external read or write request to determine which processor internal resource is the target.

However, the R4650 does not contain any resources that are *readable* through an external read request. In response to an external read request the processor returns 1) undefined data, 2) a data identifier that has its *Erroneous Data* bit, **SysCmd(5)**, set, and then 3) takes an exception.

The *Interrupt* register is the only processor internal resource available for *write* access by an external request. The *Interrupt* register is accessed by an external write request with an address of 000_2 on bits 6:4 of the **SysAD** bus.

The interrupt register is described in detail in Chapter 16, "R4650 Processor Interrupts."

R4650 Processor Interrupts

Introduction

The R4650 processor supports the following interrupts: six hardware interrupts, one internal "timer interrupt," two software interrupts, and one unmasked/nonmaskable enabled interrupt. The processor takes an exception on any interrupt.

This chapter describes the six hardware and single nonmaskable interrupts. A description of the software and the timer interrupts can be found in Chapter 5. CPU exception processing is also described in Chapter 5. Floating-point exception processing is described in Chapter 6.

Hardware Interrupts

The six CPU hardware interrupts can be caused by external write requests to the R4650, or can be caused through dedicated interrupt pins. These pins are latched into an internal register by the rising edge of **MasterClock**.

Nonmaskable Interrupt (NMI)

The nonmaskable interrupt is caused either by an external write request to the R4650 or by a dedicated pin in the R4650. This pin is latched into an internal register by the rising edge of **MasterClock**.

Asserting Interrupts

External writes to the CPU are directed to various internal resources, based on an internal address map of the processor. When **SysAD[6:0]** = 0 during an ADDR cycle of external write request, an external write to any address writes to an architecturally transparent register called the *Interrupt* register; this register is available for external write cycles, but not for external reads.

During a data cycle, **SysAD[22:16]** are the write enables for the seven individual *Interrupt* register bits (0 = disabled, 1 = enabled) and **SysAD[6:0]** are the values to be written into these bits (0 = no interrupt, 1 = interrupt). This allows any subset of the *Interrupt* register to be set or cleared with a single write request. Figure 16.1 shows the mechanics of an external write to the *Interrupt* register.

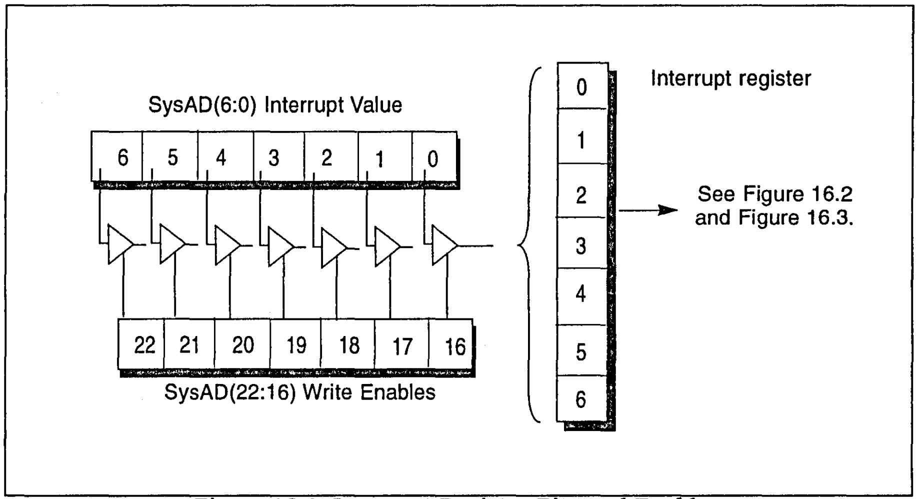

SysAD(6:0) Interrupt Value
6
5
4
3
2
1
0
22
21
20
19
18
17
16
SysAD(22:16) Write Enables
0
1
2
3
4
5
6
Interrupt register
See Figure 16.2 and Figure 16.3.

Figure 16.1 Interrupt Register Bits and Enables

Figure 16.2 shows how the R4650 interrupts are readable through the *Cause* register. The interrupt bits, **Int*(5:0)**, are latched into the internal register by the rising edge of **MasterClock**.

- Bit 5 of the *Interrupt* register in the R4650 is ORed with the **Int*(5)** pin and then multiplexed with the internal **TimerInterrupt** signal. This 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.

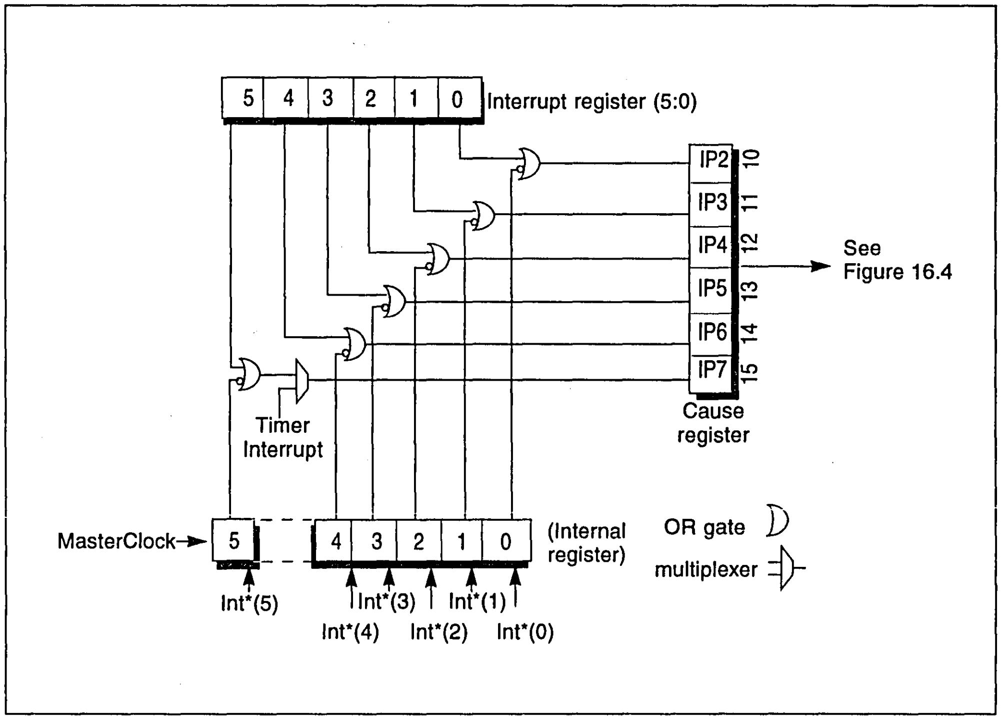


Figure 16.2 R4650 Interrupt Signals

Figure 16.3 shows the internal derivation of the nonmaskable **(NMI)** signal, for the R4650 processor.

The **NMI*** pin is latched into an internal register by the rising edge of **MasterClock**. Bit 6 of the *Interrupt* register is then ORed with the inverted value of **NMI*** to form the nonmaskable interrupt. Only the one falling edge of the latched signal will cause the NMI.

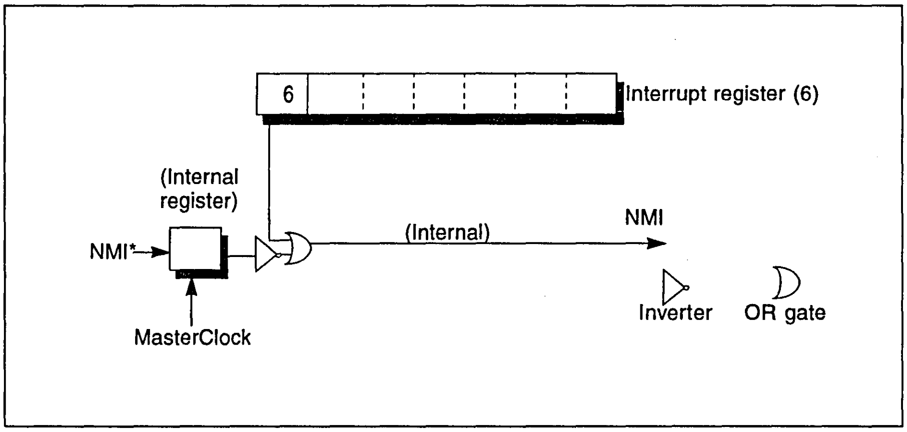


Figure 16.3 R4650 Nonmaskable Interrupt Signal

Figure 16.4 shows the masking of the R4650 interrupt signal.

- *Cause* register bits 15:8 (IP7-IP0) are AND-ORed with *Status* register interrupt mask bits 15:8 (IM7-IM0) to mask individual interrupts.
- *Status* register bit 0 is a global Interrupt Enable (IE). It is ANDed with the output of the AND-OR logic to produce the R4650 interrupt signal.

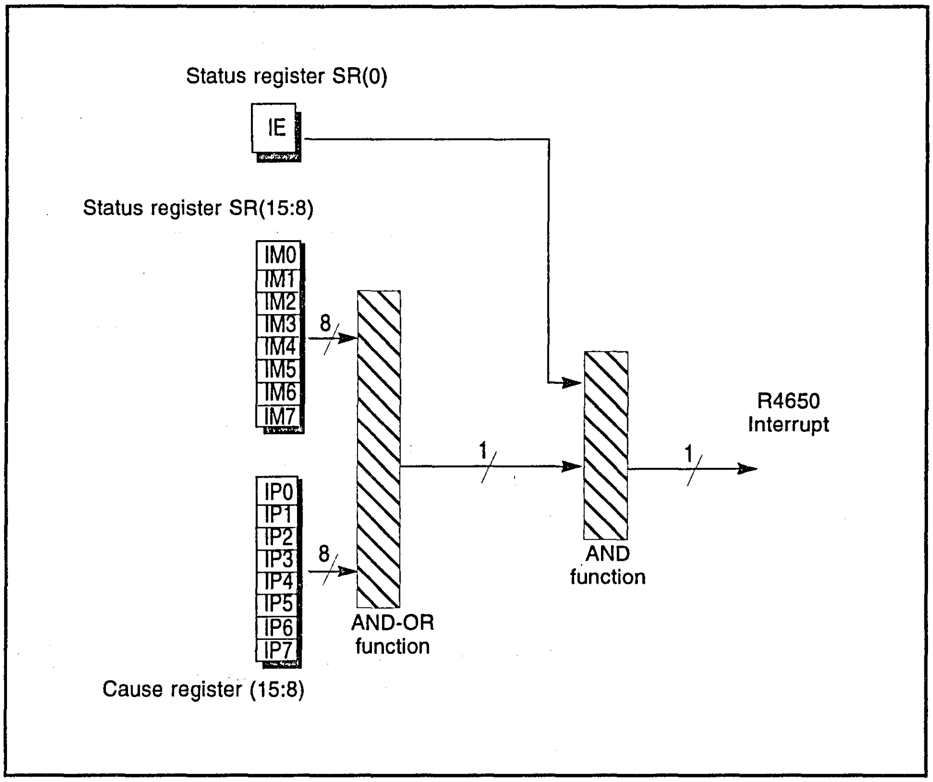


Figure 16.4 Masking of the R4650 Interrupts

Introduction

This chapter describes the Error Checking mechanism used in the R4650 processor.

Error Checking in the Processor

Error checking codes allow the processor to detect and sometimes correct errors made when moving data from one place to another.

Two major types of data errors can occur in data transmission:

- hard errors, which are permanent, arise from broken interconnects, internal shorts, or open leads
- soft errors, which are transient, are caused by system noise, power surges, and alpha particles.

Hard errors must be corrected by physical repair of the damaged equipment and restoration of data from backup. Soft errors can be corrected by using error checking and correcting codes.

Types of Error Checking

The R4650 uses even parity (error detection only).

Parity Error Detection

Parity is the simplest error detection scheme. By appending a bit to the end of an item of data—called a *parity bit*—single bit errors can be detected; however, these errors cannot be corrected.

There are two types of parity:

- **Odd Parity** adds 1 to any even number of 1s in the data, making the total number of 1s odd (including the parity bit).
- **Even Parity** adds 1 to any odd number of 1s in the data, making the total number of 1s even (including the parity bit).

Odd and even parity are shown in the example below:

Data(3:0)	Odd Parity Bit	Even Parity Bit
0 0 1 0	0	1

This example shows a single bit in **Data(3:0)** with a value of 1; this bit is **Data(1)**.

- In even parity, the parity bit is set to 1. This makes 2 (an even number) the total number of bits with a value of 1.
- Odd parity makes the parity bit a 0 to keep the total number of 1-value bits an odd number—in the case shown above, the single bit **Data(1)**.

The example below shows odd and even parity bits for various data values:

Data(3:0)	Odd Parity Bit	Even Parity Bit
0 1 1 0	1	0
0 0 0 0	1	0
1 1 1 1	1	0
1 1 0 1	0	1

Parity allows single-bit error detection, but it does not indicate which bit is in error—for example, suppose an odd-parity value of 00011 arrives. The last bit is the parity bit, and since odd parity demands an odd number (1,3,5) of 1s, this data is in error: it has an even number of 1s. However it is impossible to tell *which* bit is in error.

Error Checking Operation

The processor verifies data correctness by using even parity as it passes data from/to the system interface to/from the primary caches.

System Interface

The processor generates correct check bits for doubleword, word, or partial-word data transmitted to the system interface. As it checks for data correctness, the processor passes data check bits from the primary cache, directly without changing the bits, to the system interface.

The processor does not check data received from the system interface for external writes. By setting the *NChck* bit in the data identifier, it is possible to prevent the processor from checking read response data from the system interface.

For cache refill, if the NChck bit is set, the CPU will generate correct parity before placing data into the cache. The R4650 only checks parity for the first double word returned on a block instruction fetch, that is, for the double word that contains the instruction that was missed on in the cache. This double word is checked just as if it had been read out of the cache. This parity check is done as a byte parity check. For single read, and with the NChck bit set, the CPU will check parity for all 64-bit, even if the transfer size is less than that.

When the R4650 is checking parity it does not actually regenerate the word parity, but rather turns the byte parity supplied by the system into word parity. It XORS the bits in groups of four. As a result, if bad byte parity is supplied by the system, bad word parity will get written into the cache. This is done to be consistent with what happens in the DCache.

The processor does not check addresses received from the system interface and does not 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 it detects an error based on data check bits. Software is responsible for error handling.

System Interface Command Bus

In the R4650 processor, the system interface command bus has no parity. **SysCmdP** always drives zero out for CPU valid cycles and is not checked when the system interface is in slave state.

Summary of Error Checking Operations

Error Checking operations are summarized in Table 17.1 and Table 17.2.

Bus	Uncached Load	Uncached Store	Primary Cache Load from System Interface	Primary Cache Write to System Interface	Cache Instruction
Processor Data	From System Interface	Not Checked	From System Interface unchanged	Checked; Trap on Error	Check on cache write-back; Trap on Error
System Interface Address/Command and Check Bits: Transmit	Not Generated	Not Generated	Not Generated	Not Generated	Not Generated
System Interface Address/Command and Check Bits: Receive	Not Checked	NA	Not Checked	NA	NA
System Interface Data	Checked; Trap on Error	From Processor	Checked; Trap on Error	From Primary Cache	From Primary Cache
System Interface Data Check Bits	Checked; Trap on Error	Generated	Checked; Trap on Error	From Primary Cache	From Primary Cache

Table 17.1 Error Checking and Correcting Summary for Internal Transactions

Bus	Read Request	Write Request
Processor Data	NA	NA
System Interface Address, Command, and Check Bits: Transmit	Generated	NA
System Interface Address, Command, and Check Bits: Receive	Not Checked	Not Checked
System Interface Data	From Processor	Checked; Trap on Error
System Interface Data Check Bits	Generated	Checked; Trap on Error

Table 17.2 Error Checking and Correcting Summary for External Transactions

CPU Instruction Set Details

Appendix A

Introduction

This appendix provides a detailed description of the operation of each R4650 instruction. The instructions are listed in alphabetical order.

Exceptions that may occur due to the execution of each instruction are listed after the description of each instruction. Descriptions of the immediate cause and manner of handling exceptions are omitted from the instruction descriptions in this appendix.

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.

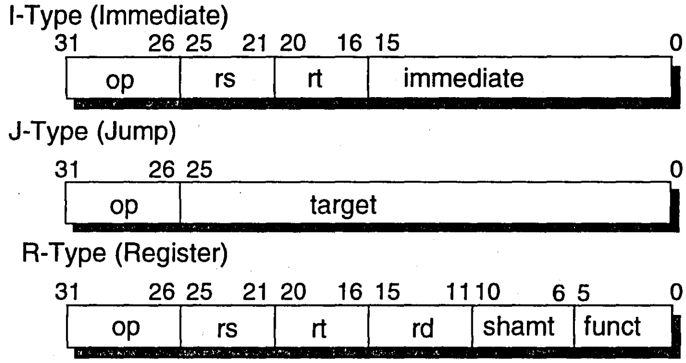
Instruction Classes

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 made 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 their 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 in Appendix B). 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.



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

Special symbols used in the notation are described in Table A. 1

Symbol	Meaning
←	Assignment.
||	Bit string concatenation.
x_y	Replication of bit value x into a y-bit string. Note: x is always a single-bit
$x_{y:z}$	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.
+	2's complement or floating-point addition.
-	2's complement or floating-point subtraction.
*	2's complement or floating-point multiplication.
div	2's complement integer division.
mod	2's complement modulo.
/	Floating-point division.
<	2's complement less than comparison.
and	Bit-wise logical AND.
or	Bit-wise logical OR.
xor	Bit-wise logical XOR.
nor	Bit-wise logical 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 endianness of the memory interface (see LoadMemory and StoreMemory), and the endianness of Kernel and Supervisor mode execution.
ReverseEndian	Signal to reverse the endianness of load and store instructions in User mode; effected by setting the *RE* bit of the *Status* register. Thus, ReverseEndian may be computed as (SR_{25} and User mode*).*
BigEndianCPU	The endianness for load and store instructions (0 → Little, 1 → Big). In User mode, this endianness 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.

Table A. 1 CPU Instruction Operation Notations

Instruction Notation Examples

The following examples illustrate the application of some of the instruction notation conventions:

Example #1:

$$GPR[rt] \leftarrow \text{immediate} \mid\mid 0^{16}$$

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:

$$(\text{immediate}_{15})^{16} \mid\mid \text{immediate}_{15..0}$$

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 R4650, as in the case of processors, the instruction immediately following a load may use the loaded contents of the register. 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 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 descriptions, the functions listed in Table A.2 are used to summarize the handling of virtual addresses and physical memory.

Function	**Meaning**
AddressTranslation	Uses the CP0 to find the physical address given the virtual address. The function fails and an exception is taken if the required translation is not present/allowed.
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.

Table A.2 Load and Store Common Functions

As shown in Table A.2, the *Access Type* field indicates the size of the data item to be loaded or stored. Regardless of access type or byte-numbering order (endianness), the address specifies the byte which has the smallest byte address 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.

Access Type Mnemonic	Value	Meaning
DOUBLEWORD	7	8 bytes (64 bits)
SEPTIBYTE	6	7 bytes (56 bits)
SEXTIBYTE	5	6 bytes (48 bits)
QUINTIBYTE	4	5 bytes (40 bits)
WORD	3	4 bytes (32 bits)
TRIPLEBYTE	2	3 bytes (24 bits)
HALFWORD	1	2 bytes (16 bits)
BYTE	0	1 byte (8 bits)

Table A.3 Access Type Specifications for Loads/Stores

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.

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 (that is, occupying the delay slot) is always executed while the target instruction is being fetched from storage. A delay slot may not itself be occupied 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 that 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

Coprocessors are alternate execution units, which have register files separate from the CPU. The R4650 architecture (MIPS III) provides three coprocessor units, or classes, and these coprocessors have two register spaces, each space containing thirty-two registers. These registers may be either 32-bits or 64-bits wide.

- 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 space, *coprocessor control* registers, may only have their contents transferred directly between the coprocessor and the processor. Coprocessor instructions may alter registers in either space.

System Control Coprocessor (CP0) Instructions

There are some special limitations imposed on operations involving CP0 that is incorporated within the CPU. The move to/from coprocessor instructions are the only valid mechanism for writing to and reading from the CP0 registers.

Several CP0 instructions are defined to directly read, write, and 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*. 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:

```
T:   temp ← GPR[rs] + GPR[rt]
     GPR[rd] ← $(\text{temp}_{31})^{32} \| \text{temp}_{31..0}$
```

Exceptions:

Integer overflow exception

ADDI Add Immediate ADDI

31 26	25 21	20 16	15 0
ADDI 0 0 1 0 0 0	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*. The *rs* 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:

```
T:  $temp \leftarrow GPR[rs] + (immediate_{15})^{48} \mid\mid immediate_{15..0}$
    $GPR[rt] \leftarrow (temp_{31})^{32} \mid\mid temp_{31..0}$
```

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. The *rs* 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:

```
T:    $temp \leftarrow GPR[rs] + (immediate_{15})^{48} \mid\mid immediate_{15..0}$
      $GPR[rt] \leftarrow (temp_{31})^{32} \mid\mid temp_{31..0}$
```

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. The source 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:

```
T:   $\text{temp} \leftarrow \text{GPR[rs]} + \text{GPR[rt]}$
     $\text{GPR[rd]} \leftarrow (\text{temp}_{31})^{32} \,||\, \text{temp}_{31..0}$
```

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:

```
T:      GPR[rd] ← GPR[rs] and GPR[rt]
```

Exceptions:

None

ANDI And Immediate ANDI

31 26	25 21	20 16	15 0
ANDI 0 0 1 1 0 0	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:

T: $\mathrm{GPR}[rt] \leftarrow 0^{48} \,\|\, (\mathrm{immediate\ and\ GPR}[rs]_{15..0})$

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 internal condition signal is sampled during the previous instruction, there must be at least one instruction between this instruction and a coprocessor instruction that changes the internal condition signal.

Operation:

```
T–1:  condition ← not COC[z]
T:    target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
T+1:  if condition then
              PC ← PC + target
      endif
```

Note: *See the table "Opcode Bit Encoding" on next page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

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	
Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC1F	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	
Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC2F	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	

Opcode (bits 31–28)
Coprocessor Unit Number (bits 27–26)
BC sub-opcode (bits 25–21)
Branch condition (bits 20–16)

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 signal, 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 internal condition signal is sampled during the previous instruction, there must be at least one instruction between this instruction and a coprocessor instruction that changes the internal condition signal.

Note: *See the table "Opcode Bit Encoding" on next page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

Operation:

```
T–1:  condition ← not COC[z]
T:    target ← $(\text{offset}_{15})^{46}$ || 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	
Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC1FL	0	1	0	0	0	1	0	1	0	0	0	0	0	0	1	0	
Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC2FL	0	1	0	0	1	0	0	1	0	0	0	0	0	0	1	0	

Opcode

Coprocessor Unit Number

BC sub-opcode Branch condition

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

Note: *See "Opcode Bit Encoding" on this page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

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 internal condition signal is sampled during the previous instruction, there must be at least one instruction between this instruction and a coprocessor instruction that changes the internal condition signal.

Operation:

```
T–1:  condition ← COC[z]
T:    target ← $(offset_{15})^{46}$ || offset || $0^2$
T+1:  if condition then
                    PC ← PC + target
      endif
```

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	

Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC1T	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	1	

Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC2T	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	1	

Opcode BC sub-opcode Branch condition

Coprocessor Unit Number

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

Note: *See "Opcode Bit Encoding" on this page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

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 signal, 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 internal condition signal is sampled during the previous instruction, there must be at least one instruction between this instruction and a coprocessor instruction that changes the internal condition signal.

Operation:

```
T–1:  condition ← COC[z]
T:    target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
T+1:  if condition then
            PC ← PC + target
      else
            NullifyCurrentInstruction
      endif
```

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	

Opcode

Coprocessor Unit Number

BC sub-opcode

Branch condition

BEQ Branch On Equal BEQ

31 26	25 21	20 16	15 0
BEQ 000100	rs	rt	offset
6	5	5	16

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:

```
T:    target ← $(\text{offset}_{15})^{46} \,||\, \text{offset} \,||\, 0^2$
      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:

```
T:      target ← $(\text{offset}_{15})^{46}$ || 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:

```
T:      target ← $(offset_{15})^{46}$ || offset || $0^2$
        condition ← $(GPR[rs]_{63} = 0)$
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:

```
T:    $target \leftarrow (offset_{15})^{46} \| offset \| 0^2$
      $condition \leftarrow (GPR[rs]_{63} = 0)$
      $GPR[31] \leftarrow PC + 8$
T+1:  if condition then
      $PC \leftarrow PC + target$
      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:

```
T:    target ← $(\text{offset}_{15})^{46} \,\|\, \text{offset} \,\|\, 0^2$
      condition ← $(\text{GPR}[\text{rs}]_{63} = 0)$
      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:

```
T:      target ← $(\text{offset}_{15})^{46}$ || 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 000111	rs	0 00000	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:

```
T:       target ← $(\text{offset}_{15})^{46}$ || 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 010111	rs	0 00000	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:

```
T:     target ← $(\text{offset}_{15})^{46}$ || 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:

```
T:    target ← $(\text{offset}_{15})^{46}$ || 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:

```
T:      target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
        condition ← ($\text{GPR[rs]}_{63}$ = 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:

```
T:      target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
        condition ← $(\text{GPR}[\text{rs}]_{63} = 1)$
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:

```
T:    target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
      condition ← $(\text{GPR}[\text{rs}]_{63} = 1)$
      GPR[31] ← PC + 8
T+1:  if condition then
                PC ← 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:

```
T:     target ← $(offset_{15})^{46}$ || offset || $0^2$
       condition ← $(GPR[rs]_{63} = 1)$
       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:

```
T:    target ← $(\text{offset}_{15})^{46} \,||\, \text{offset} \,||\, 0^2$
      condition ← $(\text{GPR}[rs]_{63} = 1)$
T+1:  if condition then
                 PC ← PC + target
      else
                 NullifyCurrentInstruction
      endif
```

Exceptions:

None

BNE Branch On Not Equal BNE

31 26	25 21	20 16	15 0
BNE 000101	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:

```
T:     target ← $(offset_{15})^{46}$ || 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:

```
T:      target ← $(\text{offset}_{15})^{46}$ II offset II $0^2$
        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:

```
T:    BreakpointException
```

Exceptions:

Breakpoint exception

CACHE Cache CACHE

31 26	25 21	20 16	15 0
CACHE 101111	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, 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 is undefined. The operation of this instruction on uncached addresses is also undefined.

The R4650 uses only the tag comparisons, not the valid bits, to choose which data it supplies to the instruction unit. This makes it important that the tags of the A and B sets are never the same.

The Index operation uses part of the virtual address to specify a cache block, with vAddr$_{13}$ selecting the set to access.

For a primary cache of 8KB with 32 bytes per tag, vAddr$_{11..5}$ specifies the block.

Index Load Tag also uses vAddr$_{4..3}$ to select the doubleword for reading parity. When the CE bit of the Status register is set, Hit WriteBack, Hit WriteBack Invalidate, Index WriteBack Invalidate, and Fill also use vAddr$_{4..3}$ to select the doubleword that has its 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 both sets are invalid or contain different addresses (a miss), no operation is performed.

Write back from a primary cache goes to memory. The address to be written is specified by the cache tag and not the translated physical address.

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 exceptions. This operation will never cause 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 - 3	NA	Undefined

Bits 20..18 (this value is listed under the **Code** column) of the instruction specify the operation as follows:

Code	Caches	Name	Operation
0	I	Index Invalidate	Set the cache state of the cache block to Invalid. Index_Invalidate_I writes the physical address of the cache op into the tag when it clears the valid bit, which is different from the R4000.
0	D	Index Write-Back Invalidate	Examine the cache state and W bit of the primary data cache block at the index specified by the virtual address. If the state is not Invalid and the W bit is set, then write back the block to memory. The address to write is taken from the primary cache tag. Set cache state of primary cache block to Invalid.
1	I, D	Index Load Tag	Read the tag for the cache block at the specified index and place it into the TagLo CP0 registers, ignoring parity errors. Also load the data parity bits into the ECC register.
2	I, D	Index Store Tag	Write the tag for the cache block at the specified index from the TagLo and TagHi CP0 registers.
3	D	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 does not contain the specified address, and the block is dirty, write it back to the 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.
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.
5	I	Fill	Fill the primary instruction cache block from 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. Uses bit 13 to pick the set.
6	D	*Hit WriteBack*	If the cache block contains the specified address, and the W bit is set, write back the data to memory and clear the W bit.
6	I	*Hit WriteBack*	If the cache block contains the specified address, write back the data unconditionally.

Operation:

```
T:   $vAddr \leftarrow ((offset_{15})^{48} \,||\, offset_{15..0}) + GPR[base]$
     $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
     CacheOp (op, vAddr, pAddr)
```

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:

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	

Opcode | Coprocessor Suboperation

Coprocessor Unit Number

COPz Coprocessor Operation COPz

31 26	25	24 0
COPz 0 1 0 0 x x*	CO 1	cofun
6	1	25

Note: *See "Opcode Bit Encoding" on this page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

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:

```
T:    CoprocessorOperation (z, cofun)
```

Exceptions:

Coprocessor unusable exception
Coprocessor interrupt or Floating-Point Exception

Opcode Bit Encoding:

COPz

Bit #	31	30	29	28	27	26	25	0
COP0	0	1	0	0	0	0	1	

Bit #	31	30	29	28	27	26	25	0
COP1	0	1	0	0	0	1	1	

Bit #	31	30	29	28	27	26	25	0
COP2	0	1	0	0	1	0	1	

Opcode
Coprocessor Unit Number
CO sub-opcode (see end of Appendix A)

CTCz Move Control to Coprocessor 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

Note: *See "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

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:

```
T:       data ← GPR[rt]
T + 1:  CCR[z,rd]  ← data
```

Exceptions:

Coprocessor unusable

DADD Doubleword Add DADD

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DADD 101100
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.

Operation:

```
T:     GPR[rd] ←GPR[rs] + GPR[rt]
```

Exceptions:

Integer overflow exception

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.

Operation:

```
T:    GPR [rt] ← GPR[rs] + $(\text{immediate}_{15})^{48}$ || $\text{immediate}_{15..0}$
```

Exceptions:

Integer overflow exception

DADDIU Doubleword Add Immediate Unsigned DADDIU

31 26	25 21	20 16	15 0
DADDIU 0 1 1 0 0 1	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 form the 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.

Operation:

$$\text{T:} \quad \text{GPR [rt]} \leftarrow \text{GPR[rs]} + (\text{immediate}_{15})^{48} \mathbin{||} \text{immediate}_{15..0}$$

Exceptions:

None

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.

Operation:

```
T:     GPR[rd] ←GPR[rs] + GPR[rt]
```

Exceptions:

None

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.

Operation:

```
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:
None

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.

Operation:

```
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:

None

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.

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.

Operation:

```
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.

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.

Operation:

```
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 in kernel mode regardless of the setting of the Status.KX bit. Execution of this instruction with in supervisor mode with Status.SX = 0 or in user mode with UX = 0, causes a reserved instruction exception. All 64-bits of the general register destination are written from the coprocessor register source. The operation of DMFC0 on a 32-bit coprocessor 0 register is undefined.

Operation:

```
T:    data ←CPR[0,rd]
T+1:  GPR[rt] ← data
```

Exceptions:

Coprocessor unusable exception

Reserved instruction exception for supervisor mode with Status.SX = 0 or user mode with Status.UX = 0.

DMTC0 Doubleword Move To System Control Coprocessor DMTC0

31 26	25 21	20 16	15 11	10 0
COP0 010000	DMT 00101	rt	rd	0 000 0000 0000
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 in kernel mode regardless of the setting of the Status.KX bit. Execution of this instruction with in supervisor mode with Status.SX = 0 or in user mode with UX = 0, causes a reserved instruction exception.

All 64-bits of the coprocessor 0 register 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 instructions and store instructions immediately prior to and after this instruction are undefined.

Operation:

```
T:      data ← GPR[rt]
T+1:  CPR[0,rd] ← data
```

Exceptions:

Reserved instruction exception for supervisor mode with Status.SX = 0 or user mode with Status.UX = 0.

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.

Operation:

```
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:

None

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.

Operation:

```
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:
None

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:

```
T:    $s \leftarrow 0 \,||\, sa$
      $GPR[rd] \leftarrow GPR[rt]_{(63-s)..0} \,||\, 0^s$
```

Exceptions:

None

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 in general register *rs*, inserting zeros into the low-order bits. The result is placed in register *rd.*

Operation:

```
T:    $s \leftarrow GPR[rs]_{5..0}$
      $GPR[rd] \leftarrow GPR[rt]_{(63-s)..0} \; || \; 0^{s}$
```

Exceptions:

None

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.*

Operation:

```
T:   $s \leftarrow 1 \,||\, sa$
     $GPR[rd] \leftarrow GPR[rt]_{(63-s)..0} \,||\, 0^{s}$
```

Exceptions:
None

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*.

Operation:

```
T:   $s \leftarrow 0 \,\|\, sa$
     $GPR[rd] \leftarrow (GPR[rt]_{63})^{s} \,\|\, GPR[rt]_{63..s}$
```

Exceptions:

None

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.*

Operation:

```
T:    s ← $GPR[rs]_{5..0}$
      $GPR[rd] \leftarrow (GPR[rt]_{63})^{s} \,||\, GPR[rt]_{63..s}$
```

Exceptions:

None

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.*

Operation:

```
T:   $s \leftarrow 1 \| sa$
     $GPR[rd] \leftarrow (GPR[rt]_{63})^{s} \| GPR[rt]_{63..s}$
```

Exceptions:

None

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.*

Operation:

```
T:    $s \leftarrow 0 \| sa$
      $GPR[rd] \leftarrow 0^s \| GPR[rt]_{63..s}$
```

Exceptions:

None

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*.

Operation:

```
T:   $s \leftarrow GPR[rs]_{5..0}$
     $GPR[rd] \leftarrow 0^{s} \; || \; GPR[rt]_{63..s}$
```

Exceptions:

None

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*.

Operation:

```
T:   $s \leftarrow 1 \parallel sa$
     $GPR[rd] \leftarrow 0^{s} \parallel GPR[rt]_{63..s}$
```

Exceptions:

None

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.

Operation:

```
T:    GPR[rd] ← GPR[rs] – GPR[rt]
```

Exceptions:

Integer overflow exception

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.

Operation:

```
T:    GPR[rd] ← GPR[rs] – GPR[rt]
```

Exceptions:

None

ERET Exception Return ERET

31 26	25	24 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	ERET 011000
6	1	19	6

Format:
ERET

Description:

ERET is the R4650 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:

```
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

31 26	25 0
J 000010	target
6	26

Format:
J target

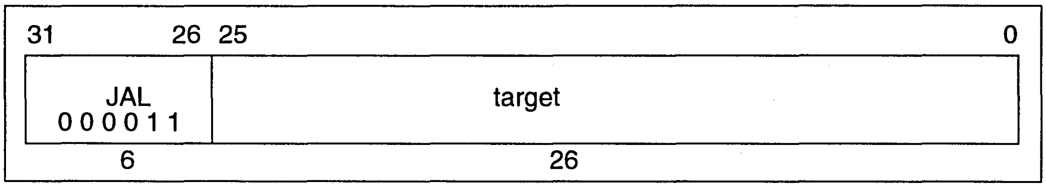
Description:
The 26-bit target address is shifted left two bits and combined with the high-order bits of the address of the delay slot. The program unconditionally jumps to this calculated address with a delay of one instruction.

Operation:

T: temp ← target
T+1: $PC \leftarrow PC_{63..28} \mathbin{||} temp \mathbin{||} 0^2$

Exceptions:
None

JAL Jump And Link JAL



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:

```
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 one 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 re-executed. 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:

```
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:

```
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:

```
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:

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:

```
T:  $vAddr \leftarrow ((offset_{15})^{48} \,||\, offset_{15..0}) + GPR[base]$
    $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
    $pAddr \leftarrow pAddr_{PSIZE-1..3} \,||\, (pAddr_{2..0}\ xor\ ReverseEndian^{3})$
    $mem \leftarrow LoadMemory\ (uncached, BYTE, pAddr, vAddr, DATA)$
    $byte \leftarrow vAddr_{2..0}\ xor\ BigEndianCPU^{3}$
    $GPR[rt] \leftarrow 0^{56} \,||\, mem_{7+8*\ byte..8*\ byte}$
```

Exceptions:

Bus error exception
Address error exception

LD Load Doubleword LD

31 26	25 21	20 16	15 0
LD 1 1 0 1 1 1	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.

Operation:

```
T:   vAddr ← $((offset_{15})^{48} \| offset_{15..0})$ + GPR[base]
     (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
     mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
     GPR[rt] ← mem
```

Exceptions:

Bus error exception
Address error exception

LDCz Load Doubleword To Coprocessor LDCz

31 26	25 21	20 16	15 0
LDCz 1 1 0 1 x x*	base	rt	offset
6	5	5	16

Note: *See "Opcode Bit Encoding" on this page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

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.

Operation:

```
T:  vAddr ← (($offset_{15}$)$^{48}$ || $offset_{15..0}$) + GPR[base]
    (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
    mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
    COPzLD (rt, mem)
```

Exceptions:

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	

Bits 31–28: Opcode; bits 27–26: 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:

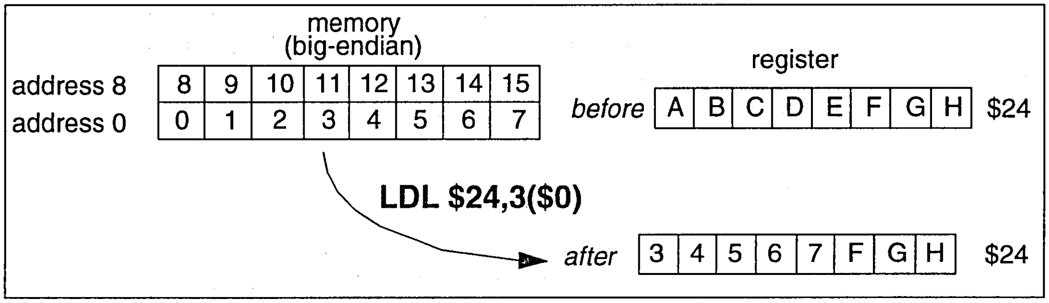
LDL rt, offset(base)

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 doubleword boundary. LDL loads the left portion of the register with the appropriate part of the high-order doubleword; LDR loads the right portion of the register with 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 loads 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.



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.

Operation:

```
T:  $vAddr \leftarrow ((offset_{15})^{48} || offset_{15..0}) + GPR[base]$
    $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
    $pAddr \leftarrow pAddr_{PSIZE-1..3} || (pAddr_{2..0} \text{ xor } ReverseEndian^3)$
    if BigEndianMem = 0 then
            $pAddr \leftarrow pAddr_{PSIZE-1..3} || 0^3$
    endif
    $byte \leftarrow vAddr_{2..0} \text{ xor } BigEndianCPU^3$
    $mem \leftarrow LoadMemory (uncached, byte, pAddr, vAddr, DATA)$
    $GPR[rt] \leftarrow mem_{7+8*byte..0} || GPR[rt]_{55-8*byte..0}$
```

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

Key to Table

LEM Little-endian memory (BigEndianMem = 0)
BEM BigEndianMem = 1
Type AccessType (see on page 2-3) sent to memory
Offset $pAddr_{2..0}$ sent to memory

Exceptions:

Bus error exception
Address error exception

LDR Load Doubleword Right LDR

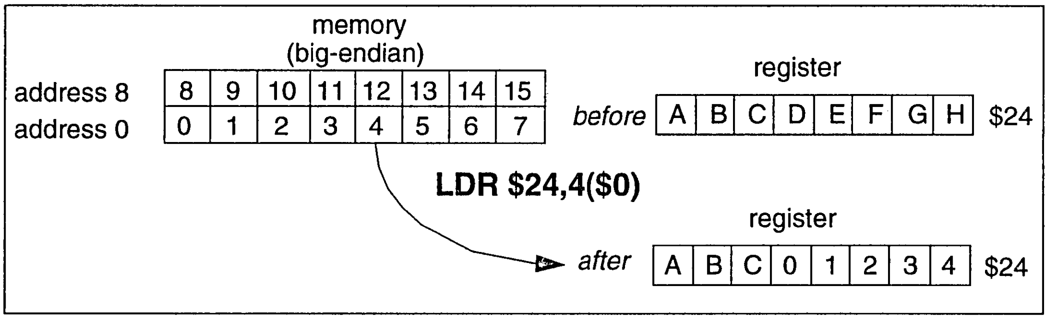
31 26	25 21	20 16	15 0
LDR 011011	base	rt	offset
6	5	5	16

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 doubleword boundary. LDR loads the right portion of the register with the appropriate part of the low-order doubleword; LDL loads the left portion of the register with 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 loads 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.



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.

Operation:

```
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 = 1 then
            $pAddr \leftarrow pAddr_{31..3} \,||\, 0^{3}$
    endif
    $byte \leftarrow vAddr_{2..0}\ xor\ BigEndianCPU^{3}$
    $mem \leftarrow LoadMemory\ (uncached, byte, pAddr, vAddr, DATA)$
    $GPR[rt] \leftarrow GPR[rt]_{63..64-8*byte} \,||\, mem_{63..8*byte}$
```

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

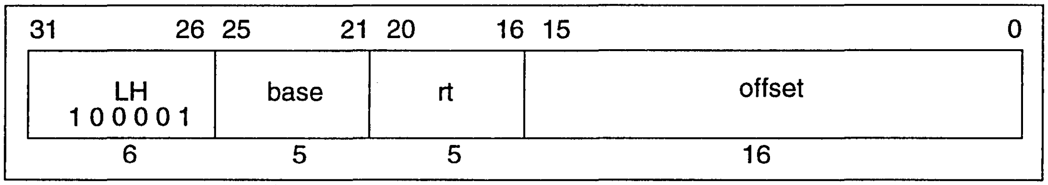
Key to Table

LEM Little-endian memory (BigEndianMem = 0)
BEM BigEndianMem = 1
Type AccessType (see on page 2-3) sent to memory
Offset $pAddr_{2..0}$ sent to memory

Exceptions:

Bus error exception
Address error exception

LH Load Halfword LH



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:

```
T:  $vAddr \leftarrow ((offset_{15})^{48} \,||\, offset_{15..0}) + GPR[base]$
    $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
    $pAddr \leftarrow pAddr_{PSIZE-1..3} \,||\, (pAddr_{2..0}\ xor\ (ReverseEndian \,||\, 0))$
    $mem \leftarrow LoadMemory\ (uncached, HALFWORD, pAddr, vAddr, DATA)$
    $byte \leftarrow vAddr_{2..0}\ xor\ (BigEndianCPU^{2} \,||\, 0)$
    $GPR[rt] \leftarrow (mem_{15+8*byte})^{16} \,||\, mem_{15+8*byte..8*byte}$
```

Exceptions:

Bus error exception
Address error exception

LHU Load Halfword Unsigned LHU

31 26	25 21	20 16	15 0
LHU 100101	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:

```
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:

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*. 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 as shown:

```
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.

This instruction is available in User mode, and it is not necessary for CP0 to be enabled.

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. A cache miss that occurs between LL and SC may cause SC to fail, so no load or store operation should occur between LL and SC, otherwise the SC may never be successful. Exceptions also cause SC to fail, so persistent exceptions must be avoided.

If either of the two least-significant bits of the effective address are non-zero, an address error exception takes place.

Operation:

```
T:  $vAddr \leftarrow ((offset_{15})^{48} \mathbin{||} offset_{15..0}) + GPR[base]$
    $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
    $pAddr \leftarrow pAddr_{PSIZE-1..3} \mathbin{||} (pAddr_{2..0}\ xor\ (ReverseEndian \mathbin{||} 0^2))$
    $mem \leftarrow LoadMemory\ (uncached, WORD, pAddr, vAddr, DATA)$
    $byte \leftarrow vAddr_{2..0}\ xor\ (BigEndianCPU \mathbin{||} 0^2)$
    $GPR[rt] \leftarrow (mem_{31+8*byte})^{32} \mathbin{||} mem_{31+8*byte..8*byte}$
    $LLbit \leftarrow 1$
    SyncOperation()
```

Exceptions:

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.

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. A cache miss that occurs between LLD and SCD may cause SCD to fail, so no load or store operation should occur between LLD and SCD, otherwise the SCD may never be successful. 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 non-zero, an address error exception takes place.

Operation:

```
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
     LLbit ← 1
     SyncOperation()
```

Exceptions:

Bus error exception
Address error exception

LUI Load Upper Immediate LUI

31 26	25 21	20 16	15 0
LUI 0 0 1 1 1 1	0 0 0 0 0 0	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*. The loaded word is sign-extended.

Operation:

T: $\mathrm{GPR}[rt] \leftarrow (\mathrm{immediate}_{15})^{32} \,\|\, \mathrm{immediate} \,\|\, 0^{16}$

Exceptions:
None

LW Load Word LW

31 26	25 21	20 16	15 0
LW 100011	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*. The loaded word is sign-extended.

If either of the two least-significant bits of the effective address is nonzero, an address error exception occurs.

Operation:

```
T:   $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
     $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
     $pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0} xor (ReverseEndian \| 0^2))$
     $mem \leftarrow LoadMemory (uncached, WORD, pAddr, vAddr, DATA)$
     $byte \leftarrow vAddr_{2..0} xor (BigEndianCPU \| 0^2)$
     $GPR[rt] \leftarrow (mem_{31+8*byte})^{32} \| mem_{31+8*byte..8*byte}$
```

Exceptions:

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

Note: *See "Opcode Bit Encoding" on this page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

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.

Operation:

```
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, DOUBLEWORD, pAddr, vAddr, DATA)
     byte ← $\text{vAddr}_{2..0}$ xor (BigEndianCPU $\| 0^2$)
     COPzLW (byte, rt, mem)
```

Exceptions:

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	
	Opcode				Coprocessor Unit Number		

LWL Load Word Left LWL

31 26	25 21	20 16	15 0
LWL 100010	base	rt	offset
6	5	5	16

Format:

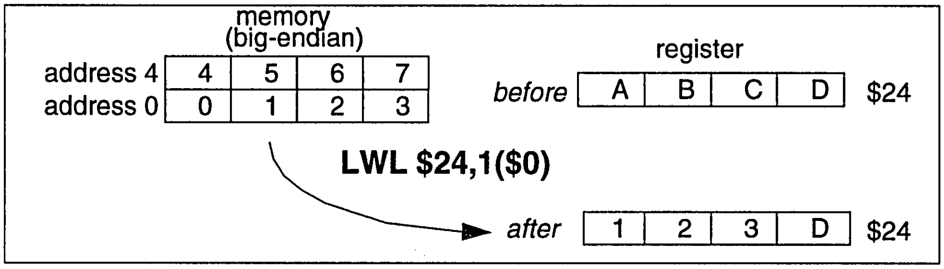
LWL rt, offset(base)

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 word boundary. LWL loads the left portion of the register with the appropriate part of the high-order word; LWR loads the right portion of the register with 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. 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 loads 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.



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:

```
T:  vAddr ← $((offset_{15})^{48}$ || $offset_{15..0})$ + GPR[base]
    (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
    pAddr ← $pAddr_{PSIZE-1..3}$ || $(pAddr_{2..0}$ xor $ReverseEndian^3)$
    if BigEndianMem = 0 then
            pAddr ← $pAddr_{PSIZE-1..3}$ || $0^3$
    endif
    byte ← $vAddr_{1..0}$ xor $BigEndianCPU^2$
    word ← $vAddr_2$ xor BigEndianCPU
    mem ← LoadMemory (uncached, 0 || byte, pAddr, vAddr, DATA)
    temp ← $mem_{31+32*word-8*byte..32*word}$ || $GPR[rt]_{23-8*byte..0}$
    GPR[rt] ← $(temp_{31})^{32}$ || temp
```

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

Key to Table

LEM Little-endian memory (BigEndianMem = 0)
BEM BigEndianMem = 1
Type AccessType (see on page 2-3) sent to memory
Offset $pAddr_{2..0}$ sent to memory
S sign-extend of $destination_{31}$

Exceptions:

Bus error exception
Address error exception

LWR Load Word Right LWR

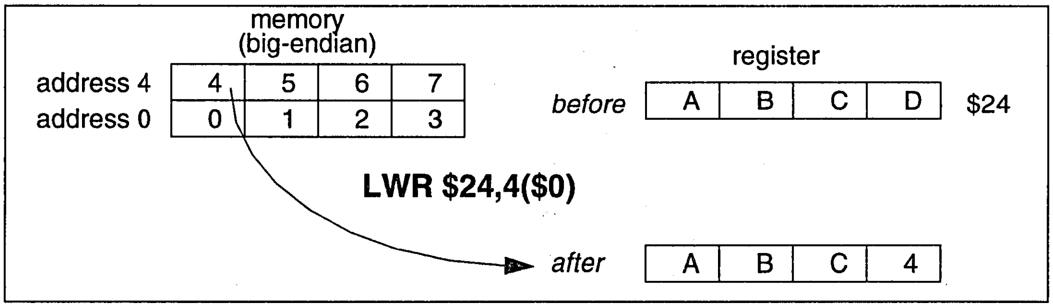
31 26	25 21	20 16	15 0
LWR 100110	base	rt	offset
6	5	5	16

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 word boundary. LWR loads the right portion of the register with the appropriate part of the low-order word; LWL loads the left portion of the register with 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. 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 loads 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:

```
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 $\text{ReverseEndian}^3$)
     if BigEndianMem = 1 then
               pAddr ← $\text{pAddr}_{\text{PSIZE}-31..3}$ || $0^3$
     endif
     byte ← $\text{vAddr}_{1..0}$ xor $\text{BigEndianCPU}^2$
     word ← $\text{vAddr}_2$ xor BigEndianCPU
     mem ← LoadMemory (uncached, 0 || byte, pAddr, vAddr, DATA)
     temp ← $\text{GPR[rt]}_{31..32-8*\text{byte}..0}$ || $\text{mem}_{31+32*\text{word}-32*\text{word}+8*\text{byte}}$
     GPR[rt] ← $(\text{temp}_{31})^{32}$ || temp
```

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

$\text{vAddr}_{2..0}$	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

Key to Table

LEM	Little-endian memory (BigEndianMem = 0)
BEM	BigEndianMem = 1
Type	AccessType (see on page 2-3) sent to memory
Offset	$\text{pAddr}_{2..0}$ sent to memory
S	sign-extend of destination_{31}

Exceptions:

Bus error exception
Address error exception

LWU Load Word Unsigned LWU

31 26	25 21	20 16	15 0
LWU 101111	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.

Operation:

```
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] ← $0^{32} \,||\, \text{mem}_{31+8*\text{byte}..8*\text{byte}}$
```

Exceptions:

Bus error exception
Address error exception

MAD Multiply/Add MAD

31 26	25 21	20 16	15 11	10 6	5 0
Special 2 011100	rs	rt	0	0	MAD 00000
6	5	5	5	5	6

Format:
MAD rs, rt

Description:
The R4650 adds a MAD instruction (multiply-accumulate, with HI and LO as the accumulator) to the base MIPS-III ISA. The MAD instruction is defined as:

$$HI,LO \leftarrow HI,LO + rs*rt$$

The lower 32-bits of the accumulator are stored in the lower 32 bits of LO, while the upper 32 bits of the result are stored in the lower 32 bits of HI. This is done to allow this instruction to operate compatibly in 32-bit processors.

The actual repeat rate and latency of this operation are dependent on the size of the operands, as explained in Appendix F, "Integer Multiply Scheduling."

Operation:

```
T:   $temp \leftarrow (HI_{31..0} \| LO_{31..0}) + ((rs_{31})^{32} \| rs_{31..0}) \times ((rt_{31})^{32} \| rt_{31..0})$
     $Hi \leftarrow (temp_{63})^{32} \| temp_{63..32}$
     $LO \leftarrow (temp_{31})^{32} \| temp_{31..0}$
```

Exceptions:
None

Note: This is an IDT proprietary extension.

MADU Multiply/Add Unsigned MADU

31 26	25 21	20 16	15 11	10 6	5 0
Special2 011100	rs	rt	0	0	MAD 00001
6	5	5	5	5	6

Format:
MADU rs, rt

Description:
The R4650 adds a MAD instruction (multiply-accumulate, with HI and LO as the accumulator) to the base MIPS-III ISA. The MAD instruction is defined as:

$$HI,LO \leftarrow HI,LO + rs*rt$$

The lower 32-bits of the accumulator are stored in the lower 32 bits of LO, while the upper 32 bits of the result are stored in the lower 32 bits of HI. This is done to allow this instruction to operate compatibly in 32-bit processors.

The actual repeat rate and latency of this operation are dependent on the size of the operands, as explained in Appendix F, "Integer Multiply Scheduling."

Operation:

```
T:  $temp \leftarrow (HI_{31..0} \,||\, LO_{31..0}) + (0^{32} \,||\, rs_{31..0}) \times (0^{32} \,||\, rt_{31..0})$
    $Hi \leftarrow (temp_{63})^{32} \,||\, temp_{63..32}$
    $LO \leftarrow (temp_{31})^{32} \,||\, temp_{31..0}$
```

Exceptions:
None

Note: This is an IDT proprietary extension.

MFC0 Move From System Control Coprocessor MFC0

31 26	25 21	20 16	15 11	10 0
COP0 010000	MF 00000	rt	rd	0 000 0000 0000
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*. May be used on both 32-bit and 64-bit CP0 registers.

Operation:

```
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	0 000 0000 0000
6	5	5	5	11

Note: *See "Opcode Bit Encoding" on this page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

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:

```
T:      if $rd_0 = 0$ then
                data ← $CPR[z,rd_{4..1} \| 0]_{31..0}$
        else
                data ← $CPR[z,rd_{4..1} \| 0]_{63..32}$
        endif
T+1:    $GPR[rt] \leftarrow (data_{31})^{32} \| data$
```

Exceptions:

Coprocessor unusable exception
Reserved instruction exception (coprocessor 3)

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	
Bit #	31	30	29	28	27	26	25	24	23	22	21	0
MFC1	0	1	0	0	0	1	0	0	0	0	0	
Bit #	31	30	29	28	27	26	25	24	23	22	21	0
MFC2	0	1	0	0	1	0	0	0	0	0	0	

Opcode (bits 31–28) Coprocessor Unit Number (bits 27–26) Coprocessor Suboperation (bits 25–21)

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:

```
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:

```
T:        GPR[rd] ← LO
```

Exceptions:

None

MTC0 — Move To System Control Coprocessor — MTC0

31 26	25 21	20 16	15 11	10 0
COP0 0 1 0 0 0 0	MT 0 0 1 0 0	rt	rd	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format:

MTC0 rt, rd

Description:

The contents of general register *rt* are loaded into coprocessor register *rd* of CP0.

Because the state of the virtual address translation system may be altered by this instruction, the operation of load instructions and store instructions immediately prior to and after this instruction are undefined.

Operation:

```
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

Note: *See "Opcode Bit Encoding" on this page, or "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

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:

```
T:    $data \leftarrow GPR[rt]_{31..0}$
T+1:  if $rd_0 = 0$
          $CPR[z,rd_{4..1} \| 0] \leftarrow CPR[z, rd_{4..1} \| 0]_{63..32} \| data$
      else
          $CPR[z,rd_{4..1} \| 0] \leftarrow data \| CPR[z,rd_{4..1} \| 0]_{31..0}$
      endif
```

Exceptions:

Coprocessor unusable exception
Reserved instruction exception (coprocessor 3)

Opcode Bit Encoding:

MTCz

	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	

Opcode Coprocessor Unit Number Coprocessor Suboperation

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:

```
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:

```
T–2:  LO ← undefined
T–1:  LO ← undefined
T:    LO ← GPR[rs]
```

Exceptions:

None

MUL Multiply MUL

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL2 011100	rs	rt	rd	0	**MUL** 00010
6	5	5	5	5	6

Format:

MUL rd, rs, rt

Description:

The R4650 adds a true 3-operand 32x32—>32 multiply instruction to the MIPS-III ISA, where by rd = rs*rt. This instruction eliminates the need to explicitly move the multiply result from the LO register back to a general register.

The execution time of this operation is operand size dependent, as explained in Appendix F, "Integer Multiply Scheduling."

The HI and LO registers are undefined after executing this instruction. For 16-bit operands, the latency of MUL is 3 cycles, with a repeat rate of 2 cycles. In addition, the MUL instruction will unconditionally slip or stall for all but 2 cycles of its latency.

Operation:

```
T:    $\text{Temp} \leftarrow \text{rs}_{31..0} \text{ x } \text{rt}_{31..0}$
      $\text{rd} \leftarrow (\text{temp}_{31})^{32} \,||\, \text{temp}_{31...0}$
      $\text{HI} \leftarrow \text{undefined}$
      $\text{LO} \leftarrow \text{undefined}$
```

Exceptions:

None

Note: This instruction is an IDT proprietary extension.

MULT Multiply MULT

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt		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. 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

Operation:

```
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. 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.

Operation:

```
T–2:  LO    ← undefined
      HI    ← undefined
T–1:  LO    ← undefined
      HI    ← undefined
T:    t     ← $(0 \| GPR[rs]_{31..0}) * (0 \| GPR[rt]_{31..0})$
      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:

```
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:

```
T:        GPR[rd] ← GPR[rs] or GPR[rt]
```

Exceptions:
None

ORI Or Immediate ORI

31 26	25 21	20 16	15 0
ORI 001101	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:

$$\text{T:} \quad \text{GPR[rt]} \leftarrow \text{GPR[rs]}_{63..16} \,||\, (\text{immediate or GPR[rs]}_{15..0})$$

Exceptions:

None

SB Store Byte SB

31 26	25 21	20 16	15 0
SB 101000	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:

```
T:   vAddr ← $((\text{offset}_{15})^{48} \mid\mid \text{offset}_{15..0}) + \text{GPR[base]}$
     (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
     pAddr ← $\text{pAddr}_{\text{PSIZE-1..3}} \mid\mid (\text{pAddr}_{2..0}$ xor $\text{ReverseEndian}^{3})$
     byte ← $\text{vAddr}_{2..0}$ xor $\text{BigEndianCPU}^{3}$
     data ← $\text{GPR[rt]}_{63-8*\text{byte}..0} \mid\mid 0^{8*\text{byte}}$
     StoreMemory (uncached, BYTE, data, pAddr, vAddr, DATA)
```

Exceptions:

Bus error exception
Address error exception

SC Store Conditional SC

31 26	25 21	20 16	15 0
SC 1 1 1 0 0 0	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 non-zero, an address error exception takes place.

If this instruction should both fail and take an exception, the exception takes precedence.

Operation:

```
T:    $\mathrm{vAddr} \leftarrow ((\mathrm{offset}_{15})^{48} \,\|\, \mathrm{offset}_{15..0}) + \mathrm{GPR[base]}$
      $(\mathrm{pAddr, uncached}) \leftarrow \mathrm{AddressTranslation\ (vAddr, DATA)}$
      $\mathrm{pAddr} \leftarrow \mathrm{pAddr}_{\mathrm{PSIZE-1..3}} \,\|\, (\mathrm{pAddr}_{2..0}\ \mathrm{xor}\ (\mathrm{ReverseEndian} \,\|\, 0^{2}))$
      $\mathrm{data} \leftarrow \mathrm{GPR[rt]}_{63-8*\mathrm{byte}..0} \,\|\, 0^{8*\mathrm{byte}}$
      if LLbit then
          StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
      endif
      $\mathrm{GPR[rt]} \leftarrow 0^{63} \,\|\, \mathrm{LLbit}$
      SyncOperation()
```

Exceptions:
Bus error exception
Address error exception

SCD Store Conditional Doubleword SCD

31 26	25 21	20 16	15 0
SCD 111100	base	rt	offset
6	5	5	16

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.

If this instruction should both fail and take an exception, the exception takes precedence.

Operation:

```
T:  vAddr ← $((offset_{15})^{48} \,||\, offset_{15..0}) + GPR[base]$
    (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
    data ← GPR[rt]
    if LLbit then
        StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
    endif
    GPR[rt] ← $0^{63} \,||\,$ LLbit
    SyncOperation()
```

Exceptions:

Bus error exception
Address error exception

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.

Operation:

```
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:
Bus error exception
Address error exception

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.

Operation:

```
T:   vAddr ← ((offset_15)^48 || offset_15..0) + GPR[base]
     (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
     data ← COPzSD(rt),
     StoreMemory (uncached, DOUBLEWORD, data, pAddr,
     vAddr, DATA)
```

Note: *See the table in this section under "Opcode Bit Encoding." Also see "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

Exceptions:

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	
	SD opcode				Coprocessor Unit Number		

SDL Store Doubleword Left SDL

31 26	25 21	20 16	15 0
SDL 101100	base	rt	offset
6	5	5	16

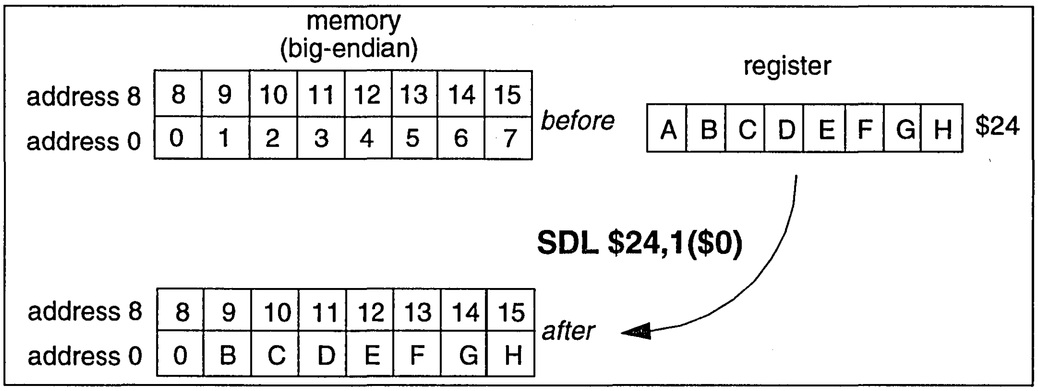
Format:
SDL rt, offset(base)

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 doubleword boundary. 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 copies 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.



Operation:

```
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 ← $\text{pAddr}_{31..3} \,\|\, 0^3$
    endif
    byte ← $\text{vAddr}_{2..0}$ xor $\text{BigEndianCPU}^3$
    data ← $0^{56-8*\text{byte}} \,\|\, \text{GPR[rt]}_{63..56-8*\text{byte}}$
    Storememory (uncached, byte, data, pAddr, vAddr, DATA)
```

Given a doubleword in a register and a doubleword in memory, the operation of SDL is as follows:

SDL

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	Little-endian memory (BigEndianMem = 0)
BEM	BigEndianMem = 1
Type	AccessType (see on page 2-3) sent to memory
Offset	pAddr$_{2..0}$ sent to memory

Exceptions:

Bus error exception
Address error exception

SDR Store Doubleword Right SDR

31 26	25 21	20 16	15 0
SDR 101101	base	rt	offset
6	5	5	16

Format:

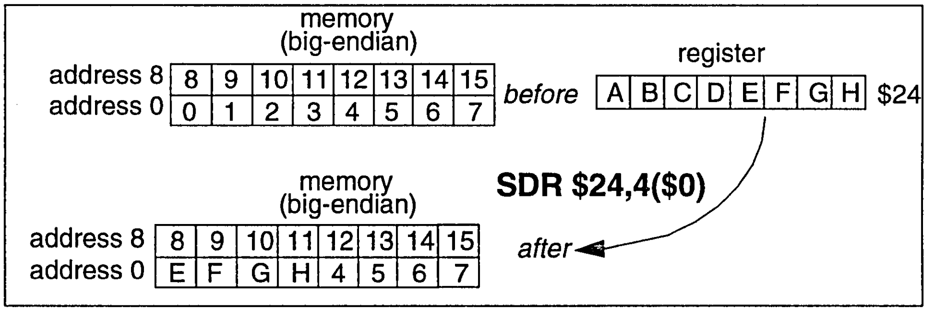
SDR rt, offset(base)

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 copies 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.



Operation:

```
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$^3$)
     If BigEndianMem = 0 then
              pAddr ← pAddr$_{PSIZE-31..3}$ || 0$^3$
     endif
     byte ← vAddr$_{1..0}$ xor BigEndianCPU$^3$
     data ← GPR[rt]$_{63-8*byte}$ || 0$^{8*byte}$
```

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

	BigEndianCPU = 0				BigEndianCPU = 1			
			offset				offset	
$vAddr_{2..0}$	destination	type	LEM	BEM	destination	type	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	Little-endian memory (BigEndianMem = 0)
BEM	BigEndianMem = 1
Type	AccessType (see on page 2-3) sent to memory
Offset	$pAddr_{2..0}$ sent to memory

Exceptions:

Bus error exception
Address error exception

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:

```
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:

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	0 00000	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.*

The operand must be a valid sign-extended, 32-bit value.

Operation:

```
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	rs	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 the number of bits specified by the low-order five bits contained in general register *rs*, inserting zeros into the low-order bits.

The result is placed in register *rd*.

The operand must be a valid sign-extended, 32-bit value.

Operation:

```
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:

```
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 001010	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:

```
T:    if $GPR[rs] < (immediate_{15})^{48} || immediate_{15..0}$ then
         $GPR[rd] \leftarrow 0^{63} || 1$
      else
         $GPR[rd] \leftarrow 0^{64}$
      endif
```

Exceptions:

None

SLTIU Set On Less Than Immediate Unsigned SLTIU

31 26	25 21	20 16	15 0
SLTIU 0 0 1 0 1 1	rs	rt	immediate
6	5	5	16

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:

```
T:   if $(0 \mid\mid \text{GPR[rs]}) < 0 \mid\mid (\text{immediate}_{15})^{48} \mid\mid \text{immediate}_{15..0}$ then
       GPR[rd] $\leftarrow 0^{63} \mid\mid 1$
     else
       GPR[rd] $\leftarrow 0^{64}$
     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:

```
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*.

The operand must be a valid sign-extended, 32-bit value.

Operation:

```
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*.

The operand must be a valid sign-extended, 32-bit value.

Operation:

```
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*.

The operand must be a valid sign-extended, 32-bit value.

Operation:

```
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*.

The operand must be a valid sign-extended, 32-bit value.

Operation:

```
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*. 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:

```
T:   $\text{temp} \leftarrow \text{GPR}[rs] - \text{GPR}[rt]$
     $\text{GPR}[rd] \leftarrow (\text{temp}_{31})^{32} \,||\, \text{temp}_{31..0}$
```

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.*

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:

```
T:    temp ← GPR[rs] - GPR[rt]
      GPR[rd] ← $(temp_{31})^{32}$ || $temp_{31..0}$
```

Exceptions:

None

SW Store Word SW

31 26	25 21	20 16	15 0
SW 101011	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 nonzero, an address error exception occurs.

Operation:

```
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 (uncached, WORD, data, pAddr, vAddr, DATA)
```

Exceptions:

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:

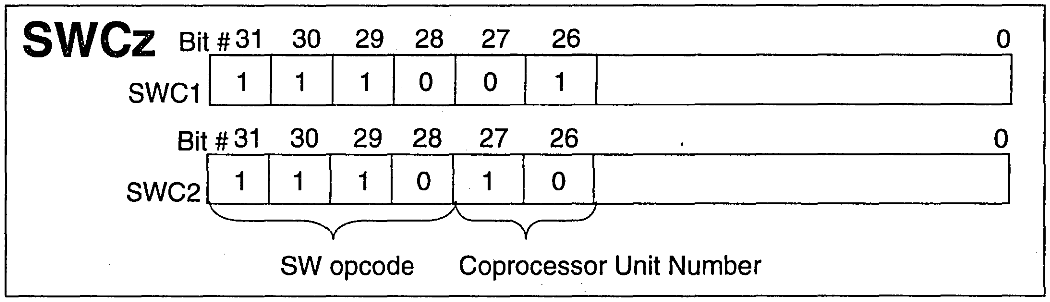
```
T:  vAddr ← $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15..0}) + \text{GPR[base]}$
    (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
    pAddr ← $\text{pAddr}_{\text{PSIZE-1..3}} \,||\, (\text{pAddr}_{2..0}$ xor $(\text{ReverseEndian} \,||\, 0^2)$
    byte ← $\text{vAddr}_{2..0}$ xor $(\text{BigEndianCPU} \,||\, 0^2)$
    data ← COPzSW (byte,rt)
    StoreMemory (uncached, WORD, data, pAddr, vAddr DATA)
```

Note: *See the table in this section under "Opcode Bit Encoding." Also see "CPU Instruction Opcode Bit Encoding" at the end of Appendix A.

Exceptions:

Bus error exception
Address error exception
Coprocessor unusable exception
Reserved instruction exception (coprocessor 3)

Opcode Bit Encoding:



SWCz

Bit #	31	30	29	28	27	26	0
SWC1	1	1	1	0	0	1	

Bit #	31	30	29	28	27	26	0
SWC2	1	1	1	0	1	0	

SWL Store Word Left SWL

31 26	25 21	20 16	15 0
SWL 101010	base	rt	offset
6	5	5	16

Format:
SWL rt, offset(base)

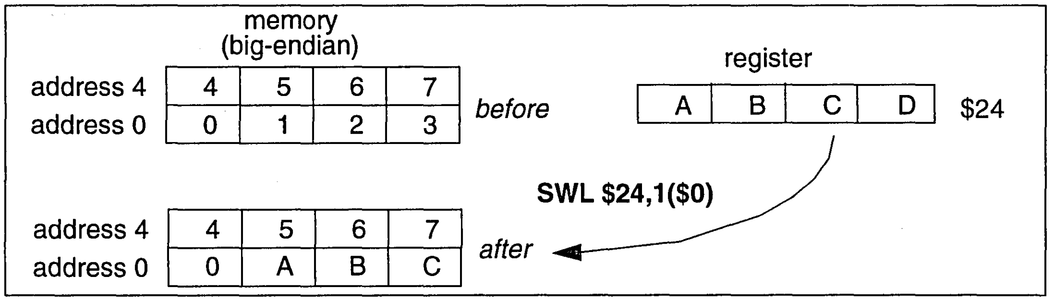
Description:

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 word boundary. 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 copies 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.



Operation:

```
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 ← $\text{pAddr}_{31..2} \,||\, 0^2$
    endif
    byte ← $\text{vAddr}_{1..0}$ xor $\text{BigEndianCPU}^2$
    if ($\text{vAddr}_2$ xor BigEndianCPU) = 0 then
        data ← $0^{32} \,||\, 0^{24-8*\text{byte}} \,||\, \text{GPR[rt]}_{31..24-8*\text{byte}}$
    else
        data ← $0^{24-8*\text{byte}} \,||\, \text{GPR[rt]}_{31..24-8*\text{byte}} \,||\, 0^{32}$
    endif
    StoreMemory(uncached, byte, data, pAddr, vAddr, DATA)
```

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	Little-endian memory (BigEndianMem = 0)
BEM	BigEndianMem = 1
Type	AccessType (see on page 2-3) sent to memory
Offset	$pAddr_{2..0}$ sent to memory

Exceptions:

Bus error exception
Address error exception

SWR Store Word Right SWR

31 26	25 21	20 16	15 0
SWR 101110	base	rt	offset
6	5	5	16

Format:
SWR rt, offset(base)

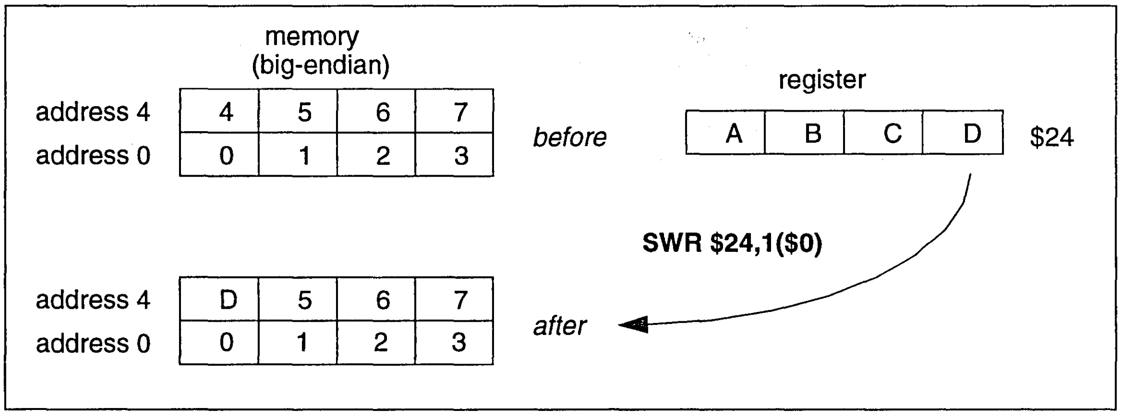
Description:

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 copies 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.



Operation:

```
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

$vAddr_{2..0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	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	Little-endian memory (BigEndianMem = 0)
BEM	BigEndianMem = 1
Type	AccessType (see on page 2-3) sent to memory
Offset	$pAddr_{2..0}$ sent to memory

Exceptions:

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 a 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:

```
T:    SyncOperation()
```

Exceptions:
None

SYSCALL System Call SYSCALL

31 26	25 6	5 0
SPECIAL 000000	Code	SYSCALL 001100
6	20	6

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:

```
T:    SystemCallException
```

Exceptions:
System Call exception

TEQ Trap If Equal TEQ

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	code	TEQ 110100
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:

```
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:

```
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:

```
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:

```
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:

```
T:  if $(0 \| \mathrm{GPR}[rs]) \geq (0 \| (\mathrm{immediate}_{15})^{48} \| \mathrm{immediate}_{15..0})$ 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 010000	CO 1	0 000 0000 0000 0000 0000	TLBP 001000
6	1	19	6

This instruction is not supported in R4650. Not guaranteed to trap.

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

This instruction is not supported in R4650. Not guaranteed to trap.

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

This instruction is not supported in R4650. Not guaranteed to trap.

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

This instruction is not supported in R4650. Not guaranteed to trap.

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:

```
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:

```
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:

```
T:   if $(0 \| \mathrm{GPR}[rs]) < (0 \| (\mathrm{immediate}_{15})^{48} \| \mathrm{immediate}_{15..0})$ 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:

```
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:

```
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:

```
T:  if GPR[rs] $\neq$ (immediate$_{15}$)$^{48}$ || immediate$_{15..0}$ then
        TrapException
    endif
```

Exceptions:
Trap exception

WAIT Wait WAIT

31 26	25	24 6	5 0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	WAIT 100000
6	1	19	6

Format:
WAIT

Description:
The WAIT instruction is used to halt the internal pipeline and thus reduce the power consumption of the CPU. See Appendix G for more details.

Operation:

```
T:    if SysAD bus is idle then
                StopPipeline
      endif
```

Exceptions:
Coprocessor unusable 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:

```
T:   GPR[rd] ← GPR[rs] xor GPR[rt]
```

Exceptions:

None

XORI Exclusive OR Immediate XORI

31 26	25 21	20 16	15 0
XORI 001110	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:

T: GPR[rt] ← GPR[rs] 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 R4600/R4700.

Table A.4. lists the R4600/R4700 Opcode Bit Encoding.

Opcode

28..26

31..29	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	DADDI	DADDIU	LDL	LDR	Special2	*	*	*
4	LB	LH	LWL	LW	LBU	LHU	LWR	LWU
5	SB	SH	SWL	SW	SDL	SDR	SWR	CACHE δ
6	LL	LWC1	LWC2	*	LLD	LDC1	LDC2	LD
7	SC	SWC1	SWC2	*	SCD	SDC1	SDC2	SD

SPECIAL function

2..0

5..3	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	DSLLV	*	DSRLV	DSRAV
3	MULT	MULTU	DIV	DIVU	DMULT	DMULTU	DDIV	DDIVU
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	*	*	SLT	SLTU	DADD	DADDU	DSUB	DSUBU
6	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	DSLL	*	DSRL	DSRA	DSLL32	*	DSRL32	DSRA32

SPECIAL function2

2..0

5..3	0	1	2	3	4	5	6	7
0	MAD	MADU	MUL	*	*	*	*	*
1	*	*	*	*	*	*	*	*
2	*	*	*	*	*	*	*	*
3	*	*	*	*	*	*	*	*
4	*	*	*	*	*	*	*	*
5	*	*	*	*	*	*	*	*
6	*	*	*	*	*	*	*	*
7	*	*	*	*	*	*	*	*

Key to Table

- * 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 R4600 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 R4600 implementations.

Table A.4. R4600/R4700 Opcode Bit Encoding (Page 1 of 2)

COPz rs

	23..21							
25,24	0	1	2	3	4	5	6	7
0	MF	DMF	CF	γ	MT	DMT	CT	γ
1	BC	γ	γ	γ	γ	γ	γ	γ
2	CO							
3								

COPz rt

	18..16							
20..19	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

	2..0							
5..3	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	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
3	ERET	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
4	WAIT	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
5	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
6	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
7	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ

REGIMM rt

	18..16							
20..19	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	*	*	*	*	*	*	*	*

Table A.4 R4600/R4700 Opcode Bit Encoding (Page 2 of 2)

Integrated Device Technology, Inc.

FPU Instruction Set Details

Appendix B

Introduction

This appendix provides a detailed description of each floating-point unit (FPU) instruction. Refer to Appendix A for details of the CPU instructions.

The instructions are listed alphabetically. Following each description is a discussion of exceptions that may result from executing the instruction. Refer to Chapter 7, "Floating Point Exceptions," for specifics about exception handling and their immediate causes.

Figure B.3 on page B-46 lists the entire bit encoding for the constant fields of the floating-point instruction set. For bit encoding for an individual instruction, refer to that instruction's description.

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
- R-Type, or Register instructions, which include the two- and three-register floating-point operations.

The instruction description subsections that follow show how these three basic instruction formats are used by:

- Load and store instructions
- Move instructions
- Floating-Point computational instructions
- 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.

Table B.1 shows the valid FPU instruction formats. Each operation is valid for certain formats only. Implementations *may* support some of these formats and operations through emulation, but they only need to support combinations that are valid.

Valid combinations are marked with a *V*. The combinations marked with an *R* are not currently specified for the R4650, and they will cause an unimplemented instruction trap. They will be available for future extensions to the architecture.

Operation	Source Format			
	Single	**Double**	**Word**	**Longword**
ADD	V	R	R	R
SUB	V	R	R	R
MUL	V	R	R	R
DIV	V	R	R	R
SQRT	V	R	R	R
ABS	V	R	R	R
MOV	V	R		
NEG	V	R	R	R
TRUNC.L	V	R		
ROUND.L	V	R		
CEIL.L	V	R		
FLOOR.L	V	R		
TRUNC.W	V	R		
ROUND.W	V	R		
CEIL.W	V	R		
FLOOR.W	V	R		
CVT.S		R	V	V
CVT.D	R	R	R	R
CVT.W	V	R		
CVT.L	V	R		
C	V	R	R	R

Key to Table:
V Valid combination.
R Not currently specified for the R4650; causes an unimplemented instruction trap.

Table B.1 Valid FPU Instruction Formats

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.

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

Table B.2 Logical Negation of Predicates by Condition True/False

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 *General Purpose* registers. These operations are unformatted; no format conversions are performed and, therefore, no floating-point exceptions can 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. Note, however, that doubleword moves do cause an unimplemented exception.

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 operation set includes:

- floating-point add
- floating-point subtract
- floating-point multiply
- floating-point divide
- floating-point square root
- convert between fixed-point and floating-point formats
- convert between floating-point formats
- floating-point compare

These operations satisfy the requirements of IEEE Standard 754 requirements for accuracy. Specifically, these operations obtain a result which is identical to an infinite-precision result rounded 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 Notation Conventions

In this appendix, all variable subfields in an instruction format (such as *fs, ft, immediate*, and so on) are shown in lower-case. The instruction name (such as ADD, SUB, and so on) is shown in upper-case.

For clarity, an alias is sometimes used for a variable subfield in the formats of specific instructions. For example, *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, the instruction subfields *op* and *function* can have constant 6-bit values. When reference is made to these instructions, upper-case mnemonics are used. For instance, in the floating-point ADD instruction we use *op* = COP1 and *function* = FADD. In other cases, a single field has both fixed and variable subfields, so the name contains both upper and lower case characters. Bit encoding for mnemonics are shown in Figure B.3 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

The following examples illustrate the application of some of the instruction notation conventions:

Example #1:

$$\text{GPR[rt]} \leftarrow \text{immediate} \,||\, 0^{16}$$

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 General Purpose Register rt.

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 R4650 implementation, the instruction immediately following a load may use the contents of the register being 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.

The behavior of the load store instructions is dependent on the width of the *FGR*s.

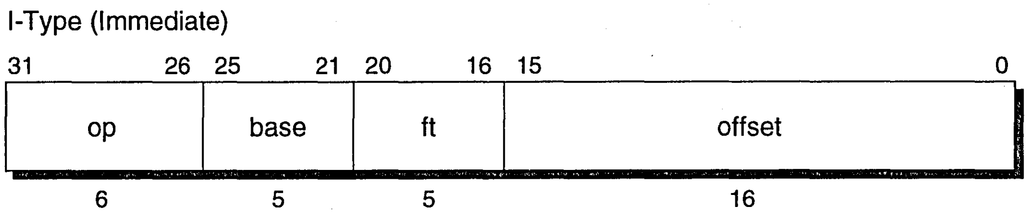
- When the *FR* bit in the *Status* register equals zero, there are 16 *Floating-Point General* registers (*FGR*s), each 32-bits wide.
- When the *FR* bit in the *Status* register equals one, there are 32 32-bit *Floating-Point General* registers (*FGR*s).

In the load and store operation descriptions, the functions listed in Table B.3 are used to summarize the handling of virtual addresses and physical memory.

Function	**Meaning**
AddressTranslation	Uses the CP0 to find the physical address given the virtual address. The function fails and an exception is taken if the required translation is not present/allowed.
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.

Table B.3 Load and Store Common Functions

Figure B.1 shows the I-Type instruction format used by load and store operations.



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.1 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.

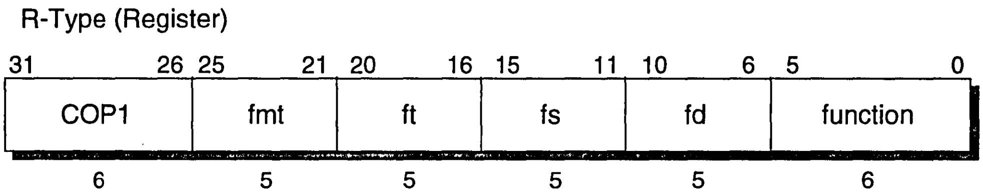
For doubleword loads and stores, the access type field is always DOUBLEWORD, and the low-order three bits of the address must always be zero.[1]

Regardless of byte-numbering order (Endianness), the address specifies that byte which has the smallest byte-address 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.2 shows the R-Type instruction format used for computational operations.



COP1	is a 6-bit 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.2 Computational Instruction Format

The *function* field indicates the floating-point operation to be performed.

Each floating-point instruction can be applied to a number of operand *formats*. The operand format for an instruction is specified by the 5-bit *format* field; decoding for this field is shown in Table B.4.

[1] Causes an unimplemented trap.

Code	Mnemonic	Size	Format
16	S	single	Binary floating-point
17	D[†]	double	Binary floating-point
18	Reserved		
19	Reserved		
20	W	single	32-bit binary fixed-point
21	L	longword	64-bit binary fixed-point
22–31	Reserved		
Note:	[†]Causes an unimplemented trap.		

Table B.4 Format Field Decoding

Table B.5 lists all floating-point instructions.

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 $+\infty$
11	FLOOR.L[†]	Convert to single fixed-point, rounded to $-\infty$
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 $+\infty$
15	FLOOR.W	Convert to single fixed-point, rounded to $-\infty$
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 32-bit binary fixed-point
37	CVT.L[†]	Convert to 64-bit binary fixed-point
38–47	–	Reserved
48–63	C	Floating-point compare
Note: [†]Causes an unimplemented trap.		

Table B.5 Floating-Point Instructions and Operations

In the following pages, the notation *FGR* refers to the 32 *General Purpose* registers *FGR0* through *FGR31* of the FPU, and *FPR* refers to the floating-point registers of the FPU.

- When the *FR* bit in the *Status* register (SR(26)) equals zero, only the even floating-point registers are valid and the 32 *General Purpose* registers of the FPU 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 32 *General Purpose* registers of the FPU are 32-bits wide.

The following routines are used in the description of the floating-point operations to retrieve the value of an FPR or to change the value of an FGR:

```
FR = 0

value ← ValueFPR(fpr, fmt)
case fmt of
    S, W:
        if $FGR_0 = 0$
            value ← FGR[fpr]
        else
            value ← FGR[fpr - 1]
        endif
    D:
        /* undefined for fpr not even */
        value ← FGR[fpr]
end

StoreFPR(fpr, fmt, value):
case fmt of
    S, W:
        if $FGR_0 = 0$
            FGR[fpr] ← $FGR[fpr]_{63..32}$ || value
        else
            FGR[fpr - 1] ← value || $FGR[fpr - 1]_{31..0}$
        endif
    D:
        /* undefined for fpr not even */
        FGR[fpr] ← value
end
```

```
FR = 1

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
Unimplemented (.fmt = .D)

Coprocessor Exceptions:

Unimplemented operation exception (*e.g.* .D)
Invalid operation exception

ADD.fmt Floating-Point Add ADD.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	ft	fs	fd	ADD 000000
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 (*e.g.* .D)
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 010001	BC 01000	BCF 00000	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:

```
T–1:   condition ← not COC[1]
T:     target ← $(\text{offset}_{15})^{46}$ || 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 010001	BC 01000	BCFL 00010	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:

```
T–1:   condition ← not COC[1]
T:     target ← $(\text{offset}_{15})^{46}$ || 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 010001	BC 01000	BCT 00001	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:

```
T–1:   condition ← COC[1]
T:     target ← $(\text{offset}_{15})^{46}$ || 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:

```
T–1:    condition ← COC[1]
T:      target ← $(\text{offset}_{15})^{46}$ || 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 as 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.

Note: *See "FPU Instruction Opcode Bit Encoding" at the end of Appendix B.

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 (*e.g.* .D)
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- or double-precision floating-point formats. 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 traps on the R4650, which does not support the.L format.

Operation:

```
T:    StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Exceptions:

Coprocessor unusable exception
Floating-Point exception

Coprocessor Exceptions:

Invalid operation exception
Unimplemented operation exception (*e.g.* .D)
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 010001	fmt	0 00000	fs	fd	CEIL.W 001110
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.

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 (*e.g.* .D)
Inexact exception
Overflow exception

CFC1 Move Control Word From FPU (Coprocessor 1) CFC1

31 26	25 21	20 16	15 11	10 0
COP1 010001	CF 00010	rt	fs	0 000 0000 0000
6	5	5	5	11

Format:

CFC1 rt, fs

Description:

The contents of the FPU 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:

```
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 010001	CT 00110	rt	fs	0 000 0000 0000
6	5	5	5	11

Format:
CTC1 rt, fs

Description:
The contents of general register *rt* are loaded into FPU control register *fs*. This operation is only defined when *fs* equals 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:

T: temp ← $\mathrm{GPR[rt]}_{31..0}$

T+1: FCR[fs] ← temp

COC[1] ← $\mathrm{FCR[31]}_{23}$

Exceptions:
Coprocessor unusable exception
Floating-Point exception

Coprocessor Exceptions:
Unimplemented operation exception (*e.g.* .D)
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.

This instruction traps on the R4650, which does not support the.D format.

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* 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*.

This instruction is valid only for conversions from single- or double-precision floating-point formats.

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 traps on the R4650, which does not support the .L format.

Operation:

```
T:    StoreFPR (fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Exceptions:

Coprocessor unusable exception
Floating-Point exception

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 010001	fmt	0 00000	fs	fd	CVT.S 100000
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 (*e.g.* .D)
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 (*e.g.* .D)
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, ValueFPR(fs, fmt)/ValueFPR(ft, fmt))
```

Exceptions:

Coprocessor unusable exception
Floating-Point exception

Coprocessor Exceptions:

Unimplemented operation exception (*e.g.* .D)
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 010001	DMF 00001	rt	fs	0 000 0000 0000
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 registers of the R4650 are addressable. When *FR* equals zero, 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.

DMFC1 will always trap on the R4650.

Operation:

```
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.
Unimplemented operation exception.

DMTC1 Doubleword Move To Floating-Point Coprocessor DMTC1

31 26	25 21	20 16	15 11	10 0
COP1 010001	DMT 00101	rt	fs	0 000 0000 0000
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 registers of the R4650 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.

DMTC1 will always trap on the R4650.

Operation:

```
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.

Unimplemented operation 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 -∞ (3).

This instruction is valid only for conversion from single- or double-precision floating-point formats.

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 traps on the R4650, which does not support the .L format.

Operation:

```
T:    StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Exceptions:

Coprocessor unusable exception
Floating-Point exception

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 –∞ (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.

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 (*e.g.* .D)
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:
LDC1 will always trap.

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 non-zero, an address error exception occurs.

Operation:

```
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 || $0^2$)
      if $\text{SR}_{26}$ = 1 then
      CPR[1, ft] ← $\text{undefined}^{32}$ || $\text{mem}_{31+8*\text{byte}..8*\text{byte}}$
      else if $\text{ft}_0$=0 then
      CPR[1, $\text{ft}_{4..1}$ || 0] ← CPR[1, $\text{ft}_{4..1}$ || $0]_{64..32}$ || $\text{mem}_{31+8*\text{byte}..8*\text{byte}}$
      else
      CPR[1, $\text{ft}_{4..1}$ || 0] ← $\text{mem}_{31+8*\text{byte}..8*\text{byte}}$ || CPR[1, $\text{ft}_{4..1}$ || $0]_{31..0}$
      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 010001	MF 00000	rt	fs	0 000 0000 0000
6	5	5	5	11

Format:

MFC1 rt, fs

Description:

The contents of register *fs* from the floating-point coprocessor are loaded 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 R4650 are addressable. If *FR* equals zero, MFC1 loads 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:

```
T:      if $SR_{26}$ = 1 then
                data ← CPR[1, fs]
        else if $fs_0$ = 0 then
                data ← $CPR[1, fs_{4..1} \| 0]_{31..0}$
        else
                data ← $CPR[1, fs_{4..1} \| 0]_{63..32}$
        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 (*e.g.* .D)

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 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 R4650 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:

```
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 010001	fmt	ft	fs	fd	MUL 000010
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 (*e.g.* .D)
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 (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, Negate(ValueFPR(fs, fmt)))
```

Exceptions:

Coprocessor unusable exception
Floating-Point exception

Coprocessor Exceptions:

Unimplemented operation exception (*e.g.* .D)
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- or double-precision floating-point formats.

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 traps on the R4650, which does not support the .L format.

Operation:

```
T:    StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Exceptions:
Coprocessor unusable exception
Floating-Point exception

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 the 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.

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 (*e.g.* .D)
Inexact exception
Overflow exception

SDC1 Store Doubleword from FPU (Coprocessor 1) SDC1

31 26	25 21	20 16	15 0
SDC1 111101	base	ft	offset
6	5	5	16

Format:
SDC1 ft, offset(base)

Description:
SDC1 will always trap on the R4650.
Coprocessor exceptions
Unimplemented operation exception

SQRT.fmt Floating-Point Square Root SQRT.fmt

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	SQRT 000100
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 (*e.g.* .D)
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 010001	fmt	ft	fs	fd	SUB 000001
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 (*e.g.* .D)
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 = 0, SWC1 stores either the high or low half of the 16 even floating-point registers.

If FR = 1, 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.

Operation:

```
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]$_{63..32-8*byte}$
      endif
      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- or double-precision floating-point formats.

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 always traps on the R4650, which does not support the .L format.

Operation:

```
T:    StoreFPR(fd, L, ConvertFmt(ValueFPR(fs, fmt), fmt, L))
```

Exceptions:

Coprocessor unusable exception
Floating-Point exception

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 0 1 0 0 0 1	fmt	0 0 0 0 0 0	fs	fd	TRUNC.W 0 0 1 1 0 1
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} -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 (*e.g.* .D)
Inexact exception
Overflow exception

FPU Instruction Opcode Bit Encoding

Figure B.3 shows the bit encoding for FPU instructions.

Opcode

28..26 31..29	0	1	2	3	4	5	6	7
0								
1								
2		COP1						
3								
4								
5								
6		LWC1				LDC1		
7		SWC1				SDC1		

sub

23..21 25..24	0	1	2	3	4	5	6	7
0	MF	DMF ηδ	CF	γ	MT	DMT ηδ	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

18..16 20..19	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

function

2..0 5..3	0	1	2	3	4	5	6	7
0	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1	ROUND.L η	TRUNC.L η	CEIL.L η	FLOOR.L η	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	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

Key to Table

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 the R4650.

η Valid when 64-bit operand opcodes are enabled.

Figure B.3 Bit Encoding for FPU Instructions

Cache Operations Timing Appendix C

Introduction

This appendix lists cycle operation counts and caveats for R4650 cache operations timing.

Caveats About Cache Operations

- All cycle counts are in processor cycles.
- All cache ops have lower priority than cache misses, write backs and external requests. If the write back buffer contains unwritten data when a cache op is executed, the write back buffer will be retired before the cache op is begun.

If an instruction cache miss occurs at the same time as a cache op is executed, the instruction cache miss will be handled first. Cache ops are mutually exclusive with respect to data cache misses. External requests will be completed before beginning a cache op.

- For all data cache ops the cache op machine waits for the store buffer and response buffer to empty before beginning the cache op. This can add 3 cycles to any data cache op if there is data in the response buffer or store buffer. The response buffer contains data from the last data cache miss that has not yet been written to the data cache. The store buffer contains delayed store data waiting to be written to the data cache.
- Cache ops of the form *xxxx_Writeback_xxxx* may perform a write back which will fill the write back buffer. Write backs can affect subsequent cache ops, since they will stall until the write back buffer is written back to memory. Cache ops which fill the write back buffer are noted as (writeback) in the following tables.
- All cycle counts are best case assuming no interference from the mechanisms described above.

Cache Operations Tables

Table C.1 and Table C.2 show data cache and instruction cache operations information. A detailed explanation of the Fill_I equation follows Table C.2.

Code[1]	Name	Number of Cycles
0	Index_Writeback_Invalidate_D	10 cycles if the cache line is clean. 12 cycles if the cache line is dirty (Writeback).
1	Index_Load_Tag_D	7 cycles.
2	Index_Store_Tag_D	8 cycles.
3	Create_Dirty_Exclusive_D	10 cycles for a cache hit. 13 cycles for a cache miss if the cache line is clean. 15 cycles for a cache miss if the cache line is dirty (Writeback).
4	Hit_Invalidate_D	7 cycles for a cache miss. 9 cycles for a cache hit.
5	Hit_Writeback_Invalidate_D	7 cycles for a cache miss. 12 cycles for a cache hit if the cache line is clean. 14 cycles for a cache hit if the cache line is dirty (Writeback).
7	Hit_Writeback_D	7 cycles for a cache miss. 10 cycles for a cache hit if the cache line is clean. 14 cycles for a cache hit if the cache line is dirty (Writeback).

Note:
[1] Code number corresponds to the code column of the CACHE instruction in Appendix A.

Table C.1 Primary Data Cache Operations

Code[1]	Name	Number of Cycles
0	Index_Invalidate_I	7 cycles.
1	Index_Load_Tag_I	7 cycles.
2	Index_Store_Tag_I	8 cycles.
3	n/a	n/a
4	Hit_Invalidate_I	7 cycles for a cache miss. 9 cycles for a cache hit.
5	Fill_I	Cycle number must be calculated based on the system response to a memory access, because Fill_I causes an instruction cache refill from memory. This equation yields the number of processor cycles for a Fill_I cache op:[2] Number_of_cycles_for_a_Fill_I_CacheOp = 10 + {0 - (SYSDIV - 1)} + (2 x SYSDIV) + (ML x SYSDIV) + (D x SYSDIV) [3]
6	Hit_Writeback_I	7 cycles for a cache miss. 20 cycles for a cache hit (Writeback).

Note:

[1] Code number corresponds to the code column of the CACHE instruction in Appendix A.

[2] For definitions and discussion of the Fill_I equation variables refer to the subsection "Details of the Fill_I Equation," which follows this table.

[3] The term {0 - (SYSDIV - 1) has a value between 0 and (SYSDIV - 1), depending on the alignment of the execution of the cache op with the system clock.

Table C.2 Primary Instruction Cache Operations

Fill_I Equation Definitions

These are the definitions for the Hit_Writeback_I equation in Table C.2:

SYSDIV: Number of processor cycles per system cycle; ranges from 2 - 8.

ML: Number of system cycles of memory latency, defined as the number of cycles the SysAD bus is driven by the external agent before the first double word of data appears.

D: Number of system cycles required to return the block of data, defined as the number of cycles beginning when the first double word of data appears on the SysAD bus and ending when the last double word of data appears on the SysAD bus, inclusive.

Integrated Device Technology, Inc.

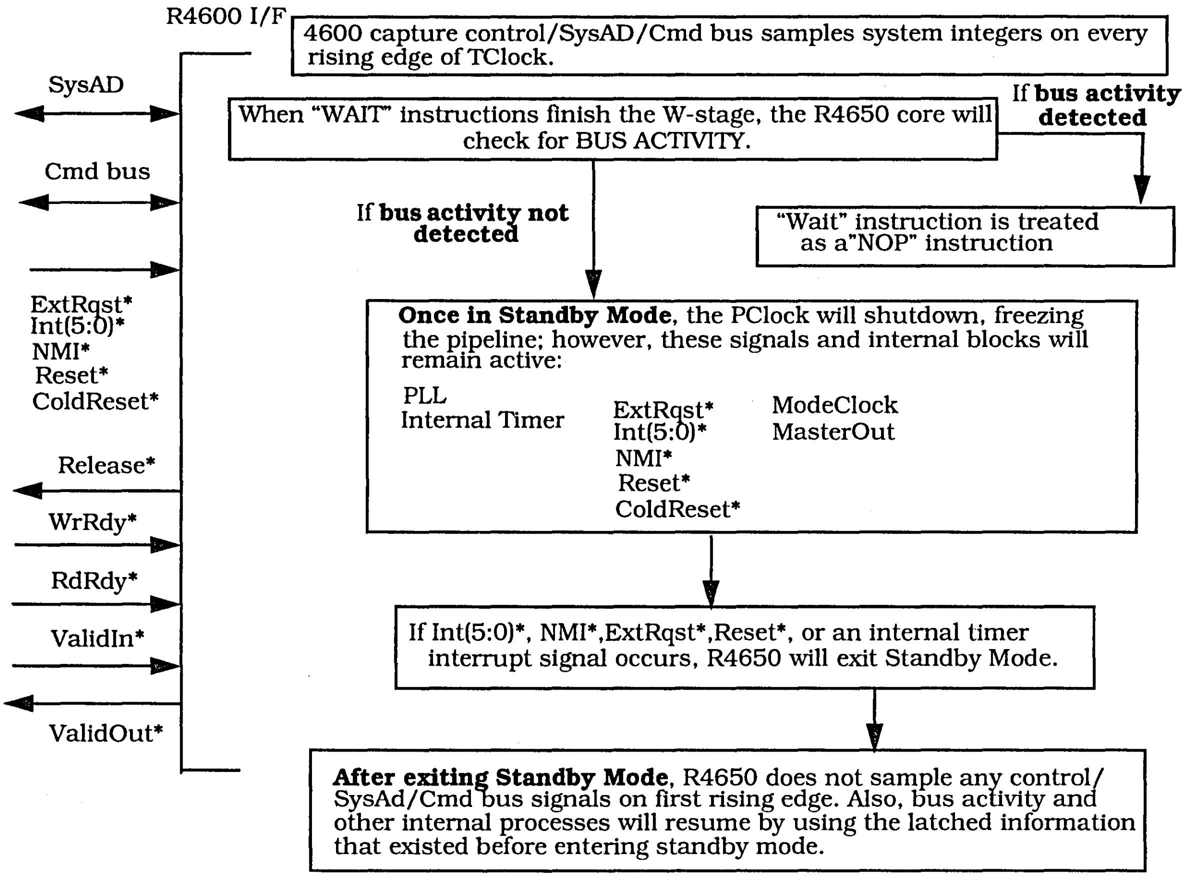
Standby Mode Operation Appendix D

The Standby Mode operation is a means of reducing the internal core's power consumption when the CPU is in a "standby" state. In this section, the Standby Mode operation is discussed.

Entering Standby Mode

To enter standby mode, first execute the WAIT instruction. When the WAIT instruction finishes the W pipe-stage, if the **SysAD** bus is currently idle, the internal clocks will shut down, thus freezing the pipeline. The PLL, internal timer, some of the input pin clocks (**Int[5:0]***, **NMI***, **ExtRqst***, **Reset*** and **ColdReset***), and the output clock (**ModeClock**) will continue to run. If the conditions are not correct when the WAIT instruction finishes the W pipe-stage (i.e., the **SysAD** bus is not idle), the WAIT is treated as a NOP.

Once the CPU is in standby mode, any interrupt, including **ExtRqst*** or **Reset***, will cause the CPU to exit standby mode. Figure D.1, located on page 2, illustrates the Standy Mode Operation.



4600 capture control/SysAD/Cmd bus samples system integers on every rising edge of TClock.

When "WAIT" instructions finish the W-stage, the R4650 core will check for BUS ACTIVITY.

"Wait" instruction is treated as a"NOP" instruction

Once in Standby Mode, the PClock will shutdown, freezing the pipeline; however, these signals and internal blocks will remain active:

PLL
Internal Timer

ExtRqst*
Int(5:0)*
NMI*
Reset*
ColdReset*

ModeClock
MasterOut

If Int(5:0)*, NMI*,ExtRqst*,Reset*, or an internal timer interrupt signal occurs, R4650 will exit Standby Mode.

After exiting Standby Mode, R4650 does not sample any control/ SysAd/Cmd bus signals on first rising edge. Also, bus activity and other internal processes will resume by using the latched information that existed before entering standby mode.

Note: During standby mode, all control signals for the CPU must be deasserted or put into the appropriate state, and all input signals, except Int(5:0)*, NMI*,Reset*,ColdReset*, and ExtRqst*, must remain unchanged. If a change occurs, the signal will be unaffected.

Figure D.1 Standby Mode Operation

Coprocessor 0 Hazards Appendix E

Introduction

This appendix identifies the R4650 Coprocessor 0 hazards. Certain combinations of instructions are not permitted because the results of executing such combinations are unpredictable in combination with some events, such as pipeline delays, cache misses, interrupts, and exceptions.

Most hazards result from instructions modifying and reading state in different pipeline stages. Such hazards are defined between pairs of instructions, not on a single instruction in isolation. Other hazards are associated with restartability of instructions in the presence of exceptions.

List of Hazards

These are the CP0 hazards:

- An mtc0 CAlg must not change the field corresponding to the address space that is currently active. The result is undefined.
- An mtc0 that changes any base or bounds register must be done in unmapped space. Mapped space cannot be entered for five instructions following a change to these registers.
- An mtc0 followed by an mfc0 is undefined. One instruction delay between mtc0 and mfc0 is needed for proper operation.
- When DWatch is enabled, the two instruction immediately following may not be checked for a match with the watch value.
- When IWatch is enabled, the five instructions following may not be checked for a match with the I match value.
- When bit 23 of the Status register is changed, refills to set A may not be disabled until five instructions later.
- When bit 24 of the Status register is changed, refills to set A may not be disabled until three instructions later.

Integrated Device Technology, Inc.

Integer Multiply Scheduling

Appendix F

Integer Multiply Scheduling

Integer multiply performance is substantially enhanced in the R4650. The R4650 adds a MAD instruction (multiply-accumulate, with HI and LO as the accumulator). Multiply performance is 2 cycles repeat, 3 cycles of latency for 16-bit operands (-2^{15} to $2^{15}-1$). Multiply-accumulate and multiplication (DMULT and DMULTU) for 64-bit operands are also supported.

The MAD (multiply/add) and MADU (multiply/add unsigned) are defined as follows, where HI and LO act as a 64-bit accumulator. These instructions do not trap on addition overflow.

MAD rs, rt	temp ← (HI $_{31..0}$ || LO $_{31..0}$) + ((rs $_{31}$)^{32} || rs$_{31..0}$) x ((rt$_{31}$)^{32} || rt$_{31..0}$)
	HI ← (temp $_{63}$)^{32} || temp $_{63..32}$
	LO ← (temp $_{31}$)^{32} || temp $_{31..0}$

MADU rs, rt	temp ← (HI $_{31..0}$ || LO $_{31..0}$) + (0^{32} || rs$_{31..0}$) x (0^{32} || rt$_{31..0}$)
	HI ← (temp $_{63}$)^{32} || temp $_{63..32}$
	LO ← (temp $_{31}$)^{32} || temp $_{31..0}$

In addition, the R4650 implements another new multiply opcode that allows the multiply result to be returned directly to the integer register file:

MUL rd, rs, rt	temp ← rs $_{31..0}$ x rt $_{31..0}$
	rd ← (temp$_{31}$)^{32} || temp $_{31...0}$
	HI ← undefined
	LO ← undefined

After executing this instruction, the HI and LO registers are undefined. For 16-bit operands, the latency of MUL is 3 cycles, with a repeat rate of 2 cycles. The MUL instruction will also unconditionally slip or stall for all but 2 cycles of its latency.

The performance of integer multiply and divide is summarized in Table F.1.

Opcodes	Condition	Latency	Repeat	Stall
MULT, MAD	$-2^{15} < rt < 2^{15}-1$	3	2	0
MULTU, MADU	$0 < rt < 2^{15}-1$	3	2	0
MULT, MAD	$rt < -2^{15}$ or $rt > 2^{15}-1$	4	3	0
MULTU, MADU	$rt > 2^{15}-1$	4	3	0
MUL	$-2^{15} < rt < 2^{15}-1$	3	2	1
	$rt < -2^{15}$ or $rt > 2^{15}-1$	4	3	2
DMULT, DMULTU	any	6	5	0
DIV, DIVU	any	36	36	0
DDIV, DDIVU	any	68	68	0

Table F.1 Integer Multiply and Divide Performance

As a special case, a MAD or MADU that is followed by a MUL instruction has one additional cycle of repeat above the value specified in the table.

In the R4600, the MFLO and MFHI instructions do not make their results available immediately. If the R4600 instruction references the MFLO/MFHI destination, then a 1-cycle slip occurs. On the R4650, however, the result is available immediately and there is no slip.

Integrated
Device Technology, Inc.

2975 Stender Way
P.O. Box 58015
Santa Clara, CA 95052-8015
800-345-7015
FAX 408-492-8674

ELECTRONIC ACCESS

Internet:	www.idt.com
E-Mail:	info@idt.com
FAX-On-Demand:	800-9-IDT-FAX (in U.S.) 408-492-8391 (outside U.S.)