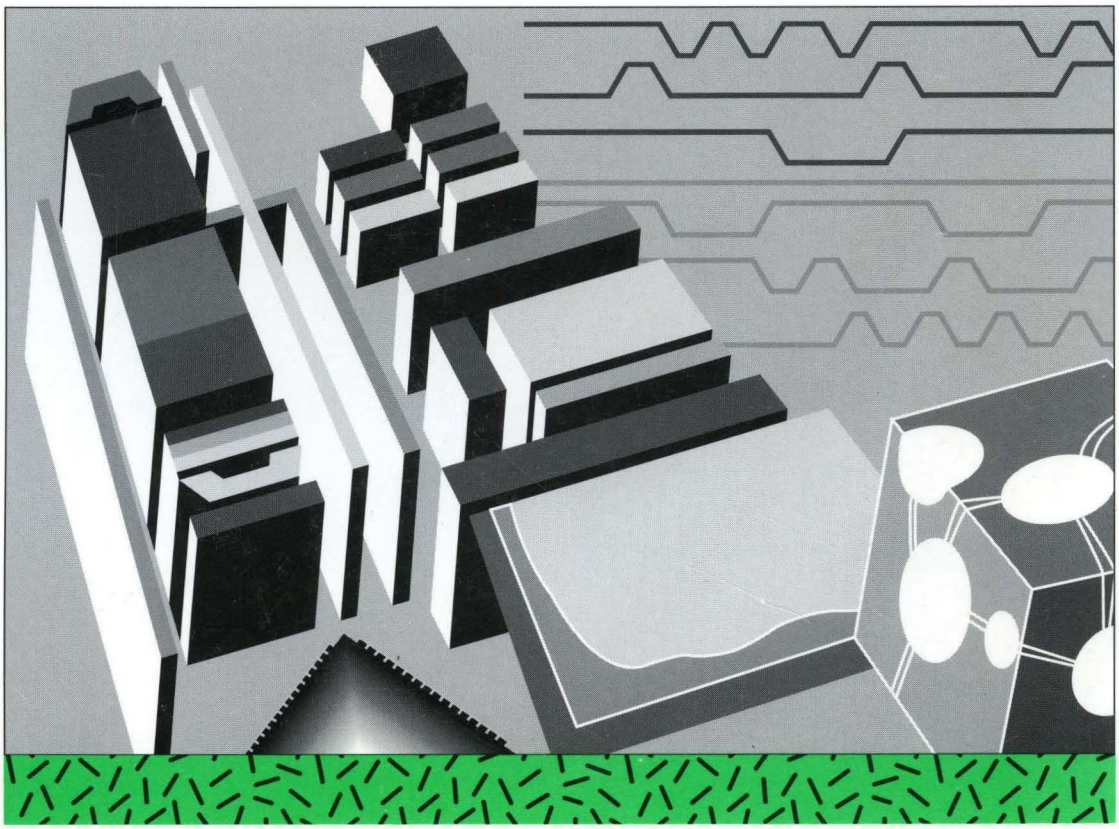
VR4300™ MIPS RISC Microprocessor

VR Series™

User's Manual



VR4300™ MIPS RISC Microprocessor

June 1995

NEC

NEC

MIPS V_R4300 Microprocessor User's Manual

MIPS Technologies, Inc.

2011 North Shoreline

Mountain View, California 94039-7311

http://www.mips.com

Acknowledgements

This book is a result of the labors of the entire V_R4300 development team. Specific acknowledgement goes to the following:

Jack Choquette

Mayank "Mike" Gupta

Andy Hsu

James Machale

Stephen Rhodes

Barbara Zivkov

--Joe Heinrich

April 1995

Contents

2 The V_R4300 Processor Pipeline

3 Memory Management

4

CPU Exception Processing

5 Floating-Point Operations

6 Floating-Point Exceptions

7

V_R4300 Processor Signal Descriptions

8 Initialization Interface

9

Clock Interface

10 Cache Organization and Operation

11 System Interface

12 JTAG Interface

13

V_R4300 Processor Interrupts

14 Power Management

15 Electrical Specifications

Introduction

1

The MIPS V_R4300 microprocessor is a low-cost, low-power microprocessor developed for interactive consumer applications including set-top terminals and video games. The V_R4300 provides performance equivalent to a high-end personal computer, but at less than a tenth of the cost.

The V_R4300 is compatible with the MIPS I, MIPS II, and MIPS III Instruction Set Architecture (ISA). This guarantees that user programs conforming to the ISA execute on any MIPS hardware implementation.

The chip does not provide on-chip support for a secondary cache or multiprocessing.

1.1 Processor Characteristics

The V_R4300 processor has the following characteristics:

- 64-bit processing
- optimized 5-stage pipeline, 16 Kbyte I-cache and 8 Kbyte D-cache size, and 32-double-entry TLB size
- 32-bit physical address space, 40-bit virtual address space
- single datapath shared by integer and floating point operations
- flush buffer, used as temporary data storage for outgoing data
- instruction trace support
- low-voltage operation (3.3 volt)
- cache bank partitioning; only one of four banks (instruction cache) or two banks (data cache) are powered on at any time, saving power on each cache access
- dynamic logic design, reducing transistor count and dissipated power
- write-back data cache, reducing store activity on the system bus
- instruction cache prefetching; retrieving two consecutive 32-bit words on each instruction cache access reduces cache activity and power consumption
- instruction micro-TLB, which holds the translations for two most-recently-used instruction pages, eliminating a large number of main TLB accesses
- power management features, which include the following three operating modes
  - standard operating mode, operating at 40 MHz (external) and 80 MHz (internal)
  - reduced power mode, in which clock speeds are reduced to a quarter of normal: 20 MHz internal, 10 MHz external
  - power-down mode, in which processor state is written out to nonvolatile RAM (NVRAM) and retained there for "instant-on" power-up capability
- ceramic packaging (179-pin ceramic pin-grid array, CPGA)
- software compatibility with all MIPS processors

1.2 Processor Implementation

This section describes the following:

- the 64-bit architecture of the V_R4300 processor
- the CPU instruction pipeline (described in detail in Chapter 2)
- an overview of the CPU registers (detailed in Chapters 3 and 4)
- data formats and byte ordering
- coprocessors, including the System Control Coprocessor, CP0
- caches and memory, including a description of instruction and data caches, the memory management unit (MMU), and the translation lookaside buffer (TLB).

64-bit Architecture

The natural mode of operation for the V_R4300 processor is as a 64-bit microprocessor; however, 32-bit applications may be run when the processor operates as a 64-bit processor. Figure 1-1 is an internal block diagram of the V_R4300 processor.

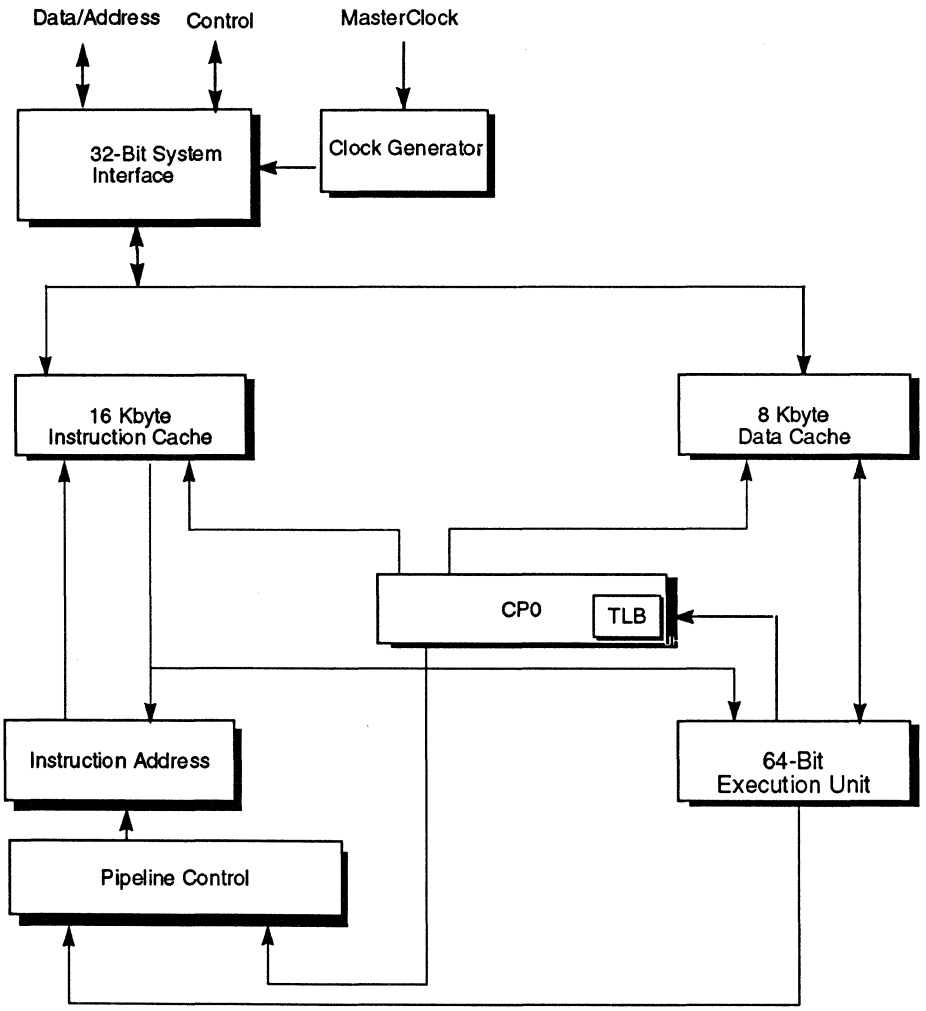


Figure 1-1 V_R4300 Processor Internal Block Diagram

The **Execution Unit** contains the hardware resources necessary to execute all of the MIPS integer and floating point instructions. It contains a 64-bit-wide register file, a 64-bit-wide integer/mantissa data path, and a 12-bit-wide exponent data path.

Coprocessor 0 contains the memory management unit (MMU) and the exception processing unit. The memory management unit is responsible for effective virtual-to-physical address translation, and for performing memory access checks between kernel, supervisor, user memory segments.

The **Translation Lookaside Buffer** (TLB) translates virtual addresses to physical addresses. The V_R4300 processor supports a virtual address space (VSIZE) of 40 bits, and physical address space (PSIZE) of 32 bits. The V_R4300 (like the V_R4000/V_R4400) supports the following page sizes:

- 4 Kbytes
- 16 Kbytes
- 64 Kbytes
- 256 Kbytes
- 1 Mbyte
- 4 Mbytes
- 16 Mbytes.

The TLB contains 32 entries, each entry mapping to an odd/even pair of page frame numbers.

The **Exception Processing Unit** contains all of the CP0 registers.

Clock Generator multiplies the input clock frequency (**MasterClock**) to produce the pipeline clock. The ratio of **PClock** (pipeline clock) to **MasterClock** is set by the **DivMode(1:0)** pins. The DivMode values of 0, 1, 2 and 3 define **PClock** to **MasterClock** ratios of 1:1, 1.5:1, 2:1, and 3:1 respectively. The system interface clock runs at the same frequency as the **MasterClock**. Using the Reduced Power (*RP*) bit of the *Status* register, both pipeline and interface clocks can be switched to run at quarter speed. To minimize the skew between the input clock and the internal clocks, the chip uses phase-locked loop (PLL) technology.

Pipeline Control assures that the V_R4300 instruction pipeline operates properly when conditions such as the following occur: cache miss, flush buffer full, multicycle instruction, or system exception.

Instruction Address calculates the effective address of the next instruction to be fetched. It contains the PC incrementer, branch address adder, and the conditional branch address selector.

Instruction Cache is direct mapped, virtually indexed, and physically tagged. Each line includes 8 instructions, 21 tag bits, and 1 valid bit. The cache data interface is 64 bits wide. Cache parity is not supported.

The **Data cache** is a directly-mapped, virtually-indexed, physically-tagged, write-back cache. Each cache line includes 4 words of data, 21 tag bits, 1 dirty bit, and 1 valid bit. The cache read operation takes one cycle, but a store operation keeps the data cache busy for two cycles. Cache parity is not supported.

The **System interface** allows the processor to access external resources. The System interface is a 32-bit-wide, multiplexed address and data bus, with clock signals, interrupts, and a number of control signals.

Instruction Pipeline

The V_R4300 processor has a 5-stage instruction pipeline that is shared by integer and floating-point operations; under normal circumstances, one instruction is issued each cycle.

The instruction pipeline of the V_R4300 processor operates at a multiple of **MasterClock**, as defined by the **DivMode(1:0)** bits. The processor achieves high throughput by shortening register access times and implementing virtually-indexed caches.

Processor Register Overview

The processor provides the following registers:

- 32 64-bit general purpose registers, *GPR*s
- 32 64-bit floating-point general purpose registers, *FPR*s

In addition, the processor provides the following special registers:

- 64-bit Program Counter, the *PC* register
- 64-bit *HI* register, containing the integer multiply and divide upper doubleword result
- 64-bit *LO* register, containing the integer multiply and divide lower doubleword result
- 1-bit Load/Link *LLBit* register
- 32-bit floating-point *Implementation/Revision* register, FCR0
- 32-bit floating-point *Control/Status* register, FCR31

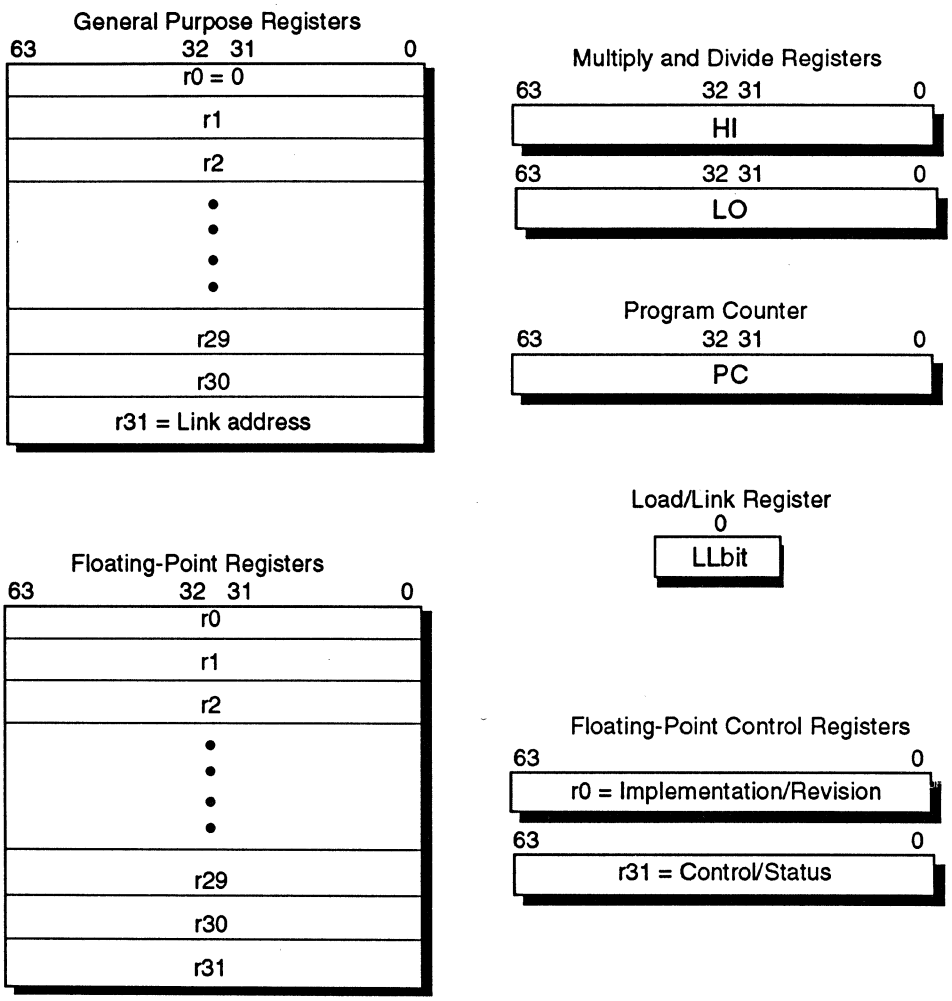
Two of the CPU general purpose registers have assigned functions:

- *r0* is hardwired to a value of zero, and can be used as the target register for any instruction whose result is to be discarded. *r0* can also be used as a source when a zero value is needed.
- *r31* is the link register used by Jump and Link instructions. It can be used by other instructions, with caution.

CPU registers can operate as either 32-bit or 64-bit registers, depending on the V_R4300 processor mode of operation.

Figure 1-2 shows the V_R4300

processor registers.



Register width depends on mode of operation: 32-bit or 64-bit

Figure 1-2 V_R4300 *Processor Registers*

The V_R4300 processor has no *Program Status Word* (PSW) register as such; this is covered by the *Status* and *Cause* registers incorporated within the System Control Coprocessor (CP0).

Data Formats and Addressing

The V_R4300 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 all of the larger data formats—halfword, word, doubleword—can be configured in either big-endian or little-endian order through the *BE* bit in the *Config* register. Endianness refers to the location of byte 0 within the multi-byte data structure. Figures 1-3 and 1-4 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 V_R4300 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 1-3 shows this configuration.

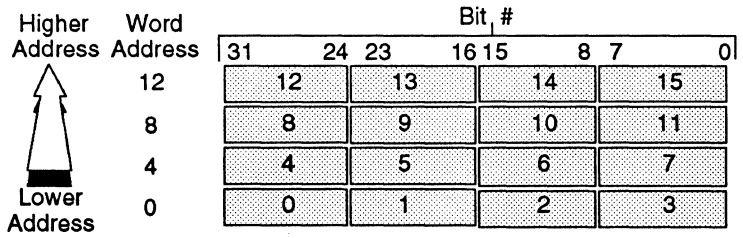


Figure 1-3 Big-Endian Byte Ordering

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 1-4 shows this configuration.

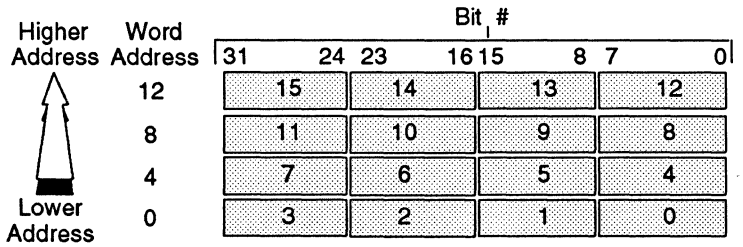


Figure 1-4 Little-Endian Byte Ordering

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 1-5 and 1-6 show little-endian and big-endian byte ordering in doublewords.

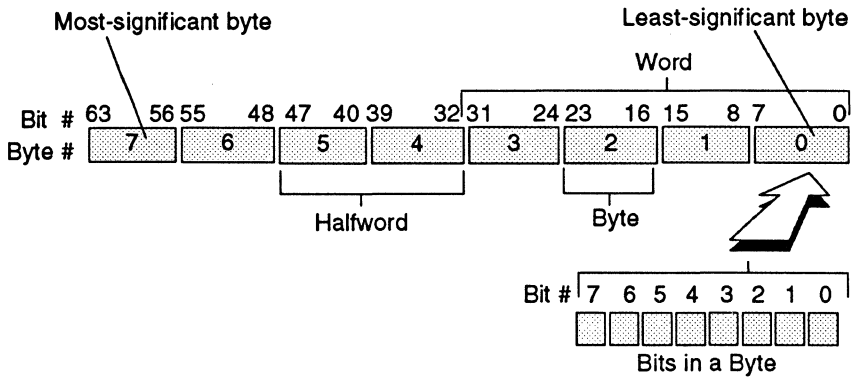


Figure 1-5 Little-Endian Data in a Doubleword

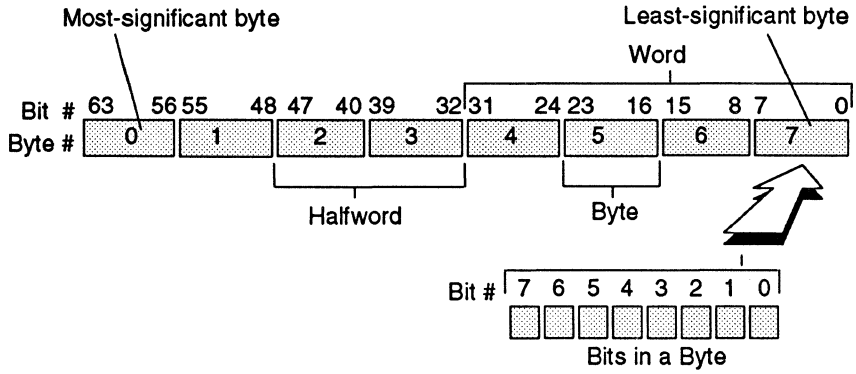


Figure 1-6 Big-Endian Data in a Doubleword

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

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

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

LWL	**LWR**	**SWL**	**SWR**
LDL	**LDR**	**SDL**	**SDR**

These instructions are used in pairs to provide addressing of misaligned words. Addressing misaligned data incurs one additional instruction cycle over that required for addressing aligned data.

Figures 1-7 and 1-8 show the access of a misaligned word that has byte address 3.

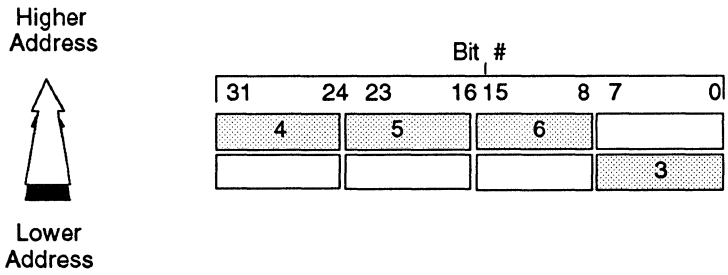


Figure 1-7 Big-Endian Misaligned Word Addressing

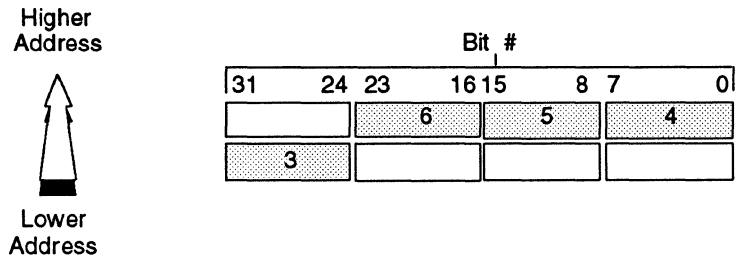


Figure 1-8 Little-Endian Misaligned Word Addressing

Coprocessors (CP0-CP2)

The MIPS ISA 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 reserved for on-chip, floating-point coprocessor operations.
- Coprocessor 2 (**CP2**) is reserved for future definition by MIPS.

CP0 and CP1 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, supervisor, and user states. CP0 also controls the cache subsystem, as well as providing diagnostic control and error recovery facilities.

The CP0 registers are described in detail in Chapters 3 and 4. Figure 1-9 shows the CP0 registers, and Table 1-1 provides a brief description of each.

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

Exception Processing Memory Management Reserved

Figure 1-9 CP0 Registers

Table 1-1 System Control Coprocessor (CP0) Register Definitions

Number	Register	Description
0	Index	Programmable pointer into TLB array
1	Random	Pseudorandom pointer into TLB array *(read only)*
2	EntryLo0	Low half of TLB entry for even virtual address (VPN)
3	EntryLo1	Low half of TLB entry for odd virtual address (VPN)
4	Context	Pointer to kernel virtual page table entry (PTE) in 32-bit addressing mode
5	PageMask	TLB Page Mask
6	Wired	Number of wired TLB entries
7	—	Reserved
8	BadVAddr	Bad virtual address
9	Count	Timer Count
10	EntryHi	High half of TLB entry
11	Compare	Timer Compare
12	SR	Status register
13	Cause	Cause of last exception
14	EPC	Exception Program Counter
15	PRId	Processor Revision Identifier
16	Config	Configuration register
17	LLAddr	Load Linked Address
18	WatchLo	Memory reference trap address low bits
19	WatchHi	Memory reference trap address high bits
20	XContext	Pointer to kernel virtual PTE table in 64-bit addressing mode
21–25	—	Reserved
26	PErr	(not used)
27	CacheErr	(not used)
28	TagLo	Cache Tag register
29	TagHi	Cache Tag register
30	ErrorEPC	Error Exception Program Counter
31	—	Reserved

Floating-Point Unit (FPU), CP1

The MIPS floating-point unit (FPU) is designated CP1; the FPU extends the CPU instruction set to perform arithmetic operations on floating-point values. The FPU, with associated system software, fully conforms to the requirements of ANSI/IEEE Standard 754–1985, *IEEE Standard for Binary Floating-Point Arithmetic.*

The FPU includes:

- **Full 64-bit Operation.** The FPU can contain either 16 or 32 64-bit registers to hold single-precision or double-precision values. The FPU also includes a 32-bit *Status/Control* register that provides access to all IEEE-Standard exception-handling capabilities.
- **Load and Store Instruction Set.** Like the CPU, the FPU uses a load- and store-based instruction set. Floating-point operations are started in a single cycle, however execution of floating-point ops are not allowed to overlap other operations.
- **Sharing Hardware.** There is no separate FPU on theV_R4300; floating-point operations are processed by the same hardware as is used for integer instructions.

Memory Management System (MMU)

The V_R4300 processor has a 32-bit physical addressing range of 4 Gbytes. However, since it is rare for systems to implement a physical memory space this large, the CPU provides a logical expansion of memory space by translating addresses composed in the large virtual address space into available physical memory addresses. The V_R4300 processor supports the following two addressing modes:

- 32-bit mode, in which the virtual address space is divided into 2 Gbytes per user process and 2 Gbytes for the kernel.
- 64-bit mode, in which the virtual address is expanded to 1 Tbyte (2^{40} bytes) of user virtual address space.

A detailed description of these address spaces is given in Chapter 3.

The Translation Lookaside Buffer (TLB)

Virtual memory mapping is assisted by a translation lookaside buffer, which holds virtual-to-physical address translations. This fully-associative, on-chip TLB contains 32 entries, each of which maps a pair of variable-sized pages ranging from 4 Kbytes to 16 Mbytes, in multiples of four.

Joint TLB

The TLB can hold both instruction and data addresses, and is thus also referred to as a joint TLB (JTLB). An address translation value is tagged with the most-significant bits of its virtual address (the number of these bits depends upon the size of the page) and a per-process identifier. If there is no matching entry in the TLB, an exception is taken and software refills the on-chip TLB from a page table resident in memory. The JTLB entry to be rewritten is selected by a value in either the *Random* or *Index* register.

Instruction Micro-TLB

The V_R4300 processor has a two-entry instruction micro-TLB (ITLB) which assists in instruction address translation. The ITLB is completely invisible to software and exists only to increase performance. Instructions access this TLB while data accesses the Joint TLB; a miss in the micro-TLB stalls the pipeline until the micro-TLB is refilled from the joint TLB. The micro-TLB is fully associative, and uses the least-recently-used (LRU) replacement algorithm. Each micro-TLB entry maps 4 kilobytes of virtual space to physical space. This ensures each ITLB entry is a subset of any single JTLB entry.

Operating Modes

The V_R4300 processor has three operating modes:

- User mode
- Supervisor mode
- Kernel mode

The manner in which memory addresses are translated or *mapped* depends on the operating mode of the CPU; this is described in Chapter 3.

Caches

The V_R4300 processor incorporates separate on-chip instruction and data caches to fill the high-performance pipeline. Each cache has its own 64-bit data path, and each can be accessed in parallel. The V_R4300 processor instruction cache holds 16 Kbytes and the data cache holds 8 Kbytes.

The V_R4300 Processor Pipeline

2

This chapter describes the basic operation of the V_R4300 processor pipeline, which includes descriptions of the delay slots (instructions that follow a branch or load instruction in the pipeline), interruptions to the pipeline flow caused by interlocks and exceptions, CP0 hazards, and V_R4300 implementation of a flush buffer.

2.1 Pipeline Stages

The CPU has a five-stage instruction pipeline; each stage takes one PCycle (one cycle of PClock, which runs at a multiple of the **MasterClock** frequency), and each PCycle has two phases: Φ1 and Φ2, as shown in Figure 2-1. Thus, the execution of each instruction takes at least 5 PCycles (2.5 **MasterClock** cycles). An instruction can take longer—for example, if the required data is not in the cache, the data must be retrieved from main memory.

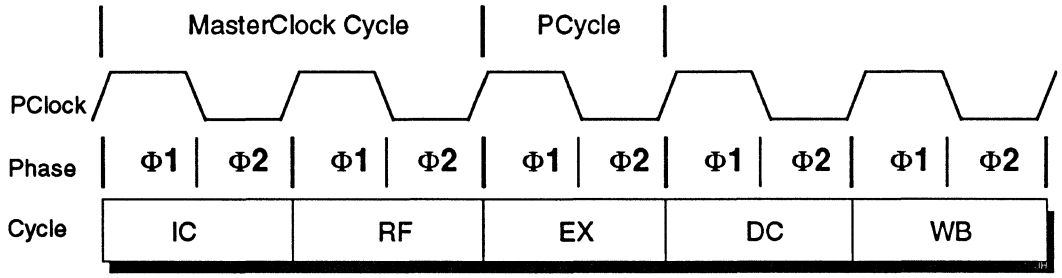


Figure 2-1 Pipeline Stages

The five pipeline stages are:

- IC - Instruction Cache Fetch
- RF - Register Fetch
- EX - Execution
- DC - Data Cache Fetch
- WB - Write Back

Once the pipeline has been filled, five instructions are executed simultaneously. Figure 2-2 shows the five stages of the instruction pipeline; the next section describes the pipeline stages.

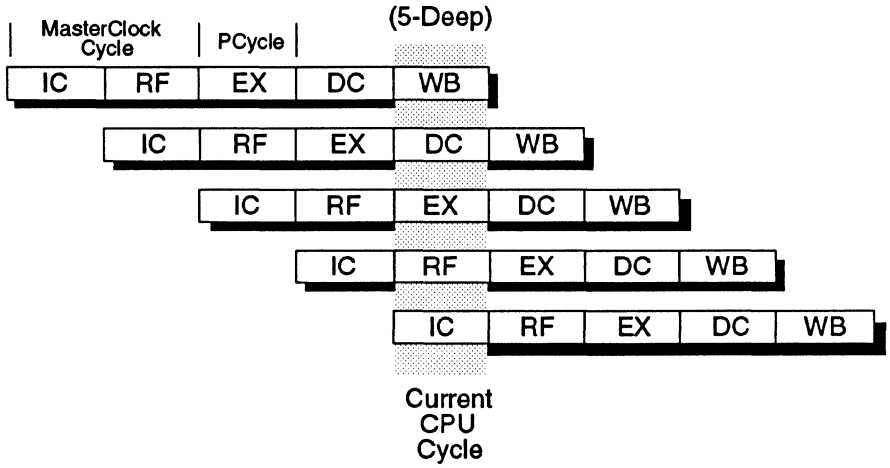


Figure 2-2 Instruction Execution in the Pipeline

Pipeline Activities

Figure 2-3 shows the activities that can occur during each pipeline stage; Table 2-1 describes these pipeline activities.

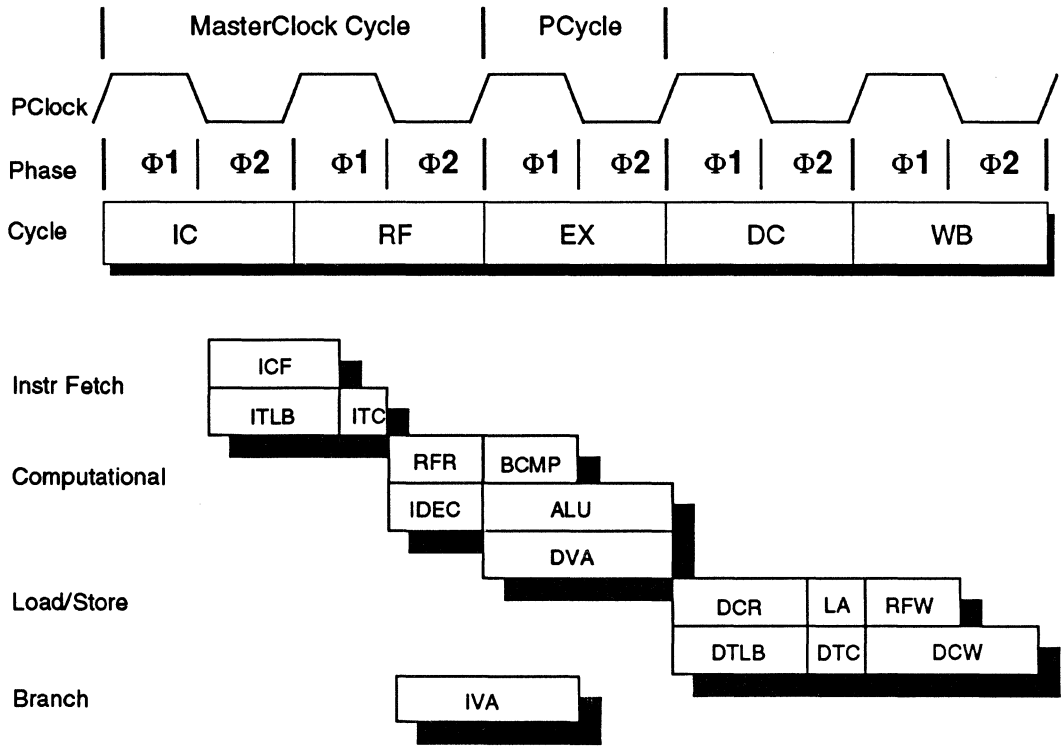


Figure 2-3 Pipeline Activities

Table 2-1 Description of Pipeline showing stage in which activities commence

Cycle	Begins During this Phase	Mnemonic	Descriptions
IC	Φ1		
	Φ2	ICF	Instruction Cache Fetch
		ITLB	Instruction micro-TLB read
RF	Φ1	ITC	Instruction cache Tag Check
	Φ2	RFR	Register File Read
		IDEC	Instruction DECode
		IVA	Instruction Virtual Address calculation
EX	Φ1	BCMP	Branch Compare
		ALU	Arithmetic Logic operation
		DVA	Data Virtual Address calculation
DC	Φ1	DCR	Data Cache Read
		DTLB	Data joint-TLB read
	Φ2	LA	Load data Alignment
		DTC	Data cache Tag Check
WB	Φ1	DCW	Data Cache Write
		RFW	Register File Write
	Φ2		

2.2 Branch Delay

The CPU pipeline has a branch delay of one cycle, as a result of the branch comparison logic operating during the EX pipeline stage of the branch, producing an instruction address that is available in the IC stage, two instructions later.

Figure 2-4 illustrates the branch delay and the location of the branch delay slot.

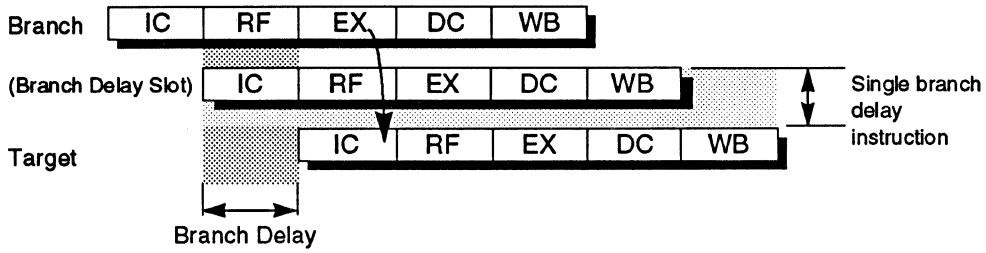


Figure 2-4 CPU Pipeline Branch Delay

2.3 Load Delay

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 V_R4

Add Instruction

```
ADD rd,rs,rt
```

IC stage. In phase 2, the fourteen least significant bits (LSBs) of the virtual address are used to address the instruction cache. The two most significant bits of this virtual address are used to select one of four instruction cache banks. The remaining LSBs are used to address the selected bank and the cache line physical page frame number (PPFN) tag. The micro-TLB translates the virtual page frame number (VPFN) to the PPFN. Late in phase 1 of the next RF pipestage, the PPFN is compared with the PPFN tag from the cache and the cache hit/miss signal is produced. The virtual PC is incremented by four so that the next sequential instruction can be fetched in the following cycle.

RF stage In phase 1 of the RF stage, the cache index is compared with the page frame number from the micro-ITLB and the cache data is read out. The cache hit/miss signal is valid late in phase 1 of the RF stage, and the virtual PC is incremented by 4 so that the next instruction can be fetched.

During phase 2, the 2-port register file is addressed with the *rs* and *rt* fields and the register data is valid at the register file output. At the same time, bypass multiplexers select inputs from either the EX- or DC-stage output in addition to the register file output, depending on the need for an operand bypass.

EX stage The ALU controls are set to do an A+B operation. The operands flow into the ALU inputs, and the ALU operation is started. The result of the ALU operation is latched into the ALU output latch during phase 2.

DC stage This stage is a NOP for this instruction. The data from the output of the EX stage (the ALU) is moved into the output latch of the DC.

WB stage During phase 1, the WB latch feeds the data to the inputs of the register file, which is addressed by the *rd* field. The file write strobe is enabled. By the end of phase 1, the data is written into the file.

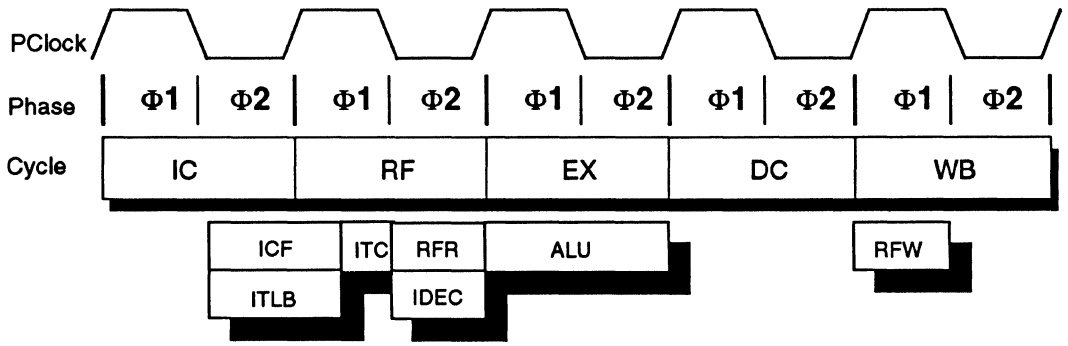


Figure 2-5 Add Instruction Pipeline Activities

Jump and Link Register

```
JALR rd,rs
```

IC stage — Same as the IC stage for the ADD instruction.

RF stage — During phase 2 of the RF stage, the register addressed by the *rs* field is read out of the file.

EX stage — During phase 1 of the EX stage, the value of register *rs* is clocked into the virtual PC latch. This value is used in phase 2 to fetch the next instruction.

The value of the virtual PC incremented during the RF stage is incremented again to produce the link address PC+8 where PC is the address of the JALR instruction. The resulting value is the PC to which the program will eventually return. This value is placed in the Link output latch of the Instruction Address unit.

DC stage — The PC+8 value is moved from the Link output latch to the output latch of the DC pipeline stage.

WB stage — Refer to the ADD instruction. Note that if no value is explicitly provided for *rd* then register 31 is used as the default. If *rd* is explicitly specified, it cannot be the same register addressed by *rs*; if it is, the result of executing such an instruction is undefined.

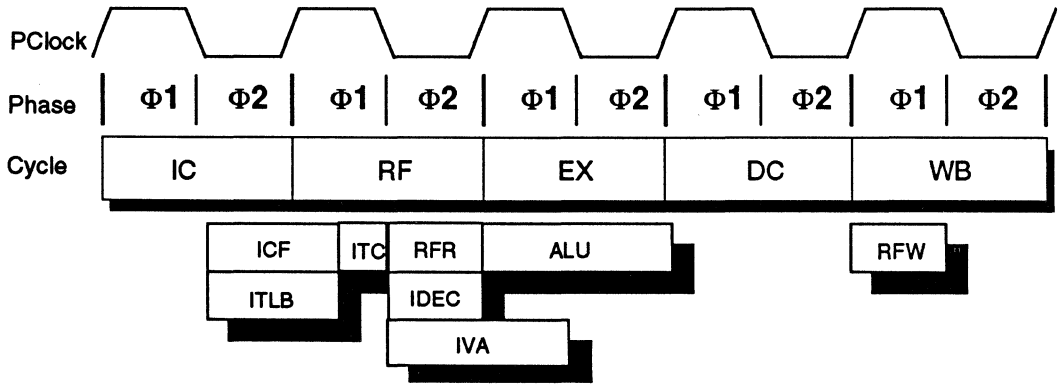


Figure 2-6 Jump and Link Register Instruction Pipeline Activities

Branch on Equal

```
BEQ rs,rt,offset
```

IC stage. Same as the IC stage for the ADD instruction.

RF stage. During phase 2, the register file is addressed with the *rs* and *rt* fields and the contents of these registers are placed in the register file output latch.

EX stage. During phase 1, a check is performed to determine if each corresponding bit position of these two operands has equal values. If they are equal, the PC is set to *PC+target*, where *target* is the sign-extended offset field. If they are not equal, the PC is set to *PC+4*.

The next PC resulting from the branch comparison is valid at the beginning of phase 2 for instruction fetch.

DC stage. This stage is a NOP for this instruction.

WB stage. This stage is a NOP for this instruction.

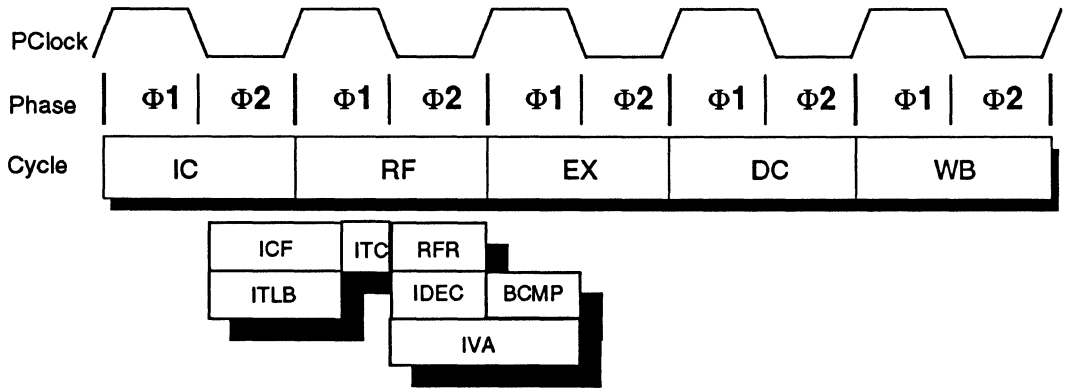


Figure 2-7 Branch on Equal Instruction Pipeline Activities

Trap if Less Than

```
TLT rs,rt
```

IC stage. Same as the IC stage for the ADD instruction.

RF stage. Same as the RF stage for the ADD instruction.

EX stage. During the phase 1, the bypass multiplexers select inputs from the RF-, EX- or DC-stage output latch, depending on the need for an operand bypass. ALU controls are set to do an A – B operation. The operands flow into the ALU inputs, and the ALU operation is started.

The result of the ALU operation is latched into the ALU output latch during phase 2.

DC stage. The sign bits of operands and of the ALU output latch are checked to determine if a *less than* condition is true. If this condition is true, the next PC is loaded with the value of the exception vector and instructions following in previous pipeline stages are killed. The *BD* bit of the CP0 *Cause* register and *EXL* bit of the *Status* register are checked.

WB stage. The *EPC* register is loaded with the value of the PC if the *less than* condition was met in the DC stage. The *Cause* register *ExcCode* field and *BD* bit are updated appropriately, as is the *EXL* bit of the *Status* register. If the less than condition was not met in the DC stage, no activity occurs in the WB stage.

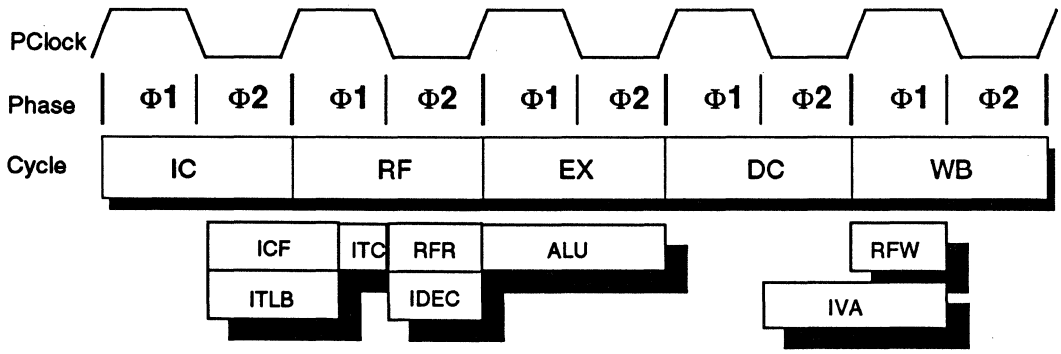


Figure 2-8 Trap if Less Than Instruction Pipeline Activities

Load Word

```
LW rt,offset(base)
```

IC stage. Same as the IC stage for the ADD instruction.

RF stage. Same as the RF stage for the ADD instruction. Note that the *base* field is in the same position as the *rs* field.

EX stage. Refer to the EX stage for the ADD instruction. For LW, the inputs to the ALU come from *GPR[base]* through the bypass multiplexer and from the sign-extended offset field. The result of the ALU operation that is latched into the ALU output latch in phase 2 represents the effective virtual address of the operand (DVA).

DC stage. The data cache is accessed in parallel with the TLB, and the cache tag field is compared with the Page Frame Number (PFN) field of the TLB entry. After passing through the load aligner, aligned data is placed in the DC output latch during phase 2.

WB stage. During phase 1, the cache read data is written into the file addressed by the *rt* field.

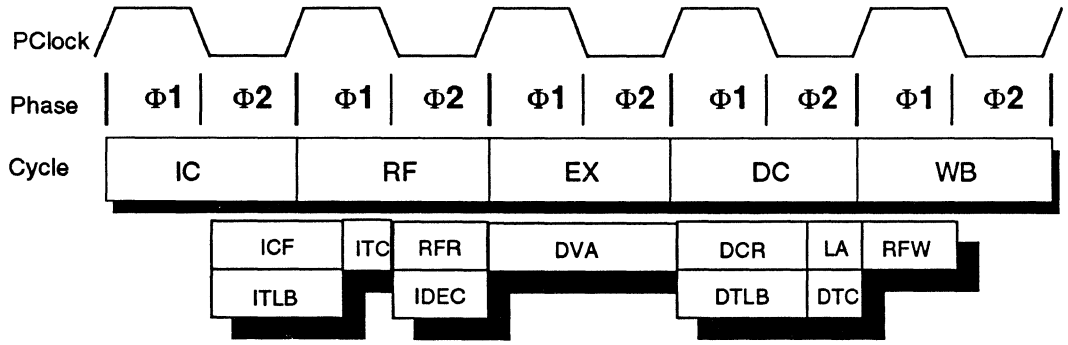


Figure 2-9 Load Word Instruction Pipeline Activities

Store Word

```
SW rt,offset(base)
```

IC stage — Same as the IC stage for the ADD instruction.

RF stage — Same as the RF stage for the LW instruction.

EX stage — Refer to the LW instruction for a calculation of the effective address. From the RF output latch the *GPR[rt]* is sent through the bypass multiplexer and into the main shifter, where the shifter performs the byte-alignment operation for the operand. The results of the ALU and the shift operations are latched in the output latches during phase 2.

DC stage — Refer to the LW instruction for a description of the cache access. Additionally, the merged data from the load aligner is moved into the store data output latch during phase 2.

WB stage — If there was a cache hit, the content of the store data output latch is written into the data cache at the appropriate word location.

Note that all store instructions use the data cache for two consecutive PCycles. If the following instruction requires use of the data cache, the pipeline is stalled for one PCycle to complete the writing of an aligned store data.

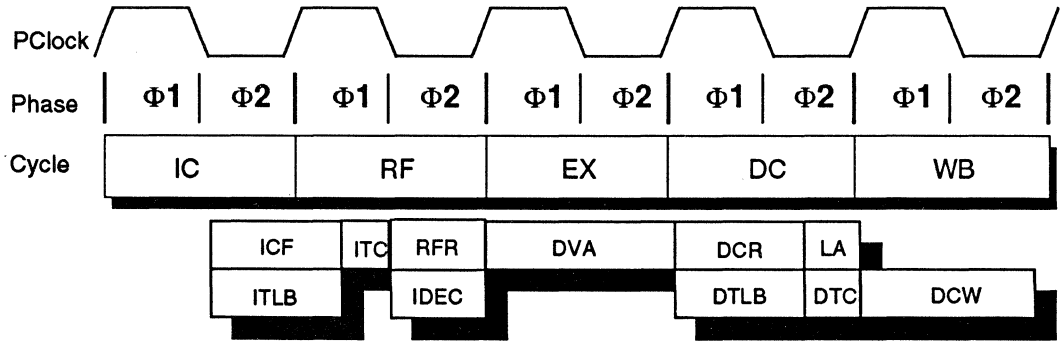


Figure 2-10 Store Word Instruction Pipeline Activities

2.5 Interlock and Exception Handling

The processor has a simple pipe which allows the overlap of instruction execution across the five pipe stages. The pipeline operates with in-order issue, in-order execution, and in-order completion, following the same order as the instruction stream.

Pipeline flow is interrupted when an interlock condition is detected or when an exception occurs. The interlock condition is resolved by stalling the whole pipeline, while an exception aborts the relevant instruction as well as all those that follow, and calls an exception handler from a pre-defined address.

As shown in Figure 2-11, all interlock and exception conditions are collectively referred to as faults.

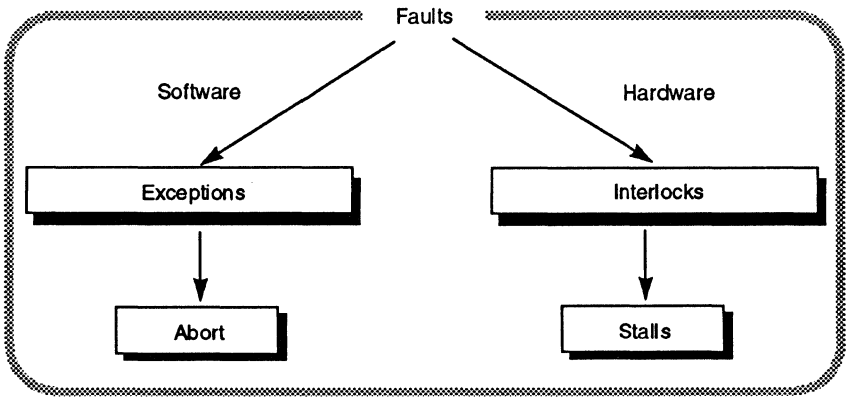
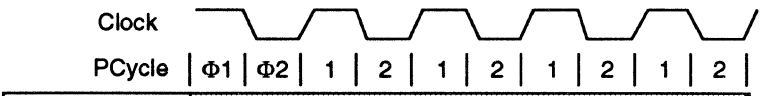


Figure 2-11 Interlocks, Exceptions, and Faults

For cases of simultaneous interlocks and exceptions from different pipeline stages, the interlocks and exceptions from the later pipeline stages are processed before those from earlier pipeline stages. For instance, an exception from the DC stage takes precedence over a coincident exception from the RF stage.

Figure 2-12 lists different interlocks and exceptions, in decreasing order of priority, for simultaneous interlocks and exceptions from the same pipeline stage, with the highest priority listed first.



State	Pipeline Stage				
	IC	RF	EX	DC	WB
Interlock		ITM	LDI	DCM	CP0I
		ICB	MCI	DCB	
			CPI	COp	
	IC	**RF**	**EX**	**DC**	**WB**
Exceptions		IADE	SYSC	RST	
		ITLB	BRPT	NMI	
		IBE	CPU	OVFL	
			RSVD	TRAP	
				FPE	
				CPE	
				DADE	
				DTLB	
				WAT	
				DBE	
				INTR	

Figure 2-12 Correspondence of Pipeline Stage to Interlock and Exception Condition

Table 2-2 describes the pipeline interlocks and exceptions listed in Figure 2-12.

Table 2-2 Description of Pipeline Exceptions and Interlocks

Exception	Description	Interlock	Description
IADE	Instruction Address Error	**ITM**	Instruction TLB Miss
ITLB	Instruction TLB Exception	**ICB**	Instruction Cache Busy
IBE	Instruction Bus Error	**MCI**	Multi-cycle Interlock
SYSC	SYSCALL Instruction	**LDI**	Load Interlock
BRPT	Breakpoint Instruction	**CPI**	Coprocessor 2 Interlock
RSVD	Reserved Instruction	**DCM**	Data Cache Miss
CPU	Coprocessor Unusable	**DCB**	Data Cache Busy
RST	External Reset	**CP0I**	CP0 Bypass Interlock
NMI	External NMI	**COp**	Cache Op
OVFL	Integer Overflow		
TRAP	TRAP instruction exception		
FPE	Floating-point Exception		
CPE	Coprocessor 2 Exception		
DADE	Data Address Error		
DTLB	Data TLB exception		
WAT	Reference to Watch Address		
DBE	Data Bus Error		
INTR	External Interrupt signals		

2.6 Pipeline Interlocks and Exceptions

When an interlock or exception condition arises, pipeline flow is interrupted. Depending upon whether the condition is an interlock or an exception, one of the following occurs:

- If an interlock condition arises, the pipeline remains *stalled* until the interlock is corrected by hardware.
- If an exception occurs, the exception-causing instruction and all that follow are *aborted*, the exception is resolved by software, and the pipeline restarted and reloaded.

Pipeline interlocks and pipeline exceptions are described in the following section. The exceptions themselves are described in Chapter 5.

Bypassing, which allows data and conditions produced in the EX, DC and WB stages of the pipeline to be made available to the EX stage, is also described in this section.

Pipeline Interlocks

When an interlock condition occurs, the pipeline stalls and remains stalled until the interlock is corrected. Should pipeline stall requests from different stages arise simultaneously, the Pipeline Control Unit prioritizes the stall requests. For instance, a stall request from the DC stage is always allowed to be resolved before a simultaneous RF-stage stall request, since both may require the same resource (TLB, memory) to be resolved. The EX stage is allowed to stall in order to complete a multicycle instruction as long as there is no load dependency between itself (the EX stage) and the DC stage. Interlock conditions for each pipeline stage are shown in Figure 2-12 and described in Table 2-2.

The remainder of this section describes in detail the following pipeline interlocks (listed here in decreasing priority, with ITM having the highest priority and CPOI the lowest):

- Instruction TLB Miss (ITM)
- Instruction Cache Busy (ICB)
- Load Interlock (LDI)
- Multicycle Instruction Interlock (MCI)
- Coprocessor 2 Interlock (CPI)
- Data Cache Miss (DCM)
- Data Cache Busy (DCB)
- Cache Operation (COp)
- CP0 Bypass Interlock (CPOI)

Instruction TLB Miss (ITM)

A pipeline stall due to an Instruction TLB Miss occurs when the virtual address of the next instruction to be fetched is not found in the instruction micro-TLB (ITLB). The pipeline stalls when the micro-TLB miss is detected in the RF stage, whereupon the pipeline controller notifies the micro-TLB to proceed in servicing the stall. The pipeline starts running again when the micro-TLB has been updated from the JTLB.

A miss penalty of 3 PCycles is incurred when the micro-TLB is updated from the JTLB.

If the virtual address also misses in the JTLB, an exception is taken which overrides the stall to allow the handler to update the JTLB. Once the update is completed, the instruction fetch is re-executed. This initiates a repeat of the ITM stall until the micro-TLB is updated from the JTLB, which was just updated by the exception handler.

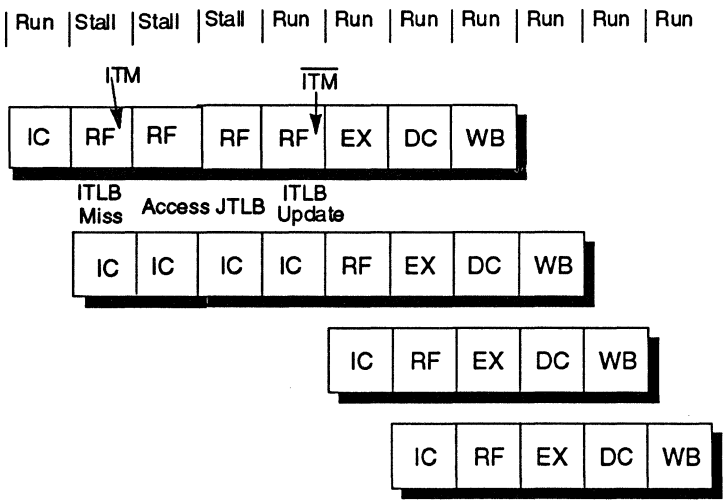


Figure 2-13 Instruction TLB Miss Interlock

Instruction Cache Busy (ICB)

A pipeline stall due to an Instruction Cache Busy interlock occurs when the next instruction is not found in the instruction cache, and the cache cannot service the Instruction Fetch. The pipeline stalls when the instruction cache miss is detected in the RF stage. After detecting the stall, the pipeline controller notifies the instruction cache to proceed in servicing the stall.

The pipeline begins running again after the entire cache line has been written into the instruction cache.

When the instruction cache is busy with a CACHE operation and the Instruction Fetch cannot be serviced, a Cache Operation (COp) exception is taken, not ICB.

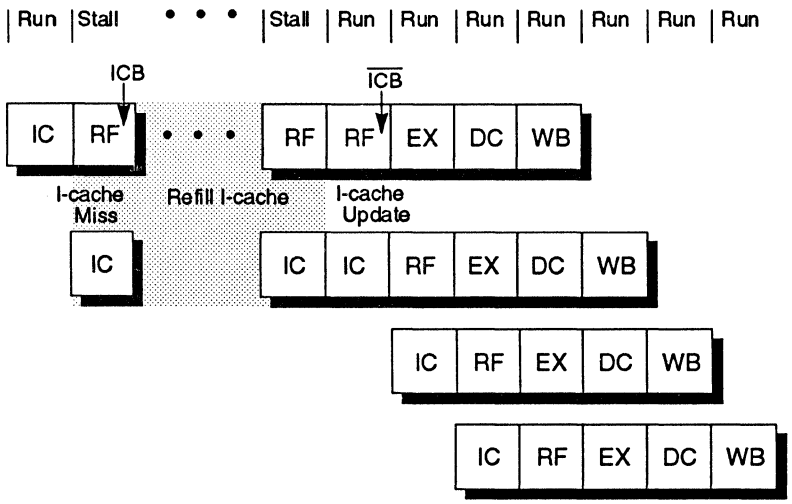


Figure 2-14 Example of an Instruction Cache Busy Interlock

Multicycle Instruction Interlock (MCI)

A pipeline stall due to a Multicycle Interlock occurs when an instruction with an execution latency of more than one pipeline clock enters the EX stage.

The pipeline begins running again during the multicycle instruction's last clock of operation in the EX stage.

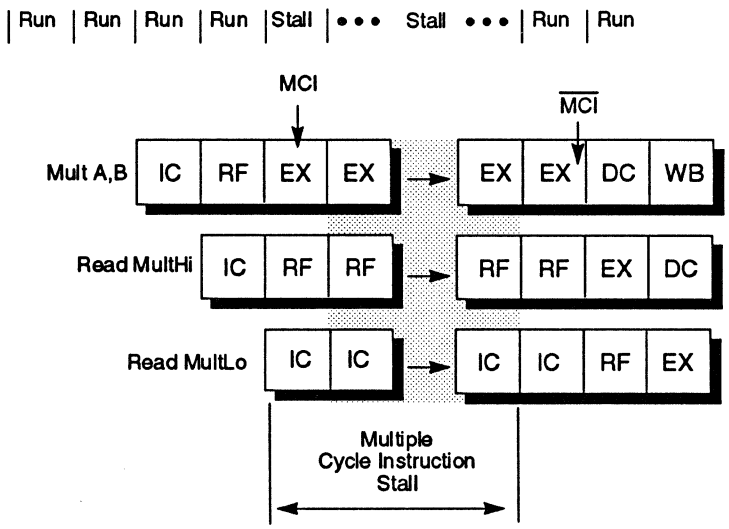


Figure 2-15 Example of a Multicycle Instruction Interlock

Load Interlock (LDI)

See Figure 3-17. A pipeline stall due to a Load Interlock occurs when data fetched by a load instruction is required by the next immediate instruction. The pipeline stalls when the load-use instruction (the instruction using the load data), enters the EX stage.

The pipeline begins running again when the clock after the target of the load is read from the data cache (in the DC stage of the "Load B" instruction below).

The Load Interlock is normally only active for one PClock cycle when the load instruction is in the DC stage and the load-use instruction is in the EX stage. The data returned from the data cache at the end of the DC stage is input into the EX stage, using the bypass multiplexers.

If the data cache misses, the Data Cache Busy interlock extends the stall until the data cache has been updated with the missing data. The LDI is still active during this time and extends the stall one clock beyond the Data Cache Interlock while the data is bypassed from the data cache into the EX stage. This case is illustrated in Figure 3-17.

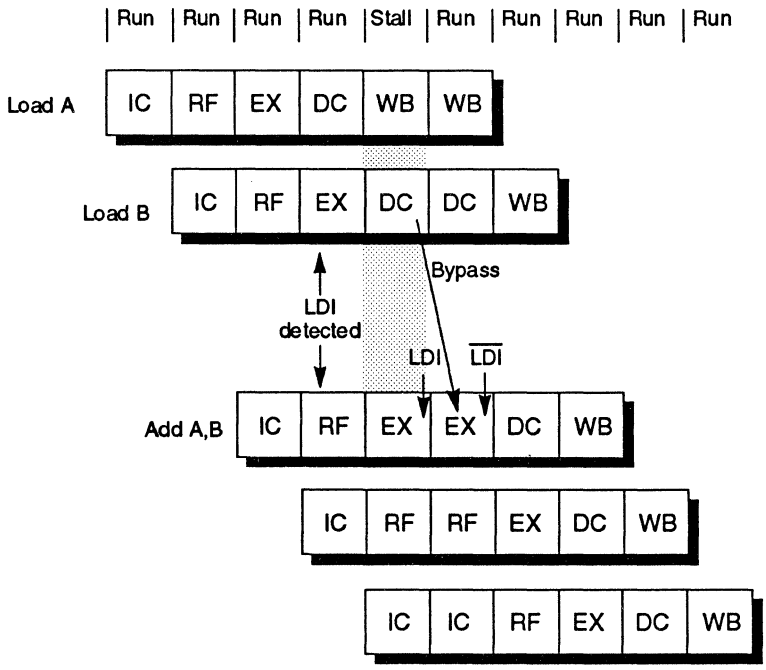


Figure 2-16 Example of a Load Interlock

Coprocessor 2 Interlock (CPI)

A pipeline stall due to a Coprocessor 2 Interlock occurs when a Coprocessor 2 instruction with an execution latency of more than one clock enters the EX stage. The pipeline controller informs CP2 that it may proceed in servicing the stall.

The pipeline begins running again during the CP2 instruction's last clock of operation in the EX stage.

Data Cache Miss (DCM)

See Figure 3-17. A pipeline stall due to a data cache miss occurs if the data cache look-up taking place in the DC stage results in a cache miss. This stall occurs for cache misses on both loads and stores. A DCM stall is really only active during the cycle in which the data cache is looked up and missed. A DCB stall is active while the new cache line is being read in.

The pipeline begins running again when the requested data word is available from the cache.

Data Cache Busy (DCB)

A pipeline stall due to the data cache being busy can occur in the following two situations:

- If the instruction immediately after a store requires use of the data cache then the pipeline is stalled in its DC stage while the store writes the data to the cache during its WB stage. On a cache store hit the pipeline only stalls for one PClock while the data is written to the data cache. On a cache store miss the pipeline stalls with the store in the DC stage until the cache line has been updated. Once the line has been updated, the pipeline restarts and moves the store instruction into the WB stage. If the instruction following the "store" (i.e. the instruction currently in the DC stage) also requires access to the data cache, the pipeline will then stall for one additional cycle while the store data is being written to the cache.
- When a miss occurs on a load, the data cache signals it is busy while it fetches the critical data word from external memory. See Figure 3-17.

The pipeline begins running again on a load when the critical data word is available from the data cache.

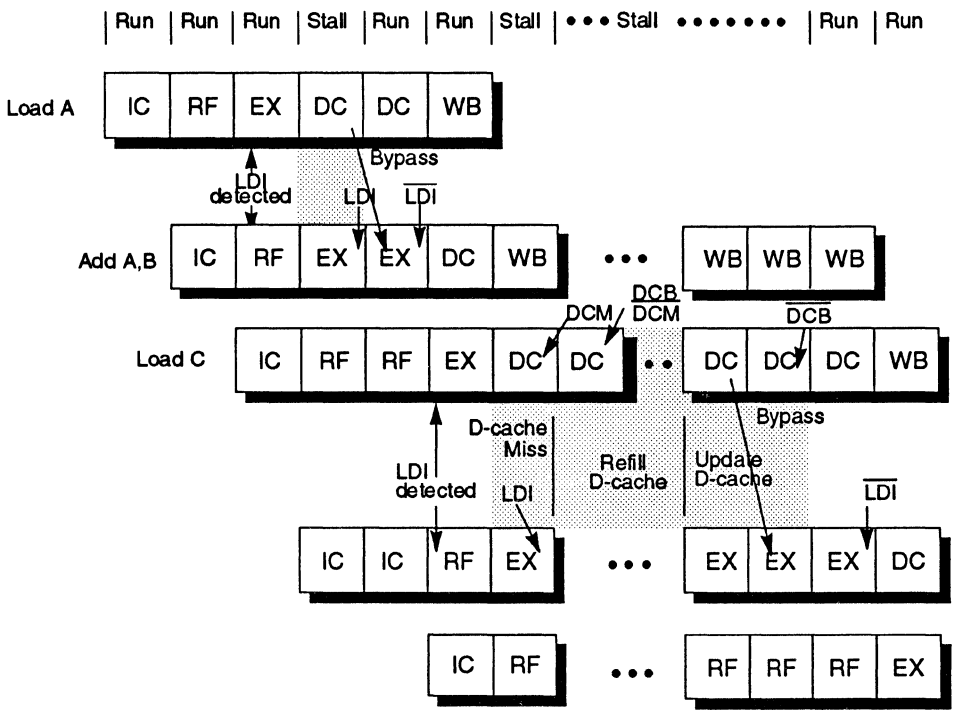


Figure 2-17 Example of a Load Interlock and a Data Cache Miss followed by a Load Interlock

CACHE Operation (COp)

A pipeline stall due to a CACHE operation can occur in the following two situations:

- When an instruction cache operation instruction enters the DC stage, the pipeline stalls while the instruction cache operation is serviced. The pipeline begins running again when the instruction cache operation is complete, allowing the instruction fetch to proceed.
- If a data cache operation instruction requiring more than one clock operation on the data cache enters the DC stage, the pipeline stalls.

Coprocessor 0 Bypass Interlock (CP0I)

A pipeline stall due to a CP0 Bypass Interlock occurs when an instruction which caused an exception reaches the WB stage and the subsequent instruction in the DC stage requests a read of any CP0 register.

This interlock causes a pipeline stall for one pcycle to allow the CP0 register to be written in WB before allowing any CP0 register to be read in the DC stage.

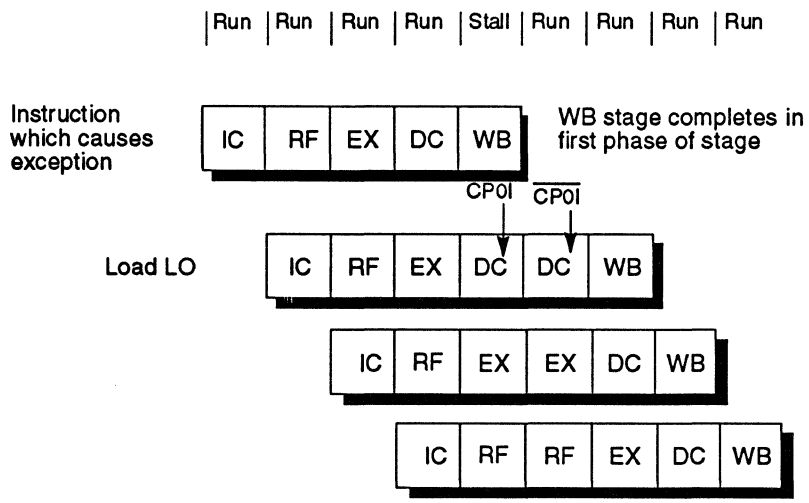


Figure 2-18 Example of a Coprocessor 0 Bypass Interlock (CP0I)

2.7 Pipeline Exceptions

When a pipeline exception condition occurs, the pipeline stalls for 2 PCycles and the instruction causing the exception as well as all those that follow it in the pipeline are aborted. Accordingly, any stall conditions and any later exception conditions from any aborted instruction are inhibited; there is no benefit in servicing stalls for an aborted instruction.

After aborting the instructions, a new instruction stream begins, starting execution at a predefined exception vector. System Control Coprocessor (CP0) registers are loaded with information that identifies the type of exception as well as auxiliary information such as the virtual address at which translation exceptions occur.

Exception conditions for each pipeline stage are shown in Figure 2-12 and described in Table 2-2.

Exceptions can split into two groups:

- those that occur independently of instruction execution (Reset, NMI, and interrupt exceptions)
- those exceptions that result from the execution of a particular instruction (an instruction-dependent exception). This category includes all other exceptions.

Exceptions are logically precise.

Instruction-Independent Exceptions (Reset, NMI, and Interrupt)

Reset, NMI and interrupt exceptions are identified and processed as follows:

- Reset exception has the highest priority of all the possible exceptions; when a Reset exception is asserted, instructions in all pipeline stages except the WB are aborted regardless of any interlocks or other exceptions that may be active.
- NMI and interrupt exception requests are only examined if the previous PCycle was a run cycle. When an NMI or interrupt exception occurs, all pipeline stages except the WB are aborted.

Instruction-Dependent Exceptions

Prioritizing between instruction-dependent exceptions and interlocks is made according to these rules:

1. an exception request from a particular pipeline stage is only processed if no stall condition from a later pipeline stage is active.
2. an exception request from a later pipeline stage always has a higher priority than an exception from an earlier pipeline stage.
3. an exception request from a pipeline stage always has higher priority than any stall request from the same or earlier pipeline stages.

Interactions Between Interlocks and Exceptions

Interlock and exception interaction between the EX and RF stages is relatively simple, even though both of these stages can continue processing during a pipeline stall. An EX exception can only occur when an instruction first enters that pipeline stage, therefore an EX exception takes priority over a stall request from the RF stage since the RF interlock has not yet begun to be resolved. If the EX exception is stalled by a DC interlock, it still takes priority over the RF stall request since the RF interlock cannot be resolved during a DC stall.

When an instruction cache miss and a multicycle instruction interlock are asserted simultaneously, both the RF and EX stages can resolve their respective interlocks. The floating-point operation may detect an exception before the ICB stall is deasserted, but the floating-point exception is not examined until the DC stage. Therefore the exception condition is held in EX until the RF interlock is resolved, eliminating the associated stall.

Once the floating-point instruction enters the DC stage, the exception is examined. If the RF interlock is asserted at the same time as an exception from the EX pipeline stage then the EX exception takes priority since the instruction causing the interlock will be aborted and no request to external memory will occur.

The above scenario does not hold for interactions between the RF and DC stages since, in a case of simultaneous stall requests, the pipeline controller only allows the DC stage to proceed. This means the RF stall cannot be resolved until the DC stall is resolved, since RF and DC interlocks may require the same resource to resolve the interlocks, e.g. system interface, TLB, etc.

Exception and Interlock Priorities

The priority for processing exceptions and interlocks within the same clock cycle is listed below. Exception and interlock requests from the WB stage always have priority over exception and interlock requests from the DC stage. Exception and interlock requests from the DC stage always have priority over exception and interlock requests from the EX stage. EX-stage exception and interlock requests in turn always have priority over any exception and interlock requests from the RF stage. In the case of multiple exception requests from the same pipeline stage, the highest-priority exception is processed first. The priority of the instruction-dependent exceptions and interlocks are shown in the following sections, with the highest-priority (WB stage) exception listed first.

WB-Stage Interlock and Exception Priorities

CP0 Bypass interlock is the only exception or interlock processed in the WB pipeline stage.

DC-Stage Interlock and Exception Priorities

Following is a prioritized list of the exceptions and interlocks processed in the DC pipeline stage.

- Reset exception (highest)
- NMI exception
- Integer Overflow exception
- Trap exception
- Floating-Point exception
- Coprocessor 2 exception
- Data Address Error exception
- Data TLB Refill exception
- Data TLB Invalid exception
- Data TLB Modified exception
- Watch exception
- Interrupt exception
- Data Cache Miss interlock
- Data Cache Busy interlock
- CACHE Op interlock
- Data Bus Error exception

EX-Stage Interlock and Exception Priorities

Following is a prioritized list of the exceptions and interlocks processed in the EX pipeline stage.

- System Call exception
- Breakpoint exception
- Coprocessor Unusable exception
- Reserved Instruction exception
- Load Data interlock
- Multi-cycle Instruction interlock
- Coprocessor 2 interlock

RF-Stage Interlock and Exception Priorities

Following is a prioritized list of the exceptions and interlocks processed in the RF pipeline stage.

- Instruction Address Error exception
- Instruction TLB Refill exception
- Instruction TLB Invalid exception
- Instruction TLB Miss interlock
- Instruction Cache Busy interlock
- Instruction Bus Error exception

If an Instruction Bus Error exception occurs during a cache refill, while an Instruction Cache Busy interlock is active, the instruction cache only signals the exception to the pipeline controller after the cache refill is complete, and therefore no stall is active.

Individual exceptions are described in detail in Chapter 5.

Bypassing

In some cases, data and conditions produced in the EX, DC and WB stages of the pipeline are made available to the EX stage (only) through the bypass datapath.

Operand bypass allows an instruction in the EX stage to continue without having to wait for data or conditions to be written to the register file at the end of the WB stage. Instead, the Bypass Control Unit is responsible for ensuring data and conditions from later pipeline stages are available at the appropriate time for instructions earlier in the pipeline.

The Bypass Control Unit is also responsible for controlling the source and destination register addresses supplied to the register file.

2.8 Code Compatibility

The V_R4300 can execute any V_R4000 code sequence, but the reverse may not necessarily be true. Standard MIPS compilers produce code which will run on both. When hand-coding assembly code, it is strongly advised to maintain compatibility with the V_R4000. For more information, refer to the V_R4000 User's Manual.

2.9 Flush Buffer

The V_R4300 processor contains an on-chip flush buffer. This buffer is used as temporary data storage for outgoing data and is organized as a 4-deep FIFO; it can buffer four addresses with four doublewords of data. For uncached write operations, the flush buffer can accept any combination of single or doubleword data until it is full, with each write occupying one entry in the buffer. For data cache block write operations, the flush buffer accepts two doublewords with one address, occupying two entries in the buffer. It is able to take two block references at a time. Instruction cache block writes use four doublewords with one address, and occupy the entire flush buffer. The flush buffer is able to take one read memory reference at a time.

The format of the flush buffer is shown in Figure 2-19 below.

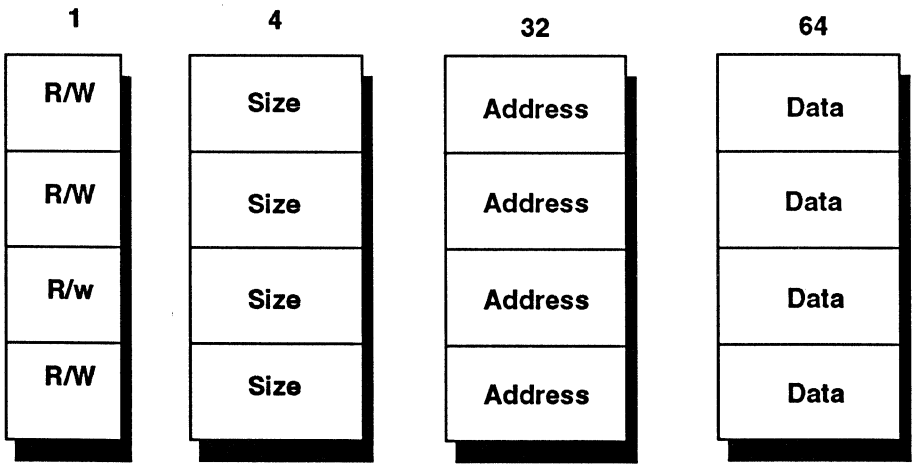


Figure 2-19 Flush Buffer Format

The flush buffer holds, for each doubleword of *data*:

- a 32-bit *physical address*
- a 4-bit *Size* field
- a single *read/write* bit

The flush buffer mechanism is inaccessible to software.

During an uncached store operation, data is held in this buffer until it can be retrieved by the external interface. The processor pipeline continues to execute while data is stored in the flush buffer.

During either a load miss or a store miss to a cache line in the dirty state, a read request is sent to the external interrface for the mising cache line; the dirty data is stored in this buffer until the requested data is returned from the external interface. The processor pipeline continues to run while the flush buffer waits (for a response from the external interface) to empty its contents to the external interface/memory.

If the flush buffer is full and the processor attempts a load or a store which requires external resources, the processor pipeline stalls until the flush buffer is emptied.

Memory Management

3

The MIPS V_R4300 processor provides a full-featured memory management unit (MMU) which uses an on-chip translation lookaside buffer (TLB) to translate virtual addresses into physical addresses.

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

3.1 Translation Lookaside Buffer (TLB)

Mapped virtual addresses are translated into physical addresses using an on-chip TLB.† The TLB is a fully-associative memory that holds 32 entries, which provide mapping to 32 odd/even page pairs (64 pages). When address mapping is indicated, each TLB entry is checked simultaneously for a match with the virtual address that is extended with an ASID stored in the *EntryHi* register.

The address is mapped to a page size ranging from 4 Kbytes to 16 Mbytes, in multiples of four; that is, 4K, 16K, 64K, 256K, 1M, 4M and 16M.

Hits and Misses

If there is a virtual address match, or "hit," in the TLB, the physical page number is extracted from the TLB and concatenated with the offset to form the physical address (see Figure 3-1).

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

Multiple Matches

If more than one entry in the TLB matches the virtual address being translated, the operation is undefined.

If one TLB entry matches, the physical address and access control bits (*C*, *D*, and *V*) are retrieved; otherwise a TLB refill exception occurs. If the access control bits (*D* and *V*) indicate that the access is not valid, a TLB modification or TLB invalid exception occurs.

† There are virtual-to-physical address translations that occur outside of the TLB. For example, addresses in *the kseg0* and *kseg1* spaces are unmapped translations. In these spaces the physical address is derived by subtracting the base address of the space from the virtual address.

3.2 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 in the TLB.

Virtual Address Space

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

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

Figure 3-1 shows the translation of a virtual address into a physical address.

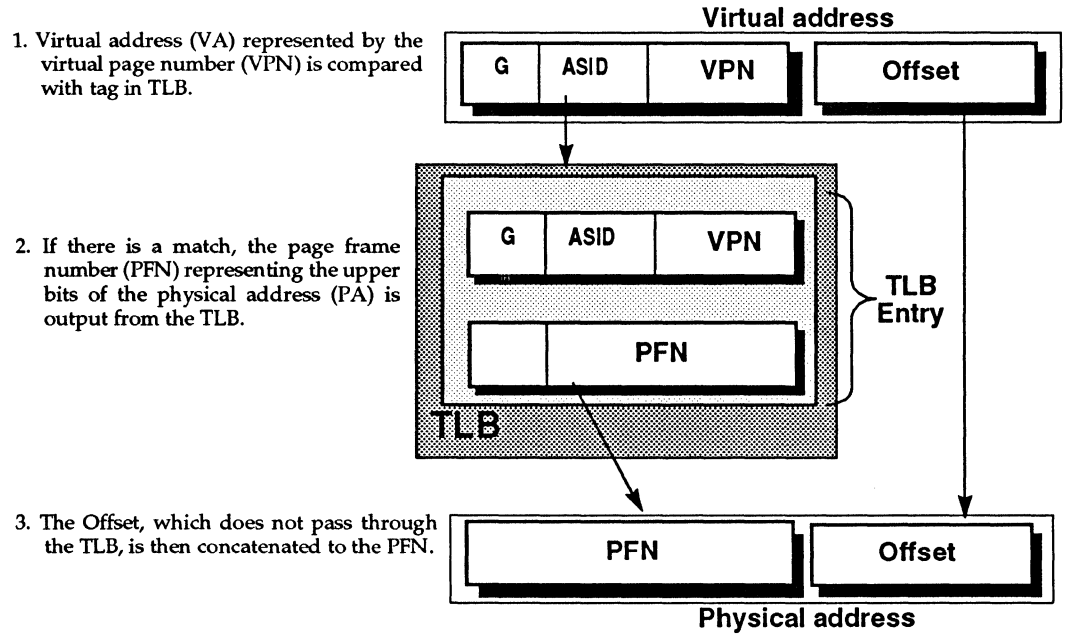


Figure 3-1 Overview of a Virtual-to-Physical Address Translation

† Figure 3-8 shows the 32-bit and 64-bit versions of the processor TLB entry.

As shown in Figures 3-2 and 3-3, the virtual address is extended with an 8-bit address space identifier (ASID), which reduces the frequency of TLB flushing when switching contexts. This 8-bit ASID is in the CP0 *EntryHi* register, described later in this chapter. The *Global* bit (*G*) is in the *EntryLo0* and *EntryLo1* registers, described later in this chapter.

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

Converting a virtual address to a physical address begins by comparing the virtual address from the processor with the virtual addresses in the TLB; there is a match when the virtual page number (VPN) of the address is the same as the VPN field of the entry, and either:

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

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

If there is a virtual address match in the TLB, the physical address is output from the TLB and concatenated with the *Offset*, which represents an address within the page frame space. The *Offset* does not pass through the TLB.

Virtual-to-physical translation is described in greater detail throughout the remainder of this chapter; Figure 3-19 is a flow diagram of the process shown at the end of this chapter.

The next two sections describe the 32-bit and 64-bit address translations.

32-bit Mode Address Translation

Figure 3-2 shows the virtual-to-physical-address translation of a 32-bit mode address. This figure illustrates two of the possible page sizes: a 4-Kbyte page (12 bits) and a 16-Mbyte page (24 bits).

- The top portion of Figure 3-2 shows a virtual address with a 12-bit, or 4-Kbyte, page size, labelled *Offset*. The remaining 20 bits of the address represent the VPN, and index the 1M-entry page table.
- The bottom portion of Figure 3-2 shows a virtual address with a 24-bit, or 16-Mbyte, page size, labelled *Offset*. The remaining 8 bits of the address represent the VPN, and index the 256-entry page table.

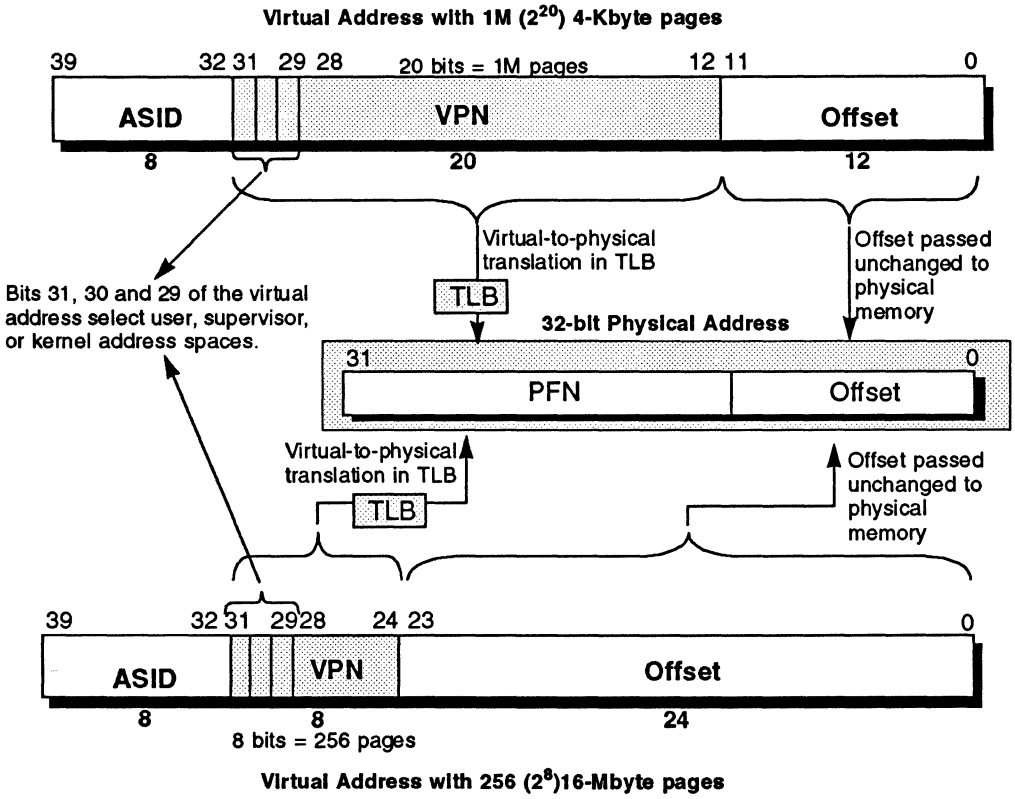


Figure 3-2 32-bit Mode Virtual Address Translation

64-bit Mode Address Translation

Figure 3-3 shows the virtual-to-physical-address translation of a 64-bit mode address. This figure illustrates two of the possible page sizes: a 4-Kbyte page (12 bits) and a 16-Mbyte page (24 bits).

- The top portion of Figure 3-3 shows a virtual address with a 12-bit, or 4-Kbyte, page size, labelled *Offset*. The remaining 28 bits of the address represent the VPN, and index the 256M-entry page table.
- The bottom portion of Figure 3-3 shows a virtual address with a 24-bit, or 16-Mbyte, page size, labelled *Offset*. The remaining 16 bits of the address represent the VPN, and index the 64K-entry page table.

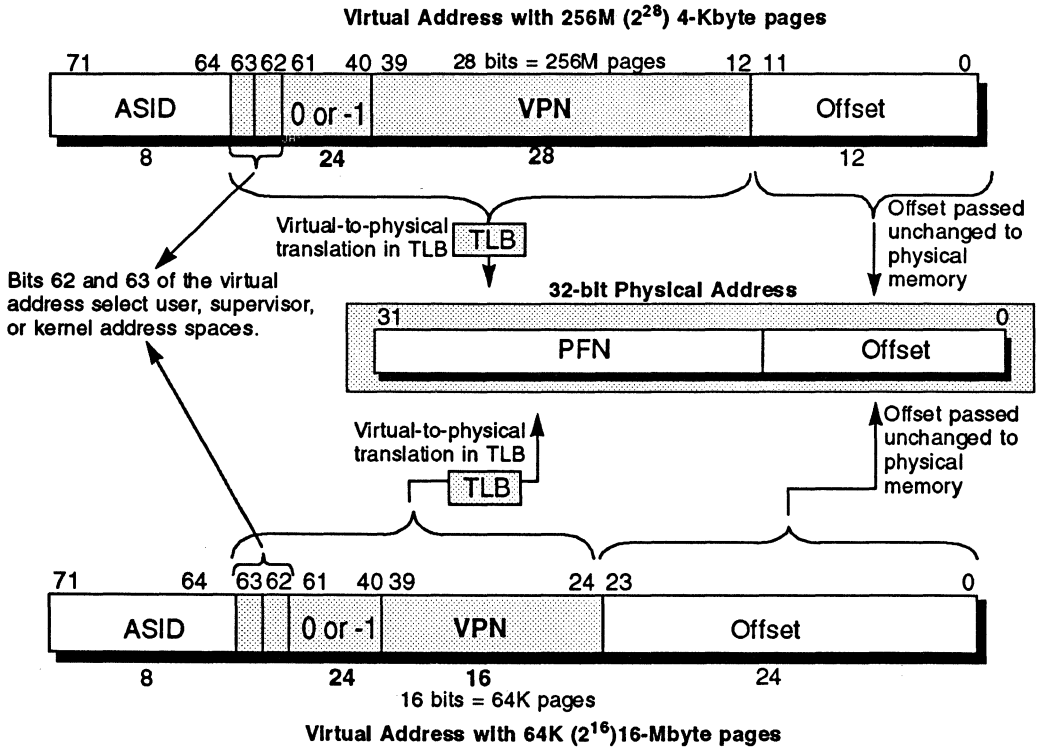


Figure 3-3 64-bit Mode Virtual Address Translation

Operating Modes

The processor has three operating modes that function in both 32- and 64-bit operations:

- User mode
- Supervisor mode
- Kernel mode

These modes are described in the next three sections.

User Mode Operations

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

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

Figure 3-4 shows User mode virtual address space.

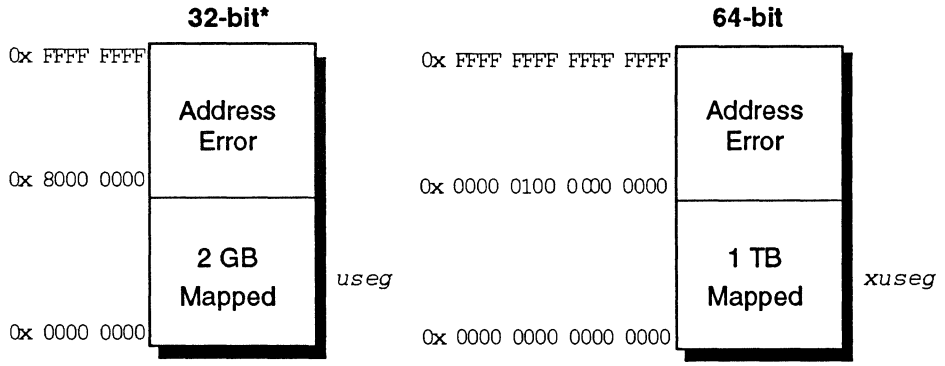


Figure 3-4 User Mode Virtual Address Space

*NOTE: The V_R4300 uses 64-bit addresses internally. When the processor is running in Kernel mode, it initializes registers before switching modes, and saves (or restores, whichever is appropriate) register values on context switches. In 32-bit mode, a valid address must be a 32-bit signed number, where bits 63:32 = bit 31. In normal operation it is not possible for a 32-bit User-mode program to produce invalid addresses. However, although it would be an error, it *is* possible for a Kernel-mode program to erroneously place a value that is not a 32-bit signed number into a 64-bit register, in which case the User-mode program generates an invalid address.

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

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

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

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

- when *UX* = 0, 32-bit *useg* space is selected and TLB misses are handled by the 32-bit TLB refill exception handler
- when *UX* = 1, 64-bit *xuseg* space is selected and TLB misses are handled by the 64-bit XTLB refill exception handler

Table 3-1 lists the characteristics of the two user mode segments, *useg* and *xuseg*.

Table 3-1 32-bit and 64-bit User Mode Segments

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

† The cached (*C*) field in a TLB entry determines whether the reference is cached; see Figure 3-8.

32-bit User Mode (*useg*)

In User mode, when *UX* = 0 in the *Status* register, User mode addressing is compatible with the 32-bit addressing model shown in Figure 3-4, and a 2-Gbyte user address space is available, labelled *useg*.

All valid User mode virtual addresses have their most-significant bit cleared to 0; any attempt to reference an address with the most-significant bit set while in User mode causes an Address Error exception.

The system maps all references to *useg* through the TLB, and bit settings within the TLB entry for the page determine the cacheability of a reference.

64-bit User Mode (*xuseg*)

In User mode, when *UX* =1 in the *Status* register, User mode addressing is extended to the 64-bit model shown in Figure 3-4. In 64-bit User mode, the processor provides a single, uniform address space of 2^{40} bytes, labelled *xuseg*.

All valid User mode virtual addresses have bits 63:40 equal to 0; an attempt to reference an address with bits 63:40 not equal to 0 causes an Address Error exception.

Supervisor Mode Operations

Supervisor mode is designed for layered operating systems in which a true kernel runs in V_R4300 Kernel mode, and the rest of the operating system runs in Supervisor mode.

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

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

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

- when *SX* = 0, 32-bit supervisor space is selected and TLB misses are handled by the 32-bit TLB refill exception handler
- when *SX* = 1, 64-bit supervisor space is selected and TLB misses are handled by the 64-bit XTLB refill exception handler

Figure 3-5 shows Supervisor mode address mapping. Table 3-2 lists the characteristics of the supervisor mode segments; descriptions of the address spaces follow.

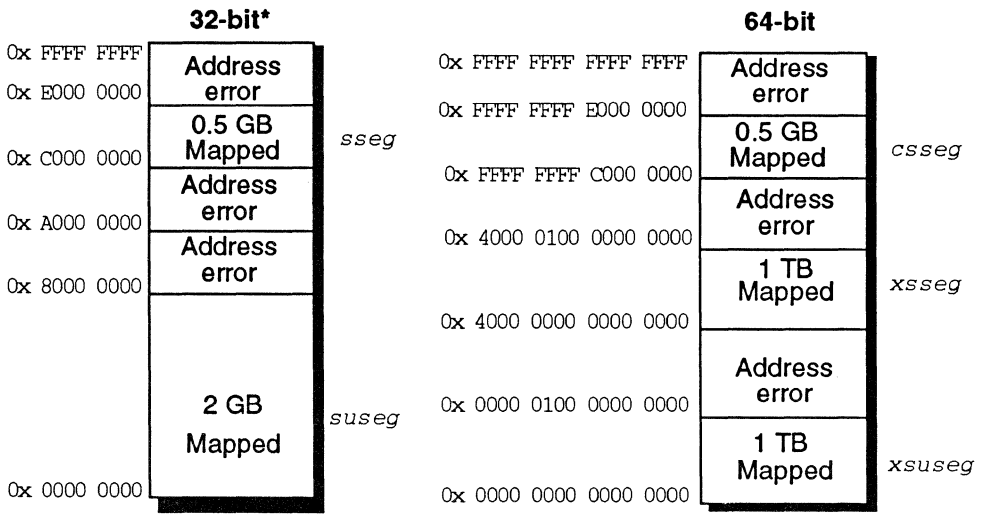


Figure 3-5 Supervisor Mode Address Space

*NOTE: The V_R4300 uses 64-bit addresses internally. In 32-bit mode, a valid address must be a 32-bit signed number, where bits 63:32 = bit 31. In normal operation it is not possible for a 32-bit Supervisor-mode program to create an invalid address through arithmetic operations. However 32-bit-mode Supervisor programs must not create addresses using *base register+offset* calculations that produce a 32-bit 2's-complement overflow; in specific, there are two prohibited cases:

- offset with bit 15 = 0 and base register with bit 31 = 0, but (*base register+offset*) bit 31 = 1
- offset with bit 15 = 1 and base register with bit 31 = 1, but (*base register+offset*) bit 31 = 0

Using this invalid address produces an undefined result.

Table 3-2 32-bit and 64-bit Supervisor Mode Segments

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

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

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

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

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

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

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

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

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

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

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

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

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

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

In Supervisor mode, when *SX* = 1 in the *Status* register and bits 63:62 of the virtual address are set to 11_2, the *csseg* separate supervisor virtual address space is selected. If bits 31:29 of the virtual address are set to 110_2, addressing of the *csseg* is compatible with addressing *sseg* in 32-bit mode. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

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

Kernel Mode Operations

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

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

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

- when *KX* = 0, 32-bit kernel space is selected and all TLB misses are handled by the 32-bit TLB refill exception handler
- when *KX* = 1, 64-bit kernel space is selected and all TLB misses are handled by the 64-bit XTLB refill exception handler

The processor enters Kernel mode whenever an exception is detected and it remains in Kernel mode until an Exception Return (ERET) instruction is executed and results in ERL and/or EXL = 0. The ERET instruction restores the processor to the mode existing prior to the exception.

Kernel mode virtual address space is divided into regions differentiated by the high-order bits of the virtual address, as shown in Figure 3-6. Table 3-3 lists the characteristics of the 32-bit kernel mode segments, and Table 3-4 lists the characteristics of the 64-bit kernel mode segments.

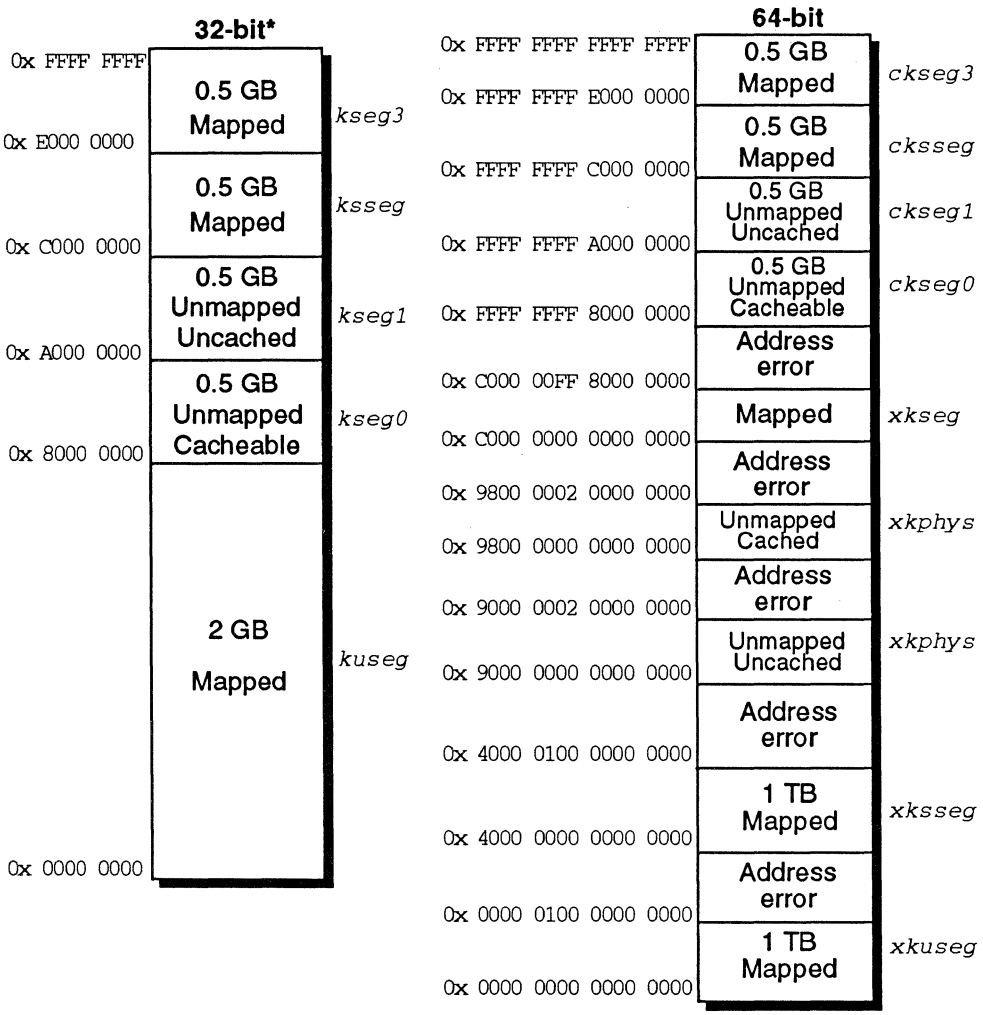


Figure 3-6 Kernel Mode Address Space

*NOTE: The V_R4300 uses 64-bit addresses internally. In 32-bit mode, a valid address must be a 32-bit signed number, where bits 63:32 = bit 31; an invalid address produces an undefined result. In 32-bit mode, a Kernel-mode program may use 64-bit instructions, but must not create addresses using *base register+offset* calculations that produce a 32-bit 2's-complement overflow; in specific, there are two prohibited cases:

- offset with bit 15 = 0 and base register with bit 31 = 0, but (*base register+offset*) bit 31 = 1
- offset with bit 15 = 1 and base register with bit 31 = 1, but (*base register+offset*) bit 31 = 0

Table 3-3 32-bit Kernel Mode Segments

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

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

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

When ERL = 1 in the *Status* register, the user address region becomes a 2^{31}-byte unmapped (that is, mapped directly to physical addresses) uncached address space. See the Cache Error exception in Chapter 4 for more information.

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

In Kernel mode, when *KX* = 0 in the *Status* register and the most-significant three bits of the virtual address are 100_2, 32-bit *kseg0* virtual address space is selected; it is the 2^{29}-byte (512-Mbyte) kernel physical space.

References to *kseg0* are not mapped through the TLB; the physical address selected is defined by subtracting 0x8000 0000 from the virtual address.

The *K0* field of the *Config* register, described in this chapter, controls cacheability and coherency.

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

In Kernel mode, when *KX* = 0 in the *Status* register and the most-significant three bits of the 32-bit virtual address are 101_2, 32-bit *kseg1* virtual address space is selected; it is the 2^{29}-byte (512-Mbyte) kernel physical space.

References to *kseg1* are not mapped through the TLB; the physical address selected is defined by subtracting 0xA000 0000 from the virtual address.

Caches are disabled for accesses to these addresses, and physical memory (or memory-mapped I/O device registers) are accessed directly.

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

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

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

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

Table 3-4 64-bit Kernel Mode Segments

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

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

In Kernel mode, when *KX* = 1 in the *Status* register and bits 63:62 of the 64-bit virtual address are 00_2, the *xkuseg* virtual address space is selected; it covers the current user address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

When ERL = 1 in the *Status* register, the user address region becomes a 2^{31}-byte unmapped (that is, mapped directly to physical addresses) uncached address space. See the Cache Error exception in Chapter 4 for more information.

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

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

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

In Kernel mode, when *KX* = 1 in the *Status* register and bits 63:62 of the 64-bit virtual address are 10_2, one of the two unmapped *xkphys* address spaces are selected, either cached or uncached. Accesses with address bits 58:32 not equal to 0 cause an address error.

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

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

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

64-bit Kernel Mode, Compatibility Spaces (*ckseg1:0, cksseg, ckseg3*)

In Kernel mode, when *KX* = 1 in the *Status* register, bits 63:62 of the 64-bit virtual address are 11_2, and bits 61:31 of the virtual address equal –1, the lower two bytes of address, as shown in Figure 3-6, select one of the following 512-Mbyte compatibility spaces.

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

3.3 System Control Coprocessor

The System Control Coprocessor (CP0) is implemented as an integral part of the CPU, and supports memory management, address translation, exception handling, and other privileged operations. CP0 contains the registers shown in Figure 3-7 plus a 32-entry TLB. The sections that follow describe how the processor uses each of the memory management-related registers†.

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

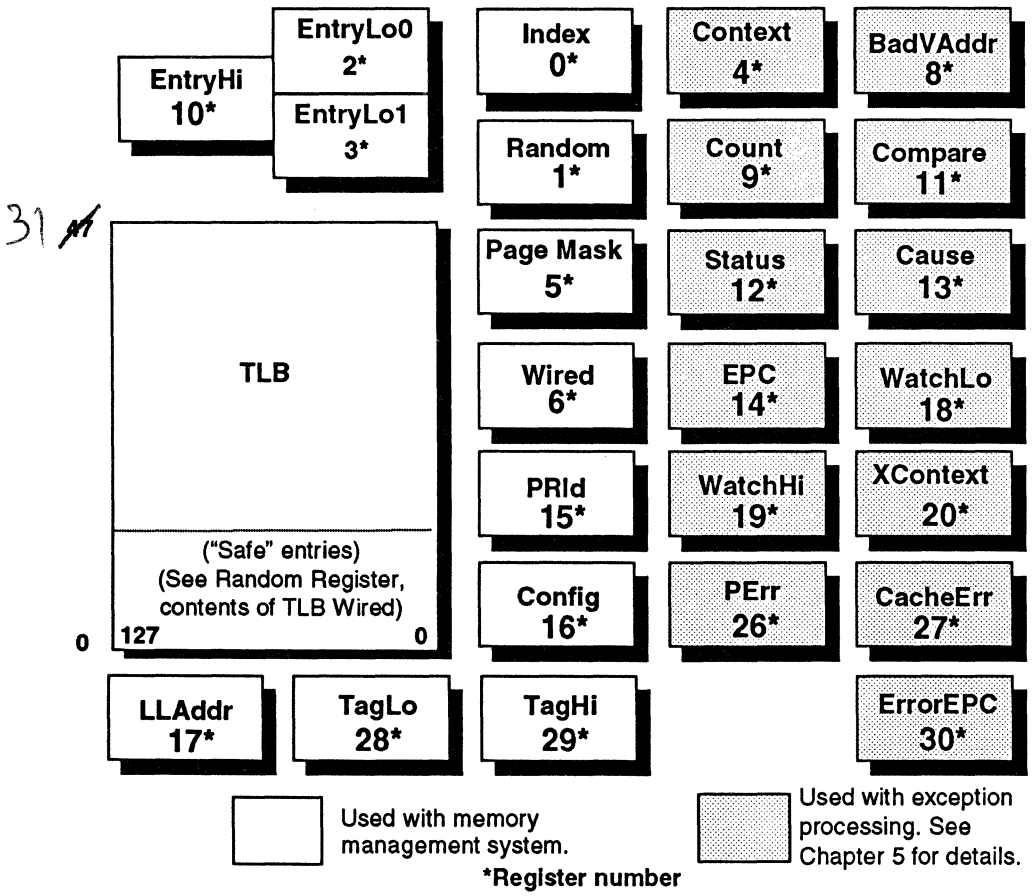


Figure 3-7 CP0 Registers and the TLB

† For a description of CP0 data dependencies and hazards, please see Appendix B.

Format of a TLB Entry

Figure 3-8 shows the TLB entry formats for both 32- and 64-bit modes. Each field of an entry has a corresponding field in the *EntryHi*, *EntryLo0*, *EntryLo1*, or *PageMask* registers, as shown in Figures 3-9 and 3-10; for example the *Mask* field of the TLB entry is also held in the *PageMask* register.

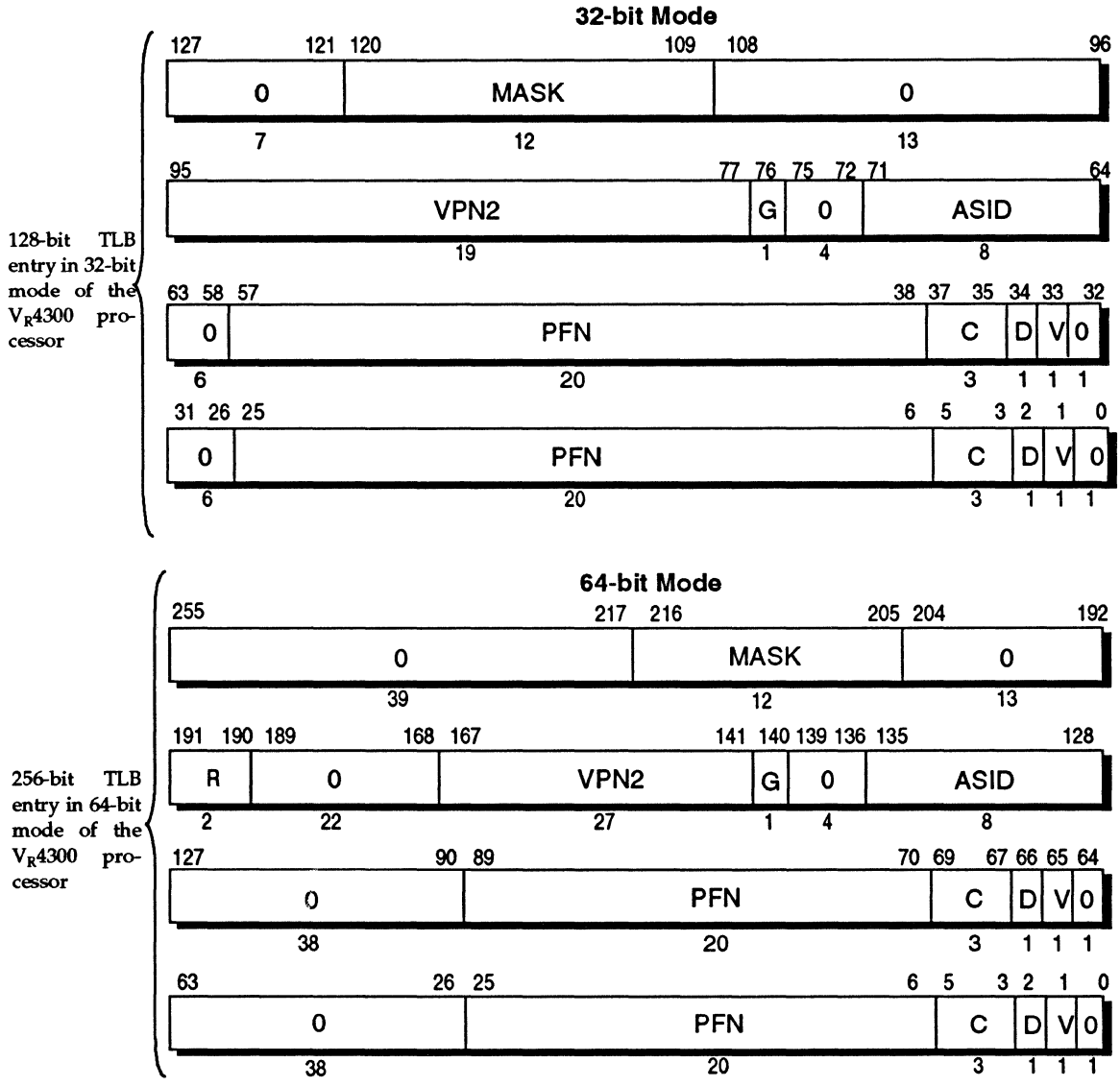
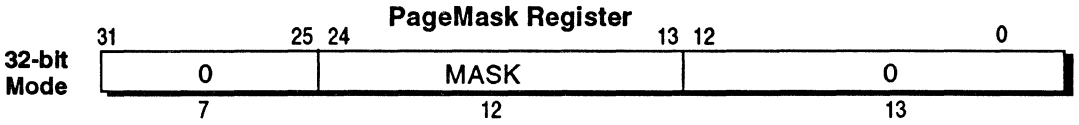
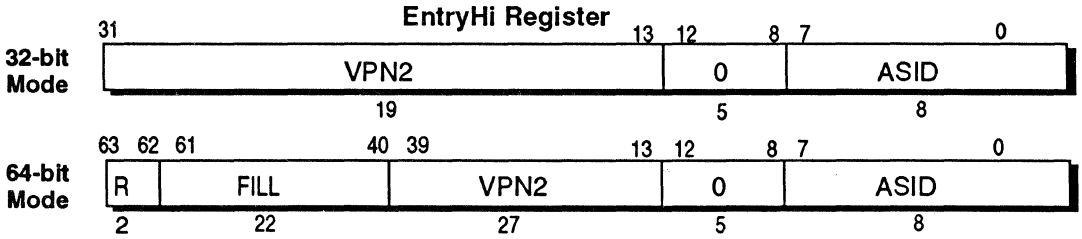


Figure 3-8 Format of a TLB Entry

The format of the *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers are nearly the same as the TLB entry. The one exception is the *Global* field (*G* bit), which is used in the TLB, but is reserved in the *EntryHi* register. Figures 3-9 and 3-10 describe the TLB entry fields shown in Figure 3-8.

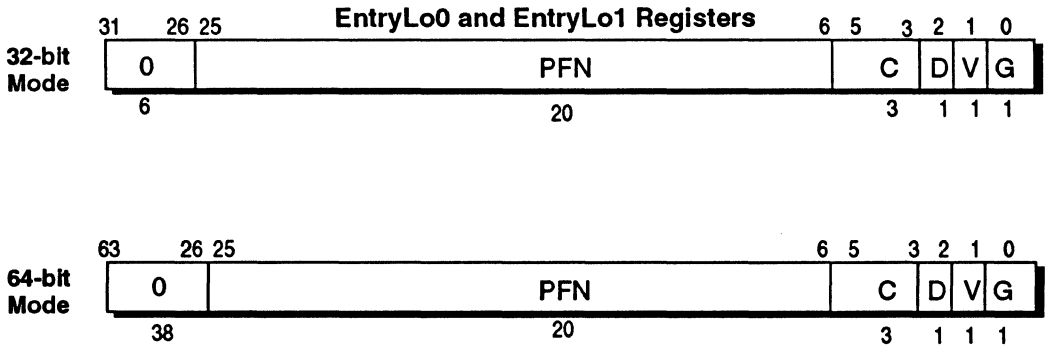


Mask.... Page comparison mask.
0..... Reserved. Must be written as zeroes, and returns zeroes when read.



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

Figure 3-9 Fields of the PageMask and EntryHi Registers



PFN..... Page frame number; the upper bits of the physical address.
C..... Specifies the TLB page attribute; see Table 3-5.
D..... Dirty. If this bit is set, the page is marked as dirty and, therefore, writable. This bit is actually a write-protect bit that software can use to prevent alteration of data.
V..... Valid. If this bit is set, it indicates that the TLB entry is valid; otherwise, a TLBL or TLBS miss occurs.
G..... Global. If this bit is set in both Lo0 and Lo1, then the processor ignores the ASID during TLB lookup.
0..... Must be written as zeroes, and returns zeroes when read.
Rsvd..... Reserved. Must be written as zeroes, and returns zeroes when read.

Figure 3-10 Fields of the EntryLo0 and EntryLo1 Registers

The TLB page coherency attribute (C) bits specify whether references to the page should be cached; if cached, the algorithm selects between cached and uncached page attribute. Table 3-5 shows the cache algorithms selected by the C bits. Although technically undefined, any value other than 1 or 2 defaults to *cached*.

Table 3-5 TLB Cache Algorithm (C) Bit Values

C(5:3) Value	Page Attribute
0	*Cached*
1	*Cached*
2	Uncached
3	Cached
4	*Cached*
5	*Cached*
6	*Cached*
7	*Cached*

CP0 Registers

The following sections describe the CP0 registers, shown in Figure 3-7, that are assigned specifically as a software interface with memory management (each register is followed by its register number in parentheses).

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

Index Register (0)

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

The *Index* register also specifies the TLB entry affected by TLB Read (TLBR) or TLB Write Index (TLBWI) instructions.

Although the *Index* register *Index* field is six bits wide, only the five least-significant bits (4:0) are used in TLB operations, since the V_R4300 TLB has 32 entries. Bit 5 is readable and writeable by software, but is ignored during TLB operations. The *Index* field should not contain any value greater than 31.

Figure 3-11 shows the format of the *Index* register; Table 3-6 describes the *Index* register fields.

Index Register

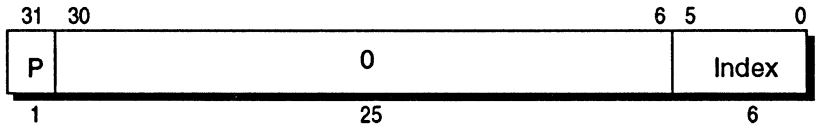


Figure 3-11 Index Register

Table 3-6 Index Register Field Descriptions

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

Random Register (1)

The *Random* register is a read-only register of which six bits index an entry in the TLB. Although the *Random* field is six bits wide, only the five least-significant bits (4:0) are used in TLB operations, since the V_R4300 TLB has 32 entries. Bit 5 is readable and writeable by software, but is ignored during TLB operations.

This register decrements as each instruction executes, and its values range between an upper and a lower bound, as follows:

- A lower bound is set by the number of TLB entries reserved for exclusive use by the operating system (the contents of the *Wired* register).
- An upper bound is set by the total number of TLB entries-1 (31).

The *Random* register specifies the entry in the TLB that is affected by the TLB Write Random instruction. The register does not need to be read for this purpose; however, the register is readable to verify proper operation of the processor.

To simplify testing, the *Random* register is set to the value of the upper bound upon Cold Reset. This register is also set to the upper bound when the *Wired* register is written.

Figure 3-12 shows the format of the *Random* register; Table 3-7 describes the *Random* register fields.

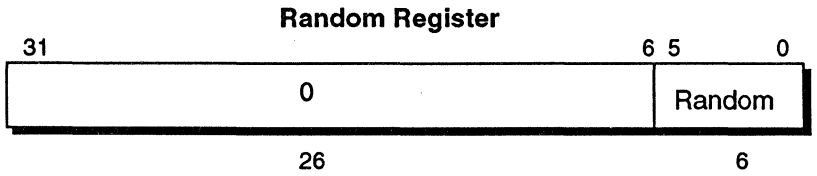


Figure 3-12 Random Register

Table 3-7 Random Register Field Descriptions

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

EntryLo0 (2), and EntryLo1 (3) Registers

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

- *EntryLo0* is used for even virtual pages.
- *EntryLo1* is used for odd virtual pages.

The *EntryLo0* and *EntryLo1* registers are read/write registers. They hold the physical page frame number (PFN) of the TLB entry for even and odd pages, respectively, when performing TLB read and write operations. Figure 3-10 shows the format of these registers.

PageMask Register (5)

The *PageMask* register is a read/write register used for reading from or writing to the TLB; it holds a comparison mask that sets the page size for each TLB entry, as shown in Table 3-8. Page sizes range from 4 Kbytes to 16 Mbytes, in increments of four. The format of the *PageMask* register is shown in Figure 3-9.

TLB read and write operations use this register as either a source or a destination; when virtual addresses are presented for translation into physical address, the corresponding bits in the TLB identify which virtual address bits among bits 24:13 are used in the comparison. When the *Mask* field is not one of the values shown in Table 3-8, the operation of the TLB is undefined.

Table 3-8 Mask Field Values for Page Sizes

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

Wired Register (6)

The *Wired* register is a read/write register that specifies the boundary between the *wired* and *random* entries of the TLB as shown in Figure 3-13. Wired entries are fixed, nonreplaceable entries, which cannot be overwritten by a TLBWR (TLB Write Random) operation. They can, however, be overwritten by a TLBWI (TLB Write Indexed) instruction. Random entries can be overwritten.

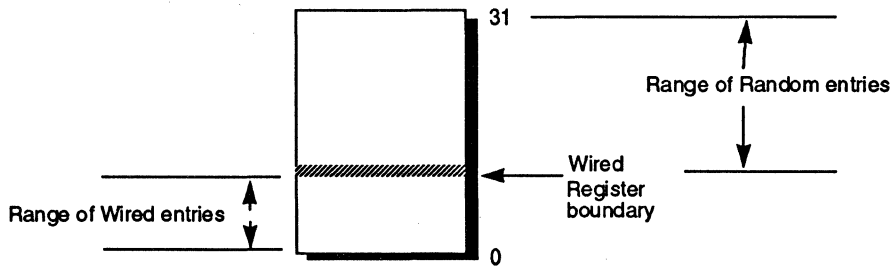


Figure 3-13 Wired Register Boundary

Although the *Wired* field is six bits wide, only the five least-significant bits (4:0) are used in TLB operations, since the V_R4300 TLB has 32 entries. Bit 5 is readable and writeable by software, but is ignored during TLB operations. The TLB entry specified by the *Wired* field is not considered to be in the *Wired* portion of the TLB; it is considered to be in the *Random* portion. The *Wired* register is set to 0 upon Cold Reset. Writing this register also sets the *Random* register to the value of its upper bound of 31 (see *Random* register, above). Figure 3-14 shows the format of the *Wired* register; Table 3-9 describes the register fields.

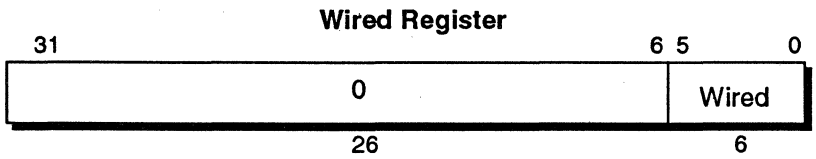


Figure 3-14 Wired Register

Table 3-9 Wired Register Field Descriptions

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

EntryHi Register (CP0 Register 10)

The *EntryHi* register holds the high-order bits of a TLB entry for TLB read and write operations.

The *EntryHi* register is accessed by the TLB Probe, TLB Write Random, TLB Write Indexed, and TLB Read Indexed instructions.

Figure 3-9 shows the format of this register.

When either a TLB refill, TLB invalid, or TLB modified exception occurs, the *EntryHi* register is loaded with the virtual page number (VPN2) and the ASID of the virtual address that did not have a matching TLB entry. (See Chapter 5 for more information about these exceptions.)

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 3-15 shows the format of the *PRId* register; Table 3-10 describes the *PRId* register fields.

PRId Register

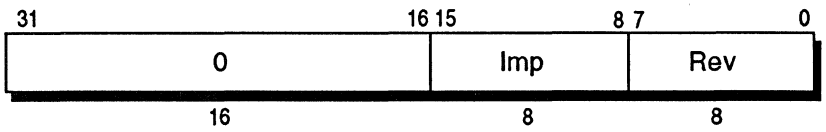


Figure 3-15 Processor Revision Identifier Register Format

Table 3-10 PRId Register Fields

Field	Description
Imp	Implementation number (0x0B for the V_R4300)
Rev	Revision number
0	Reserved. Must be written as zeroes, and returns zeroes when read.

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 content of the high-order halfword (bits 31:16) of the register are reserved.

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

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

Config Register (16)

The *Config* register specifies various configuration options that are available for V_R4300. It is compatible with the V_R4000 Config register, but only a subset of the options on the V_R4000 are available on V_R4300. For that reason, there are many fields which are set to constant values.

The *EP* and *BE* fields are written to their default values by hardware during a Cold Reset; these fields are also readable and writable by software. The default values upon Cold Reset are as follows:

- *EP*=0000
- *BE*=1.

The *CU* and *K0* fields are readable and writable by software. There is no other mechanism for writing to these fields, and their values are undefined after Reset (either Cold or Warm).

The *EP* and *BE* fields change only during processor initialization (done in uncached space before any stores). If there are any changes to these values during normal operation, correct behavior of the processor is not guaranteed.

Figure 3-16 shows the *Config* register format, and Table 3-11 describes the *Config* register fields.

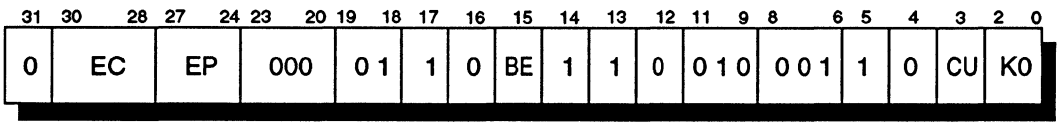


Figure 3-16 Config Register Format

Table 3-11 Config Register Fields

Field	Description
EC	System clock ratio; read-only. 110 → 1: 1 111 → 1.5: 1 000 → 2: 1 001 → 3: 1 All other patterns are undefined.
EP	Transmit data pattern (pattern for write-back data on SysAD port): 0000 → D (one doubleword every cycle) 0110 → DxxDxx (two doublewords every 6 cycles All other values are undefined
BE	BigEndianMem 0 → kernel and memory are little endian 1 → kernel and memory are big endian
CU	Reserved (software has read/write access)
K0	*kseg0* coherency algorithm (see *EntryLo0* and *EntryLo1* registers and the C field of Table 3-5).
1	Returns a 1 when read.
0	Returns a 0 when read.

Load Linked Address (LLAddr) Register (17)

The read/write *Load Linked Address* (*LLAddr*) register contains the physical address read by the most recent Load Linked instruction. This register is for diagnostic purposes only, and serves no function during normal operation. Figure 3-17 shows the format of the *LLAddr* register; *PAddr* represents bits of the physical address, PA(35:4).

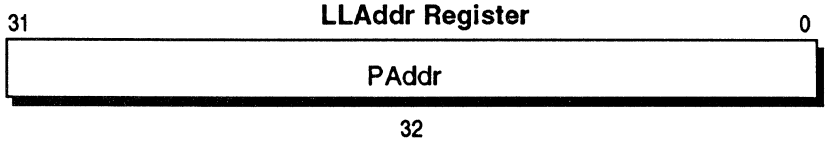


Figure 3-17 LLAddr Register Format

The most-significant-bit of an V_R4300 physical address is 31, **PA(31)**. A Load Linked instruction will therefore write 0s into bits 31:28 of the *LLAddr* register (bits **PA(35:32)** in the V_R4000). A software write can set all 32 bits of the *LLAddr* register to any value. This maintains software compatibility with V_R4000.

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

The *TagLo* and *TagHi* registers are 32-bit read/write registers that hold cache tag information for cache instructions. The *Tag* registers are written by the CACHE and MTC0 instructions. The *TagHi* register is not used in the V_R4300 implementation, and is reserved for future use.

Figure 3-18 shows the format of these registers for primary cache operations. Table 3-12 lists the field definitions of the *TagLo* and *TagHi* registers.

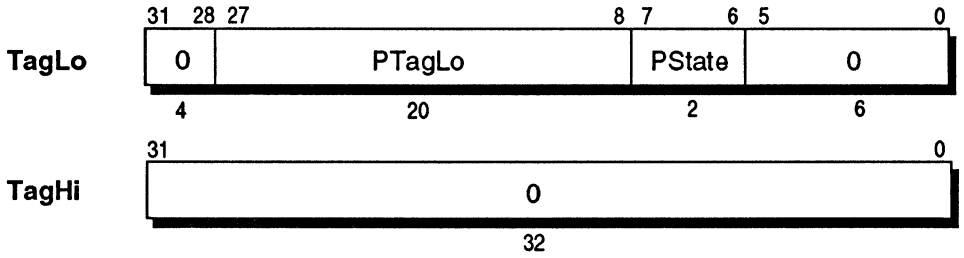


Figure 3-18 TagLo and TagHi Register Formats

Table 3-12 Cache Tag Register Fields

Field	Description
PTagLo	Physical address bits 31:12
PState	Specifies the primary cache state (11_2=Dirty, 00_2 = Invalid)
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Software can read and write all 32 bits of the *TagLo* register, using the move to/move from CP0 instructions, or the CACHE Index Store Tag instruction. Since the V_R4300 uses a 32-bit physical address (as opposed to the V_R4000 36-bit physical address and the V_R4200 33-bit physical address), the upper four bits of the *TagLo* register are always filled with zeroes by an Index Store Tag instruction.

Virtual-to-Physical Address Translation Process

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

- In 32-bit mode, the highest 7 to 19 bits (depending upon the page size) of the virtual address are compared to the contents of the TLB VPN2 (virtual page number divided by two).
- In 64-bit mode, the highest 15 to 27 bits (depending upon the page size) of the virtual address are compared to the contents of the TLB VPN2 (virtual page number divided by two).

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

Figure 3-19 illustrates the TLB address translation process.

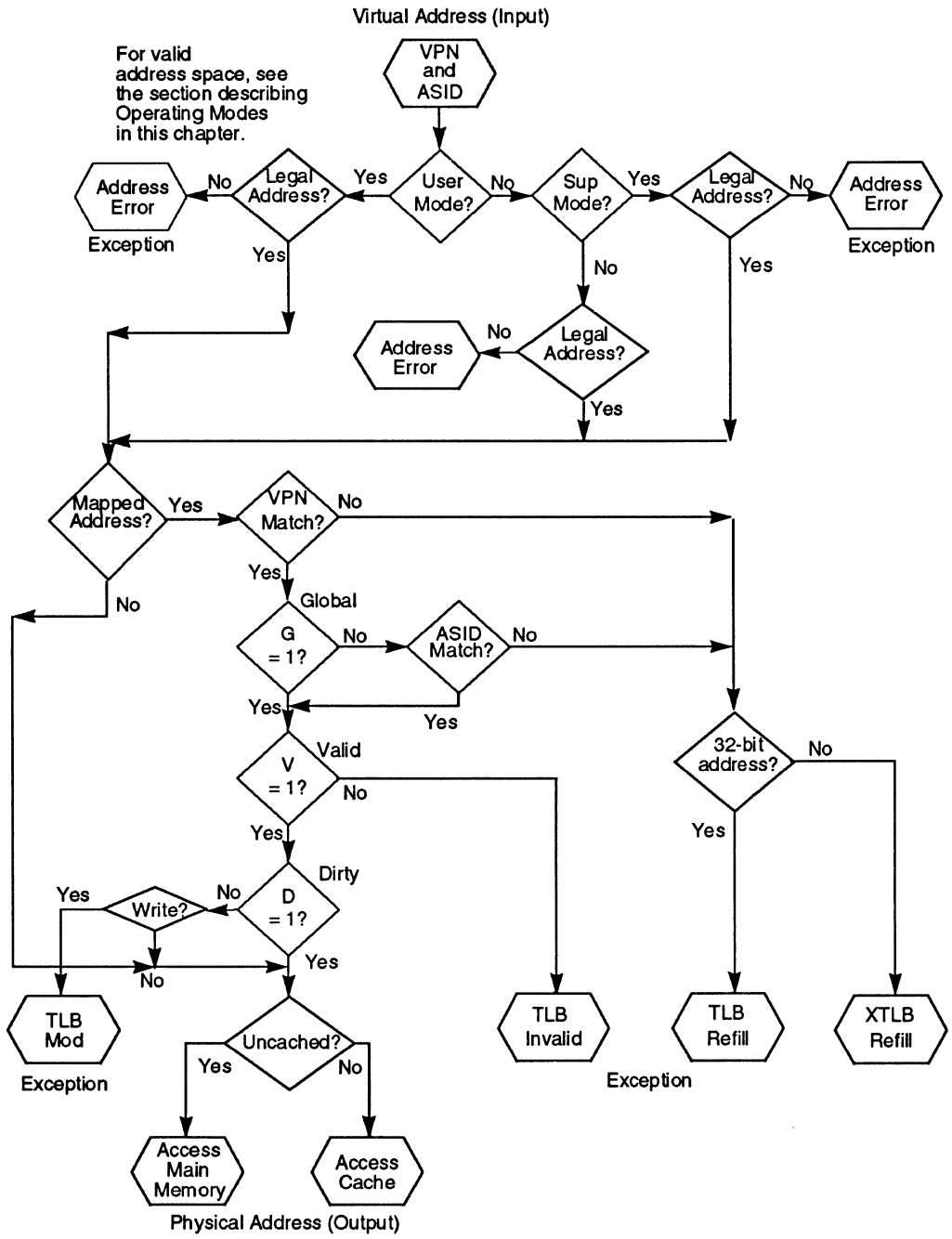


Figure 3-19 TLB Address Translation

TLB Misses

If there is no TLB entry that matches the virtual address, a TLB refill (miss) exception occurs.† If the access control bits (*D* and *V*) indicate that the access is not valid, a TLB Modification or TLB Invalid exception occurs. If the C bits equal 010_2, the physical address that is retrieved accesses main memory, bypassing the cache.

TLB Instructions

Table 3-13 lists the instructions that the CPU provides for working with the TLB. See Appendix A for a detailed description of these instructions.

Table 3-13 TLB Instructions

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

† TLB miss exceptions are described in Chapter 5.

CPU Exception Processing

4

This chapter describes CPU exception processing, including an explanation of exception processing, followed by 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 each exception. For information about Floating-Point Unit exceptions, see Chapter 6.

4.1 How Exception Processing Works

The processor receives exceptions from a number of sources, including translation lookaside buffer (TLB) misses, arithmetic overflows, I/O interrupts, and system calls. When the CPU detects an exception, the normal sequence of instruction execution is suspended and the processor enters Kernel mode (see Chapter 4 for a description of system operating modes).

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

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

The V_R4300 processor supports a supervisor mode and fast TLB refill for all address spaces. The V_R4300 provides a single interrupt enable (*IE*), a base operating mode (User, Supervisor, or Kernel), an exception level (normal or exception, as indicated by the *EXL* bit in the *Status* register), and an error level (normal or error, as indicated by the *ERL* bit in the *Status* register). Interrupts are enabled when the interrupt enable bit, *IE*, is set to a 1, both *EXL* and *ERL* are 0, and the corresponding *IM* field bits in the *Status* register are set to 1. The operating mode is specified by the base mode when the exception level is normal (0), and is set to kernel mode when either the exception level or the error level is a 1. Returning from an exception consists of resetting the exception level to normal.

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, see the description of the individual exception contained in this chapter, or the flowcharts at the end of this chapter.

4.2 Precision of Exceptions

V_R4300 exceptions are logically precise; the instruction that causes an exception and all those that follow it are aborted and can be re-executed after servicing the exception. When succeeding instructions are killed, exceptions associated with those instructions are also killed. Exceptions are not taken in the order detected, but in instruction fetch order.

4.3 Exception Processing Registers

This section describes the CP0 registers that are used in exception processing. Table 4-1 lists these registers, along with their number—each register has a unique identification number that is referred to as its *register number*. For instance, the *PErr* 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. The registers in Table 4-1 are used in exception processing, and are described in the sections that follow.

Table 4-1 CP0 Exception Processing Registers

Register Name	Reg. No.
Context	4
BadVAddr (Bad Virtual Address)	8
Count	9
Compare	11
Status	12
Cause	13
EPC (Exception Program Counter)	14
WatchLo (Memory Reference Trap Address Low)	18
WatchHi (Memory Reference Trap Address High)	19
XContext	20
PErr	26
CacheErr (Cache Error and Status)	27
ErrorEPC (Error Exception Program Counter)	30

CPU general registers are interlocked and the result of an instruction can normally be used by the next instruction; if the result is not available right away, the processor stalls until it is available. CP0 registers and the TLB are not interlocked, however; there may be some delay before a value written by one instruction is available to following instructions. For more information please see Appendix B.

Context Register (4)

The *Context* register is a read/write register containing the pointer to an entry in the kernel page table entry (PTE) array; this array is an operating system data structure that stores virtual-to-physical address translations. When there is a TLB miss, the CPU loads the TLB with the missing translation from the PTE array. The *Context* register duplicates some of the information provided in the *BadVAddr* register, but the information is arranged in a form that is more useful for a software TLB exception handler. Figure 4-1 shows the format of the *Context* register; Table 4-2 describes the *Context* register fields.

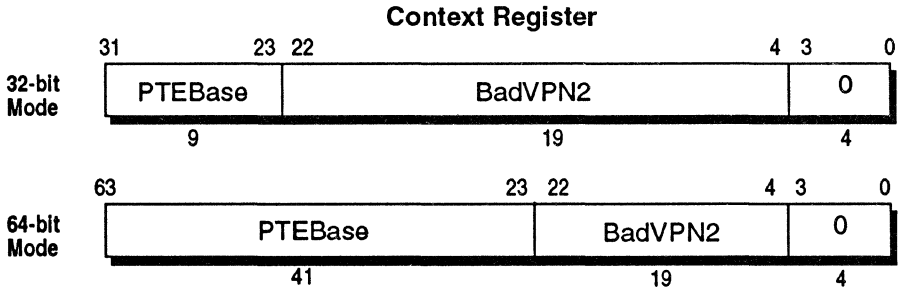


Figure 4-1 Context Register Format

Table 4-2 Context Register Fields

Field	Description
BadVPN2	Virtual Page Number of the failed address, divided by 2.
PTEBase	Read/write field containing the base address of the PTE table of the current user address space.

The 19-bit *BadVPN2* field contains bits 31:13 of the virtual address that caused the TLB miss; bit 12 is excluded because a single TLB entry maps to an even-odd page pair. For a 4-Kbyte page size, this format can directly address the pair-table of 8-byte PTEs. For 16-Mbyte page size, shifting and masking this value produces the correct address.

Bad Virtual Address Register (BadVAddr) (8)

The Bad Virtual Address register (*BadVAddr*) is a read-only register that displays the most recent virtual address that failed to have a valid translation, or that had an addressing error.

Figure 4-2 shows the format of the *BadVAddr* register.

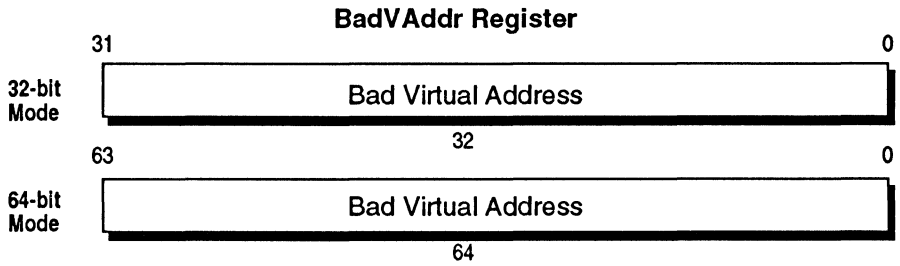


Figure 4-2 BadVAddr Register Format

Count Register (9)

The read/write *Count* register acts as a timer, incrementing at a constant rate—half the **PClock** speed—whether or not instructions are being executed, retired, or any forward progress is actually made through the pipeline. When the register reaches all ones, it rolls over to zero and continues counting. This register can be written for diagnostic purposes or system initialization.

Figure 4-3 shows the format of the *Count* register.

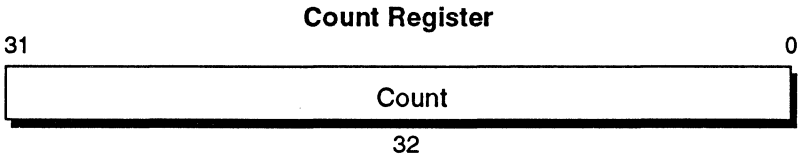


Figure 4-3 Count Register Format

Compare Register (11)

The *Compare* register acts as a timer (see also the *Count* register); it maintains a stable value that does not change on its own.

When the value of the *Count* register equals the value of the *Compare* register, interrupt bit *IP(7)* in the *Cause* register is set. This causes an interrupt as soon as the interrupt is enabled.

Writing a value to the *Compare* register clears the timer interrupt.

For diagnostic purposes, the *Compare* register is a read/write register. Figure 4-4 shows the format of the *Compare* register.

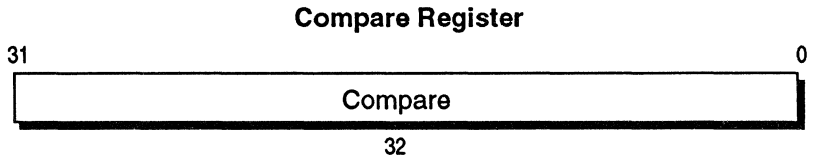


Figure 4-4 Compare Register Format

Status Register (12)

The *Status* register (SR) is a read/write register that contains the operating mode, interrupt enabling, and the diagnostic states of the processor. The following list describes some of the more important *Status* register fields; Figure 4-5 shows the format of the entire register, including descriptions of the fields. Some of the important fields include:

- The 8-bit *Interrupt Mask* (*IM*) field controls the enabling of up to eight individual interrupt conditions. Only when:
  - the interrupts are globally enabled (by setting the *IE* bit), and
  - the individual interrupt condition is enabled (by setting its corresponding *IM* bit),

  does the Interrupt exception, as indicated by the corresponding interrupt request bit (*IP*) in the *Cause* register, occur. For more information, refer to the *Interrupt Pending* (*IP*) field of the *Cause* register, and Chapter 14.
- The 4-bit *Coprocessor Usability* (*CU*) field controls the usability of 4 possible coprocessors. Regardless of the *CU* bit setting, CP0 is always usable in Kernel mode.
- The *Reverse-Endian (RE)* bit, bit 25, reverses the endianness of the machine for the user task. At boot time, the system defaults to big-endian mode, and the kernel runs in the mode indicated by the *BE* bit of the *Config* register. The *RE* bit allows the user to operate either in the same or the opposite endianness of the kernel, as follows:
  - when RE = 1, kernel endianness is specified by the BigEndian bit in the *Config* register and user endianness is the opposite of the kernel
  - when RE = 0, both kernel and user endianness are specified by the BigEndian bit in the *Config* register.

Status Register Format

Figure 4-5 shows the format of the *Status* register. Table 4-3 describes the *Status* register fields, including additional information on the *Diagnostic Status* (*DS*) field. All bits in the *DS* field except *TS* are readable and writable.

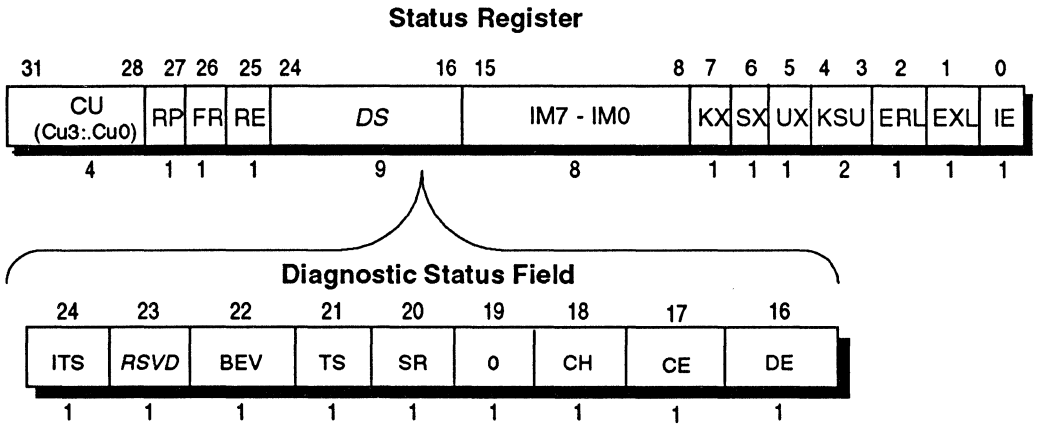


Figure 4-5 Status Register

Table 4-3 Status Register Fields

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
RP	Enables reduced-power operation by reducing the internal clock frequency to one-quarter speed. 0 → full speed 1 → one-quarter internal clock speed
FR	Enables additional floating-point registers 0 → 16 registers 1 → 32 registers
RE	*Reverse-Endian* bit, reverses endianness in User mode.
ITS	Enables Instruction Trace Support and processor status lines.
RSVD	Reserved for future use. Must be written as zeroes, and returns zeroes when read.

Table 4-3 (cont.) Status Register Fields

Field	Description
BEV	Controls the location of TLB refill and general exception vectors. 0 → normal 1 → bootstrap
TS	Indicates TLB shutdown has occurred (read-only) if more than one TLB entry matches a single virtual address. After TLB shutdown, the processor must be reset to restore the TLB.
SR	1→ Indicates a **Reset*** signal or NMI has caused a Soft Reset exception
0	Must be written as zeroes, and returns zeroes when read.
CH	CP0 condition bit. Read/write access by software only; not accessible to hardware.
CE	Contents of the *PErr* register are used to set the parity bits of the cache when CE = 1. **Not used in the V_R4300**
DE	Specifies that cache parity errors cannot cause exceptions. 0 → parity remains enabled 1 → disables parity **Not used in the V_R4300**
IM	*Interrupt Mask* field controls up to eight individual interrupt conditions. 0 → disabled 1 → enabled
KX	Enables 64-bit addressing in Kernel mode. The extended-addressing TLB refill exception is used for TLB misses on kernel addresses. 0 → 32–bit 1 → 64–bit
SX	Enables 64-bit addressing and operations in Supervisor mode. The extended-addressing TLB refill exception is used for TLB misses on supervisor addresses. 0 → 32–bit 1 → 64–bit

Table 4-3 (cont.) Status Register Fields

Field	Description
UX	Enables 64-bit addressing and operations in User mode. The extended-addressing TLB refill exception is used for TLB misses on user addresses. 0 → 32–bit 1 → 64–bit
KSU	Mode bits 10_2 → User 01_2 → Supervisor 00_2 → Kernel
ERL	Error Level; set by the processor when Reset, Soft Reset, NMI, or Cache Error are taken. 0 → normal 1 → error
EXL	Exception Level; set by the processor when any exception other than Reset, Soft Reset, NMI, or Cache Error are taken. 0 → normal 1 → exception
IE	Global Interrupt Enable 0 → disable interrupts 1 → enable interrupts

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 enable the interrupts.

Operating Modes: The following CPU *Status* register bit settings are required for User, Kernel, and Supervisor modes (see Chapter 3 for more information about operating modes).

- The processor is in User mode when $KSU = 10_2$, $EXL = 0$, and $ERL = 0$.
- The processor is in Supervisor mode when $KSU = 01_2$, $EXL = 0$, and $ERL = 0$.
- The processor is in Kernel mode when $KSU = 00_2$, or $EXL = 1$, or $ERL = 1$.

32- and 64-bit Modes: The following CPU *Status* register bit settings select 32- or 64-bit operation for User, Kernel, and Supervisor operating modes. Enabling 64-bit operation permits the execution of 64-bit opcodes and translation of 64-bit addresses. 64-bit operation for User, Kernel and Supervisor modes can be set independently.

- 64-bit addressing for Kernel mode is enabled when $KX = 1$. 64-bit operations are always valid in Kernel mode.
- 64-bit addressing and operations are enabled for Supervisor mode when $SX = 1$.
- 64-bit addressing and operations are enabled for User mode when $UX = 1$.

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

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

User Address Space Accesses: Access to the user address space is allowed in any of the three operating modes.

Status Register Reset

The contents of the *Status* register are undefined after a Cold Reset, except for the following bits:

- *TS* and *RP* = 0
- *ERL* and *BEV* = 1

The *SR* bit distinguishes between the Reset exception and the Soft Reset exception (caused either by **Reset*** or Nonmaskable Interrupt [NMI]).

Cause Register (13)

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

Figure 4-6 shows the fields of this register; Table 4-4 describes the *Cause* register fields. A 5-bit exception code (*ExcCode*) indicates one of the causes, as listed in Table 4-5.

All bits in the *Cause* register, with the exception of the *IP(1:0)* bits, are read-only; *IP(1:0)* are used for software interrupts.

Table 4-4 Cause Register Fields

Field	Description
BD	Indicates whether the last exception taken occurred in a branch delay slot. 1 → delay slot 0 → normal
CE	Coprocessor unit number referenced when a Coprocessor Unusable exception is taken.
IP(7:0)	The *IP* field indicates which interrupts are pending. *IP7* is the timer interrupt bit, set when the *Count* register equals the *Compare* register. *IP(6:2)* are the external interrupts, set when an external interrupt is signalled; an external interrupt is set at one of the external interrupt pins or by a write request on the SysAD bus. *IP(1:0)* are software interrupts; they may be written to set or clear software interrupts
ExcCode	Exception code field (see Table 4-5)
0	Reserved. Must be written as zeroes, and returns zeroes when read.

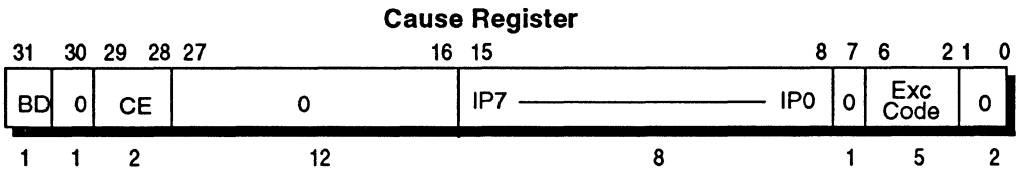


Figure 4-6 Cause Register Format

Table 4-5 Cause Register ExcCode Field

Exception Code Value	Mnemonic	Description
0	Int	Interrupt
1	Mod	TLB Modification exception
2	TLBL	TLB Refill exception (load or instruction fetch)
3	TLBS	TLB Refill exception (store)
4	AdEL	Address Error exception (load or instruction fetch)
5	AdES	Address Error exception (store)
6	IBE	Bus Error exception (instruction fetch)
7	DBE	Bus Error exception (data reference: load or store)
8	Sys	Syscall exception
9	Bp	Breakpoint exception
10	RI	Reserved Instruction exception
11	CpU	Coprocessor Unusable exception
12	Ov	Arithmetic Overflow exception
13	Tr	Trap exception
14	--	Reserved
15	FPE	Floating-Point exception
16–22	–	Reserved
23	WATCH	Reference to *WatchHi/WatchLo* address
24–31	–	Reserved

Exception Program Counter (EPC) Register (14)

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

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

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

The *EXL* bit in the *Status* register is set to a 1 to keep the processor from overwriting the address of the exception-causing instruction contained in the EPC register in the event of another exception.

Figure 4-7 shows the format of the *EPC* register.

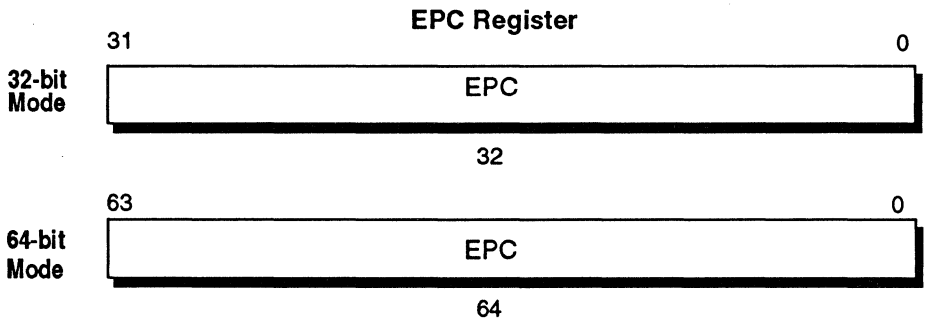


Figure 4-7 EPC Register Format

WatchLo (18) and WatchHi (19) Registers

The V_R4300 processor provides a debugging feature to detect references to a selected physical address; load and store operations to the location specified by the *WatchLo* and *WatchHi* registers cause a Watch exception (described later in this chapter).

Figure 4-8 shows the format of the *WatchLo* and *WatchHi* registers; Table 4-6 describes the *WatchLo* and *WatchHi* register fields.

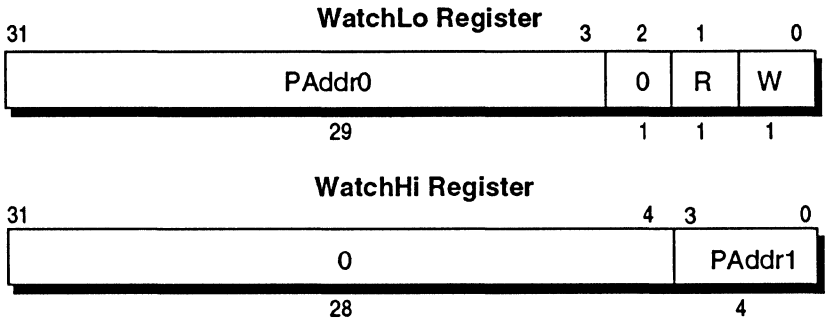


Figure 4-8 WatchLo and WatchHi Register Formats

Table 4-6 WatchHi and WatchLo Register Fields

Field	Description
PAddr1	Bits 35:32 of the physical address†
PAddr0	Bits 31:3 of the physical address
R	Trap on read access if set to 1
W	Trap on write access if set to 1
0	Reserved. Must be written as zeroes, and returns zeroes when read.

† The MSB of the V_R4300 physical address is PA(31). The remaining upper four physical address bits, PA(35:32), although ignored during address translation, are supplied by the *WatchHi* register to maintain compatibility with the V_R4000.

XContext Register (20)

The read/write *XContext* register contains a pointer to an entry in the kernel page table entry (PTE) array, an operating system data structure that stores virtual-to-physical address translations. When there is a TLB miss, the operating system software loads the TLB with the missing translation from the PTE array. The *XContext* register duplicates some of the information provided in the *BadVAddr* register, and puts it in a form useful for a software TLB exception handler. The *XContext* register is for use with the XTLB refill handler, which loads TLB entries for references to a 64-bit address space, and is included solely for operating system use. The operating system sets the PTE base field in the register, as needed. Figure 4-9 shows the format of the *XContext* register; Table 4-7 describes the *XContext* register fields.

XContext Register

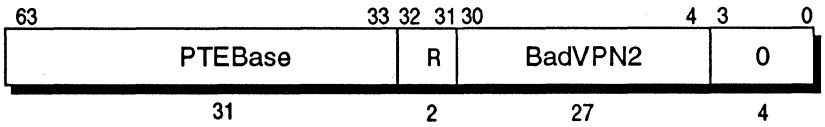


Figure 4-9 XContext Register Format

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

Table 4-7 XContext Register Fields

Field	Description
BadVPN2	VPN of the failed virtual address, divided by two.
R	The *Region* field contains bits 63:62 of the virtual address. 00 = user 01 = supervisor 11 = kernel.
PTEBase	The *Page Table Entry Base* read/write field indicates the base address of the PTE in the current user address space.

Parity Error (PErr) Register (26)

The *PErr* register is implemented only to maintain compatibility with the V_R4000/V_R4200 processors. Since the V_R4300 processor does not implement parity on caches, this register is not used by hardware. It is software readable and writable.

Figure 4-10 shows the format of the *PErr* register; Table 4-8 describes the register fields.

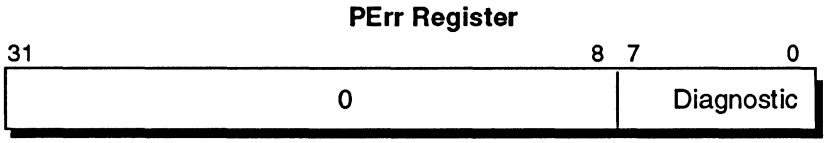


Figure 4-10 PErr Register Format

Table 4-8 PErr Register Fields

Field	Description
Diagnostic	An 8-bit field field used during diagnostics.
0	Must be written as zeroes, and returns zeroes when read.

Cache Error (CacheErr) Register (27)

The CacheErr register is implemented only to maintain compatibility with the V_R4000/V_R4200 processors. Since the V_R4300 processor does not implement cache errors, this register is not used by hardware. It is a read-only register that returns 0 when read.

CacheErr Register

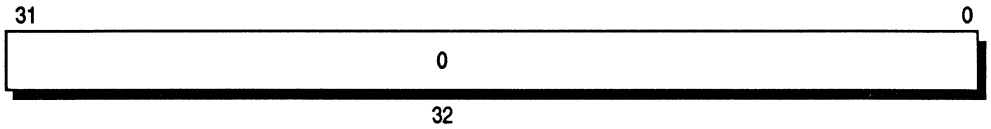


Figure 4-11 CacheErr Register Format

Table 4-9 CacheErr Register Fields

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

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

The *ErrorEPC* register is similar to the *EPC* register, and is used to store the program counter (PC) on Cold 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:

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

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

Figure 4-12 shows the format of the *ErrorEPC* register.

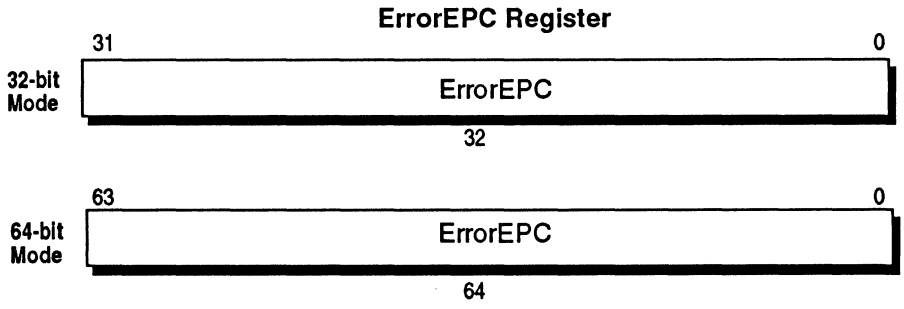


Figure 4-12 ErrorEPC Register Format

4.4 Processor Exceptions

This section describes the processor exceptions—it describes the cause of each exception, its processing by the hardware, and servicing by a handler (software). The types of exceptions, with exception processing operations, are described in the next section.

Exception Types

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

- Cold Reset
- Soft Reset
- nonmaskable interrupt (NMI)
- remaining processor exceptions

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

When the processor takes an exception, the *EXL* bit is set to 1, which means the system is in Kernel mode. After saving the appropriate state, the exception handler typically changes *KSU* to Kernel mode and resets the *EXL* bit back to 0. When restoring the state and restarting, the handler restores the previous value of the *KSU* field and sets the *EXL* bit back to 1.

Returning from an exception also resets the *EXL* bit to 0 (see the ERET instruction in Appendix A).

Exception Vector Locations

The Reset, Soft Reset, and NMI exceptions are always vectored to the dedicated Reset exception vector at an uncached and unmapped address. Addresses for all other exceptions are a combination of a *vector offset* and a *base address*.

The boot-time vectors (when *BEV* = 1 in the *Status* register) are at uncached and unmapped addresses. During normal operation (when *BEV* = 0) the regular exceptions have vectors in cached address spaces.

Table 4-10 shows the 64-bit-mode vector base address for all exceptions; the 32-bit mode address is the low-order 32 bits (for instance, the base address for NMI in 32-bit mode is 0xBFC0 0000).

Table 4-11 shows the vector offset added to the base address to create the exception address.

Table 4-10 Exception Vector Base Addresses

Exception	BEV	
	0	1
Others	0xFFFF FFFF 8000 0000	0xFFFF FFFF BFC0 0200
Reset, NMI, Soft Reset	0xFFFF FFFF BFC0 0000	

Table 4-11 Exception Vector Offsets

Exception	V_R4000 Processor Vector Offset
TLB refill, EXL = 0	0x000
XTLB refill, EXL = 0 (X = 64-bit TLB)	0x080
Others	0x180
Reset, Soft Reset, NMI	none

Priority of Exceptions

The remainder of this chapter describes exceptions in the order of their priority shown in Table 4-12 with (certain of the exceptions, such as the TLB exceptions and Instruction/Data exceptions, grouped together for convenience). While more than one exception can occur for a single instruction, only the exception with the highest priority is reported.

Table 4-12 Exception Priority Order

Cold Reset *(highest priority)*
Soft Reset caused by **Reset*** † signal
Nonmaskable Interrupt (NMI) (Soft Reset exception caused by NMI)
Address error — Instruction fetch
TLB/XTLB refill — Instruction fetch
TLB invalid — Instruction fetch
Bus error — Instruction fetch
System Call
Breakpoint
Coprocessor Unusable
Reserved Instruction
Trap
Integer overflow
Floating-Point Exception
Address error — Data access
TLB/XTLB refill — Data access
TLB invalid — Data access
TLB modified — Data write
Watch
Bus error — Data access
Interrupt *(lowest priority)*

† In the following sections—indeed, throughout this book—a signal followed by an asterisk, such as **Cold Reset***, is low active.

Generally speaking, the exceptions described in the following sections are handled ("processed") by hardware; these exceptions are then serviced by software.

Cold Reset Exception

Cause

The Cold Reset exception occurs when the **ColdReset*** signal is asserted and then deasserted. (**Reset*** must be asserted along with **ColdReset***). This exception is not maskable.

Processing

The CPU provides a special interrupt vector for this exception:

- location 0xBFC0 0000 in 32-bit mode
- location 0xFFFF FFFF BFC0 0000 in 64-bit mode

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

- *SR*, *RP*, and *TS* of the *Status* register, the *Wired* register, and *EP(3:0)* of the *Config* register are all cleared to 0
- *ERL* and *BEV* of the *Status* register, *BE* of the *Config* register are all set to 1.
- The *Random* register is initialized to the value of its upper bound.

All other bits are undefined.

Servicing

The Cold Reset exception is serviced by:

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

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

Soft Reset Exception

Cause

A Soft Reset† occurs when the **Reset*** signal goes from assertion to deassertion without the **ColdReset*** signal having been asserted immediately prior; in other words, there must be at least one cycle in which neither **Reset*** nor **ColdReset*** are asserted. A Soft Reset immediately resets all state machines, and sets the *SR* bit of the *Status* register. In the process of a Soft reset, currently executing operations may be abandoned but, in general, processor data is preserved for debugging. *Status* register bits *TS* and *RP* are set to zero, and *ERL* and *BEV* are set to one. Execution begins at the reset vector in response to the **Reset*** pin being deasserted. This exception is not maskable.

Processing

A special exception vector (0xFFFF FFFF BFC0 0000) is provided for this exception, which is the same address as that for the Cold Reset exception. This vector is located within the unmapped and uncached address space so the cache and TLB need not be initialized to handle the exception. The *SR* bit of the *Status* register differentiates a Soft Reset from a Cold Reset exception; when the *SR* bit is set to a 1, a Soft Reset exception has occurred.

As previously noted, state machines interrupted by **Reset*** may cause some register contents to be inconsistent with the other processor state. Otherwise, on an exception caused by **Reset*** or NMI 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
- *RP* bit of the *Status* register, which is set to 0
- *TS* bit of the *Status* register, which is set to 0
- PC is set to the reset vector 0xFFFF FFFF BFC0 0000

Soft reset exception processing is shown in Figure 4-17.

† In this book, a Soft Reset exception caused by assertion of the **Reset*** signal is referred to as a "soft reset" or "warm reset." A Soft Reset exception caused by a nonmaskable interrupt (NMI) is referred to as a "nonmaskable interrupt exception."

Servicing

The Soft Reset exception is serviced by saving as much of the current processor state for diagnostic purposes, and then reinitializing as if for the Cold Reset Exception.

The exception initiated by **Reset*** is intended to quickly reinitialize a previously operating processor after a fatal error.

The exceptions due to **Reset*** and NMI appear identical to software; both exceptions jump to the Reset exception vector and have the *Status* register *SR* bit set. Unless external hardware provides a way to distinguish between the two, they are serviced by saving the current user-visible processor state for diagnostic purposes and reinitializing as for the Reset exception. It is not normally possible to continue program execution after returning from this exception, since a **Reset*** signal can be accepted anytime and an NMI can occur in the midst of another error exception.

Nonmaskable Interrupt (NMI) Exception

Cause

The NMI is caused either by an assertion of the **NMI*** signal or an external write to the *Int*[6]* bit of the *Interrupt* register. Upon receiving an NMI, theprocessor jumps to the Reset exception vector and sets the *SR* bit in the *Status* register. No state machines or other bits in the chip are affected.

This exception is not maskable; it occurs regardless of the settings of the *EXL*, *ERL*, and *IE Status* register bits.

Processing

The Reset exception vector (0xFFFF FFFF BFC0 0000) is used for this exception. This vector is located within the unmapped and uncached address space so that the cache and TLB need not be initialized to handle the exception. The *SR* bit of the *Status* register is set to differentiate this exception from a Cold Reset exception.

Unlike Reset, but like other exceptions, NMI is taken only at instruction boundaries; thus the state of the caches and memory system are preserved by this exception. The caches, TLB, and normal exception vectors need not be properly initialized. The contents of all registers are preserved when this exception occurs, except for the *ErrorEPC* register, which contains the restart PC, and the *ERL* bit of the *Status* register, which is set to one.

Servicing

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

Exceptions due to **Reset*** and NMI appear identical to software; both exceptions jump to the Reset exception vector and have the *Status* register *SR* bit set. Unless external hardware provides a way to distinguish between the two, they are serviced by saving the current user-visible processor state for diagnostic purposes and reinitializing as for the Reset exception. It is not normally possible to continue program execution after returning from this exception, since a **Reset*** signal can be accepted anytime and an NMI can occur in the midst of another error exception.

Address Error Exception

Cause

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

- load or store a doubleword that is not aligned on a doubleword boundary
- load or store a word that is not aligned on a word boundary
- load or store a halfword that is not aligned on a halfword boundary
- reference the kernel address space from User or Supervisor mode
- reference the supervisor address space from User mode
- reference an address not in Kernel, Supervisor, or User space in 64-bit Kernel, Supervisor, or User mode.

This exception is not maskable.

Processing

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

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

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

Servicing

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

TLB Exceptions

Three types of TLB exceptions can occur:

- TLB Refill/Extended Addressing TLB Refill exception occurs when there is no TLB entry that matches an attempted reference to a mapped address space.
- TLB Invalid exception occurs when a virtual address reference matches a TLB entry that is marked invalid.
- TLB Modified exception occurs when a store operation virtual address reference to memory matches a TLB entry which is marked valid but is not dirty (the entry is not writable). As a result, this exception only occurs for the data cache, resulting in a lower priority for this exception.

The following three sections describe these TLB exceptions.

TLB Refill/Extended Addressing TLB Refill Exception

Cause

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

Processing

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

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

Servicing

To service this exception, the contents of the *Context* or *XContext* register are used as a virtual address to fetch memory locations containing the physical page frame and access control bits for a pair of TLB entries. The two entries are placed into the *EntryLo0/EntryLo1* register; the *EntryHi* and *EntryLo* registers are written into the TLB.

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

TLB Invalid Exception

Cause

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

Processing

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

When this exception occurs, the *BadVAddr*, *Context*, *XContext* and *EntryHi* registers contain the virtual address that failed address translation. The *EntryHi* register also contains the ASID from which the translation fault occurred. The contents of the *EntryLo* register are undefined.

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

Servicing

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

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

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

TLB Modified Exception†

Cause

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

Processing

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

When this exception occurs, the *BadVAddr*, *Context*, *XContext* and *EntryHi* registers contain the virtual address that failed address translation. The *EntryHi* register also contains the ASID from which the translation fault occurred. The contents of the *EntryLo* register are undefined.

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

Servicing

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

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

† As this exception only occurs for the data cache, in priority this exception is actually located below the Floating-Point exception (See Table 4-12). It is placed here with the other two TLB exceptions for convenience sake.

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 whether the instruction (as indicated by the *EPC* register and *BD* bit in the *Cause* register) caused the exception by an instruction reference, load operation, or store operation. The *EPC* register contains the address of the instruction that caused the exception, unless it is in a branch delay slot, in which case the *EPC* register contains the address of the preceding branch instruction and the *BD* bit of the *Cause* register is set.

Servicing

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

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

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

System Call Exception

Cause

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

Processing

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

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

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

Servicing

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

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

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

Breakpoint Exception

Cause

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

Processing

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

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

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

Servicing

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

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.

Coprocessor Unusable Exception

Cause

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

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

This exception is not maskable.

Processing

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

The contents of the *Coprocessor Usage Error* field of the coprocessor *Cause* 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 and the *BD* bit of the *Cause* register is set.

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 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 a UNIX SIGILL/ ILL_PRIVIN_FAULT (illegal instruction/privileged instruction fault) signal. This error is usually fatal.

Reserved Instruction Exception

Cause

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

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

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

This exception is not maskable.

Processing

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

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

Servicing

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

Trap Exception

Cause

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

Processing

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

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

Servicing

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

† See Appendix A for a description of these instructions.

Integer Overflow Exception

Cause

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

Processing

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

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

Servicing

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

† See Appendix A for a description of these instructions.

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.

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 and the *BD* bit of the *Cause* register is set.

Servicing

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

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

Watch Exception

Cause

A Watch exception occurs when a load or store instruction references the physical address specified in the *WatchLo/WatchHi* System Control Coprocessor (CP0) registers. The *WatchLo/WatchHi* registers specify whether a load or store or both could have initiated this exception.

The CACHE instruction never causes a Watch exception.

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

Processing

The common exception vector is used for this exception, and the *Watch* 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 and the *BD* bit of the *Cause* register is set.

Servicing

The Watch exception is a debugging aid; typically the exception handler transfers control to a debugger, allowing the user to examine the situation.

To continue, the Watch exception must be disabled to execute the faulting instruction. The Watch exception must then be reenabled. The faulting instruction can be executed either by interpretation or by setting breakpoints.

Interrupt Exception

Cause

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

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

Processing

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

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

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 and the *BD* bit of the *Cause* register is set.

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.

4.5 Exception Handling and Servicing Flowcharts

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

- general exceptions and their exception handler
- TLB/XTLB miss exception and their exception handler
- Cold Reset, Soft Reset and NMI exceptions, and a guideline to their handler.

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

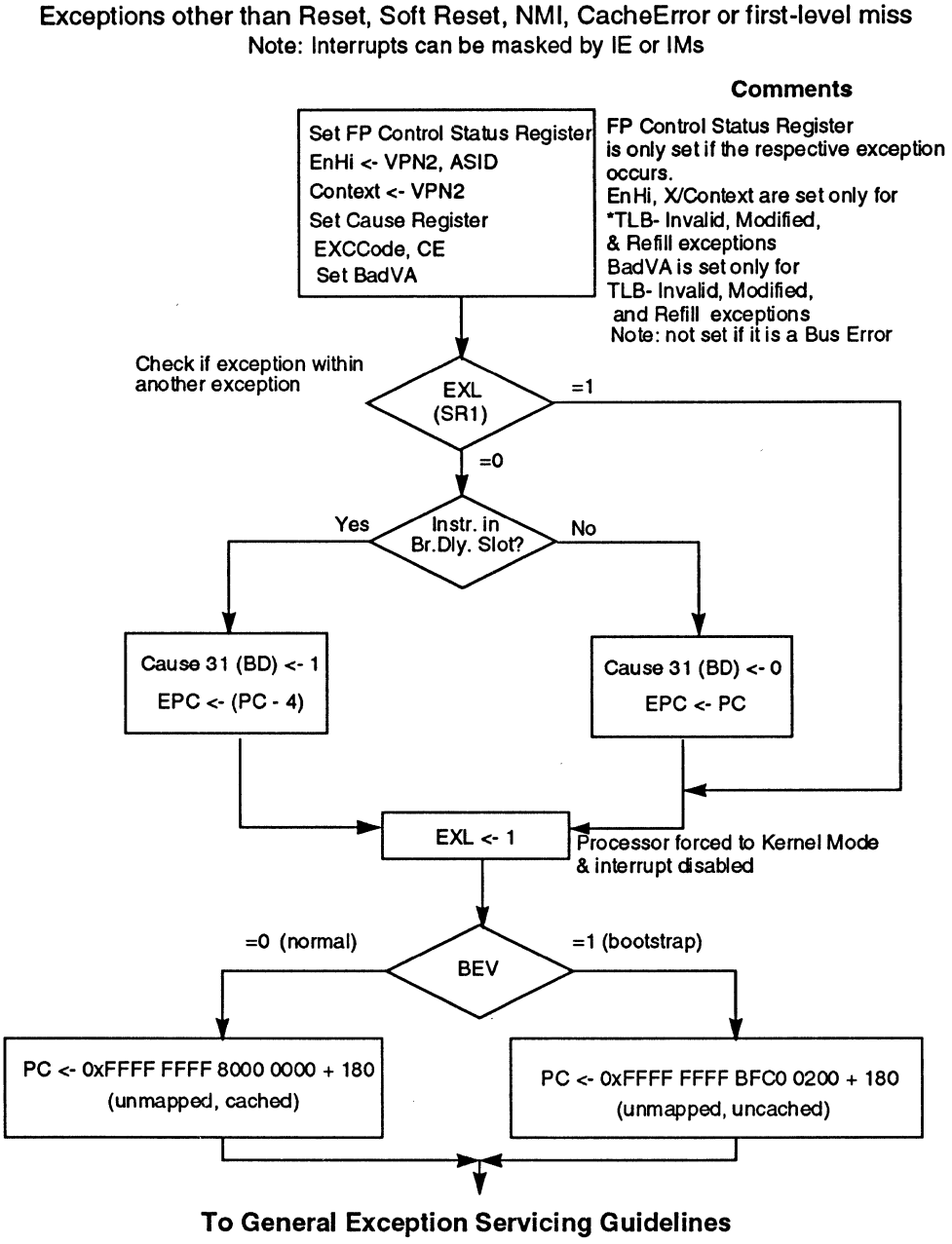


Figure 4-13 General Exception Handler (HW)

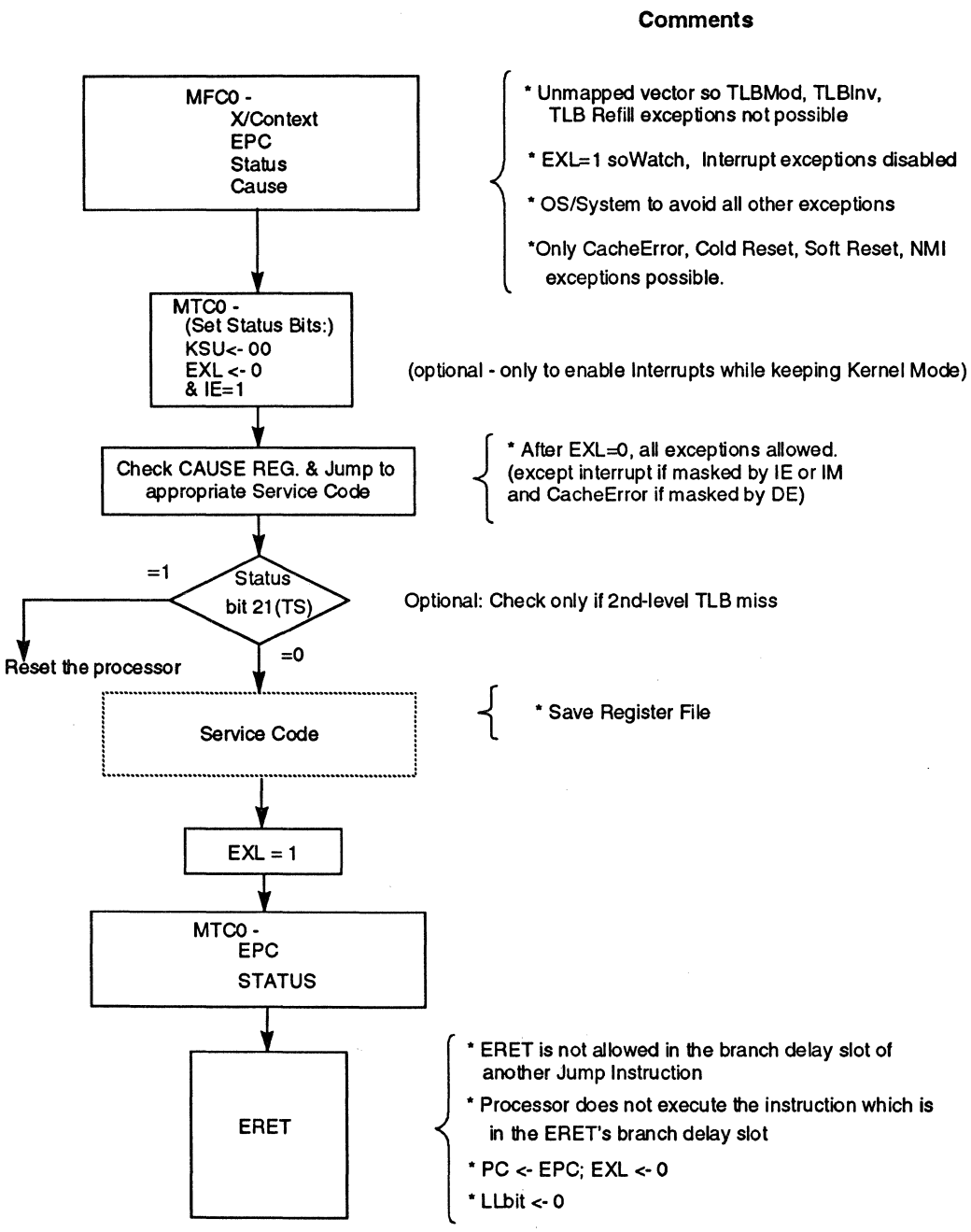


Figure 4-14 General Exception Servicing Guidelines (SW)

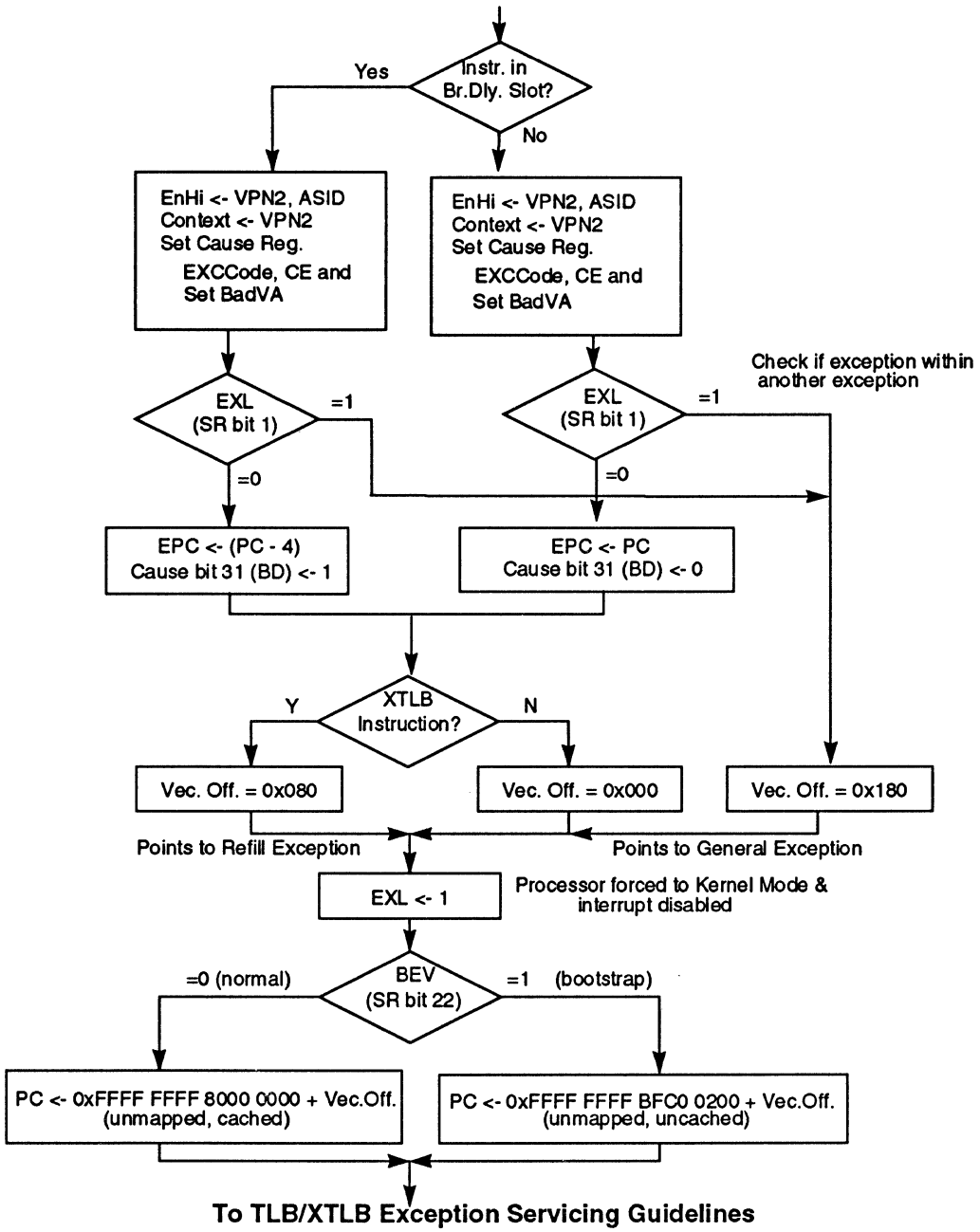


Figure 4-15 TLB/XTLB Miss Exception Handler (HW)

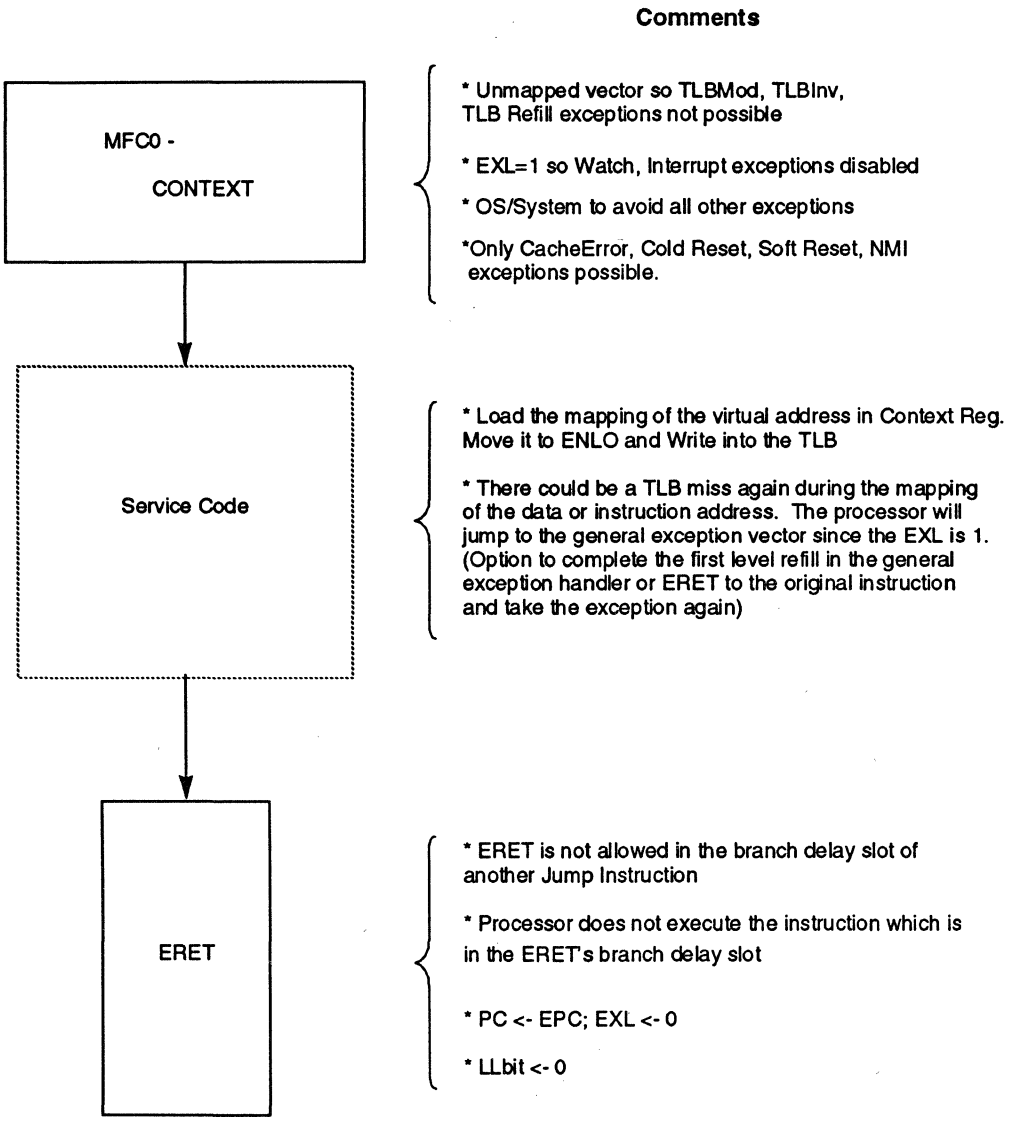


Figure 4-16 TLB/XTLB Exception Servicing Guidelines (SW)

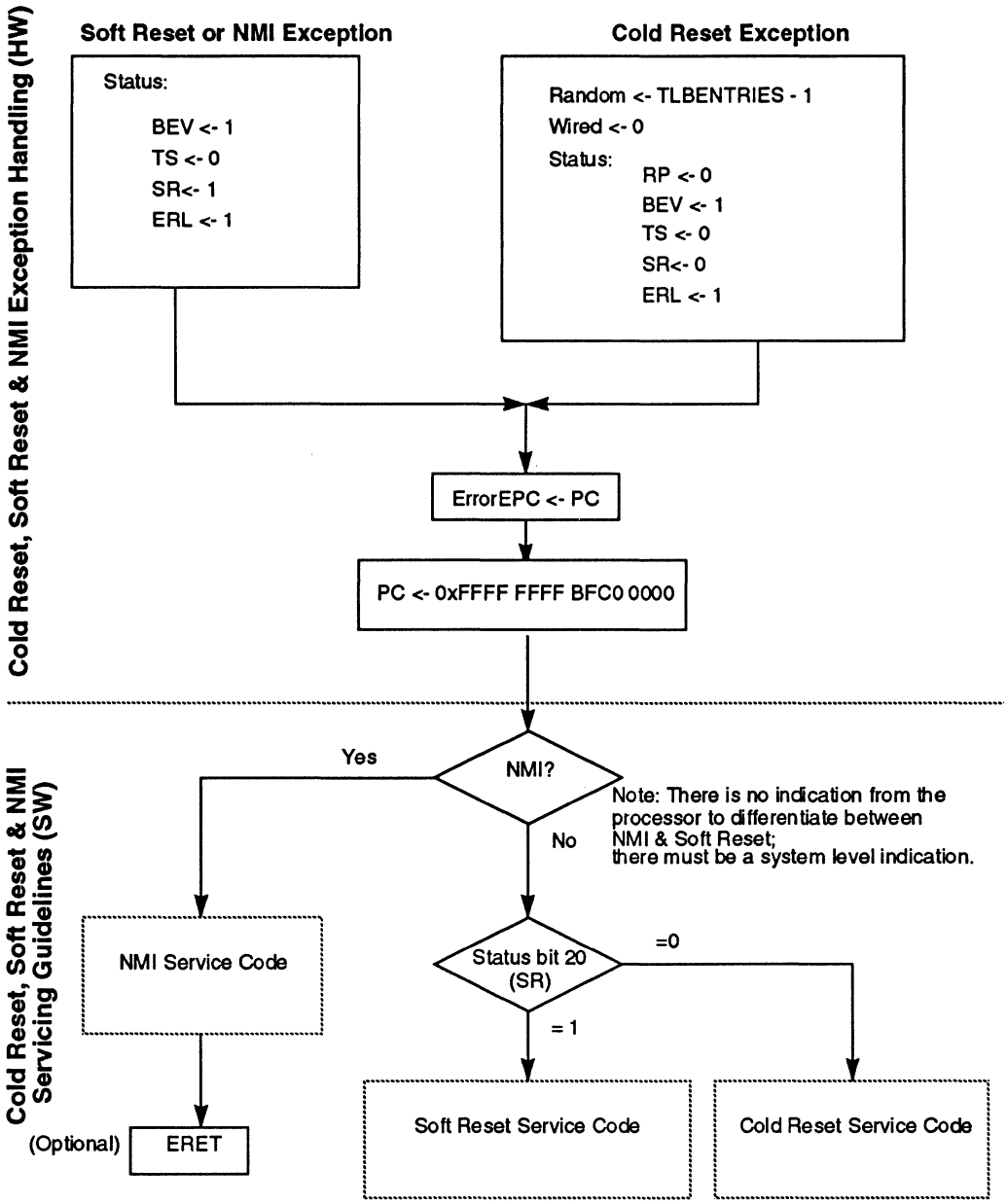


Figure 4-17 Cold Reset, Soft Reset & NMI Exception Handling (HW) and Servicing Guidelines (SW)

Floating-Point Operations

5

This chapter describes the MIPS floating-point operations, including the programming model, instruction set and formats.

MIPS floating-point operations fully conform to the requirements of ANSI/IEEE Standard 754–1985, *IEEE Standard for Binary Floating-Point Arithmetic.* In addition, the MIPS architecture fully supports the recommendations of the standard and precise exceptions.

5.1 Overview

All floating-point instructions, as defined in the MIPS ISA for the floating-point coprocessor, CP1, are processed by the same hardware unit that executes integer instructions. Logically, the FPU exists as an individual coprocessor; however, unlike previous implementations, the V_R4300 FPU is physically integrated into the CPU. The CPU and the FPU use a common datapath and FPU instructions are fully-implemented in the CPU hardware. Unlike the V_R4000 implementation, V_R4300 integer instructions cannot be executed while a multicycle floating-point instruction is completing.

The execution of floating-point instructions can be disabled by the coprocessor usability *CU* bit defined in the CP0 *Status* register.

5.2 Floating-Point General Registers (FGRs)

CP1 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 when the *FR* bit in the CPU *Status* register equals 0; or as 32 general purpose registers (32 FGRs), each of which is 64-bits wide when *FR* equals 1. The CPU accesses these registers through move, load, and store instructions.
- As 16 floating-point registers (see the next section for a description of FPRs), each of which is 64-bits wide, when the *FR* bit in the CPU *Status* register equals 0. The FPRs hold values in either single- or double-precision floating-point format. Each FPR corresponds to adjacently numbered FGRs as shown in Figure 5-1.
- As 32 floating-point registers (see the next section for a description of FPRs), each of which is 64-bits wide, when the *FR* bit in the CPU *Status* register equals 1. The FPRs hold values in either single- or double-precision floating-point format. Each FPR corresponds to an FGR as shown in Figure 5-1.

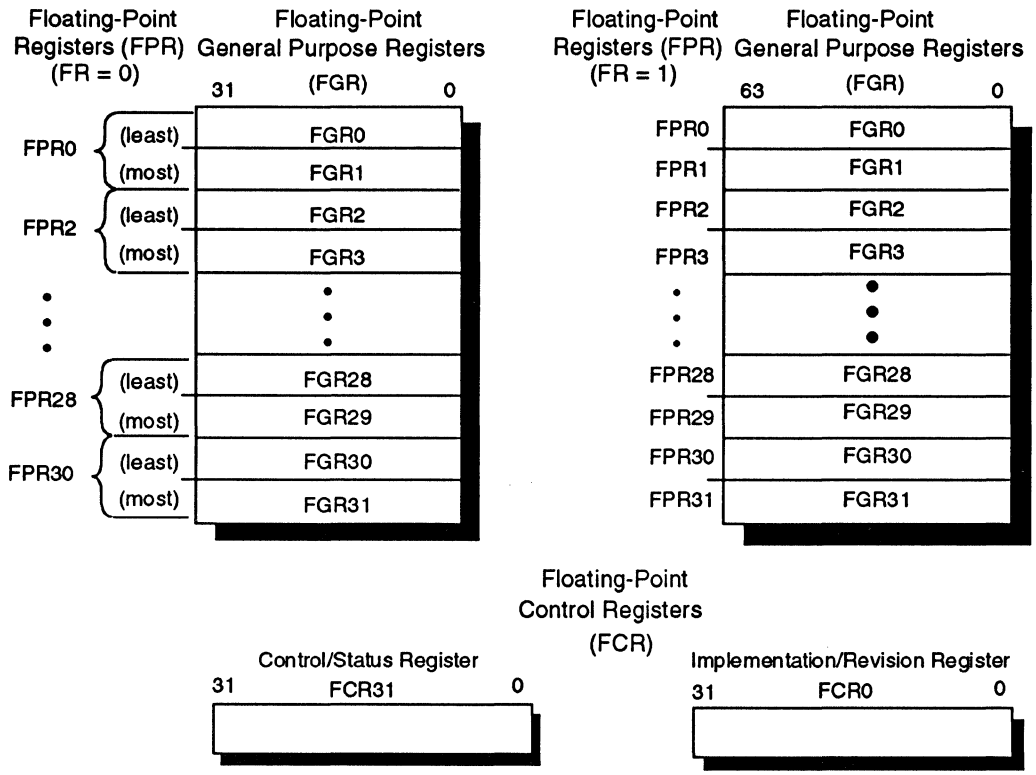


Figure 5-1 FP Registers

5.3 Floating-Point Registers

CP1 provides:

- 16 *Floating-Point* registers (*FPRs*) when the *FR* bit in the *Status* register equals 0, or
- 32 *Floating-Point* registers (*FPRs*) when the *FR* bit in the *Status* register equals 1.

These 64-bit registers hold floating-point values during floating-point operations and are physically formed from the *General Purpose* registers (*FGRs*). When the *FR* bit in the *Status* register equals 1, the *FPR* references a single 64-bit *FGR*.

The *FPRs* hold values in either single- or double-precision floating-point format. If the *FR* bit equals 0, only even numbers (the *least* register, as shown in Figure 5-1) can be used to address *FPRs*. When the *FR* bit is set to a 1, all *FPR* register numbers are valid.

If the *FR* bit equals 0 during a double-precision floating-point operation, the general registers are accessed in double pairs. Thus, in a double-precision operation, selecting *Floating-Point Register 0* (*FPR0*) actually addresses adjacent *Floating-Point General Purpose* registers *FGR0* and *FGR1*.

Floating-Point Control Registers

The MIPS RISC architecture defines 32 floating-point control registers (*FCRs*); the V_R4300 processor implements two of these registers: *FCR0* and *FCR31*. These *FCRs* are described below:

- The *Implementation/Revision* register *(FCR0)* holds revision information.
- 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 5-1 lists the assignments of the *FCRs*.

Table 5-1 Floating-Point Control Register Assignments

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

Implementation and Revision Register, (FCR0)

The read-only *Implementation and Revision* register (*FCR0*) specifies the implementation and revision number of CP1. This information can determine the coprocessor revision and performance level, and can also be used by diagnostic software.

Figure 5-2 shows the layout of the register; Table 5-2 describes the *Implementation and Revision* register (*FCR0*) fields.

Implementation/Revision Register (FCR0)

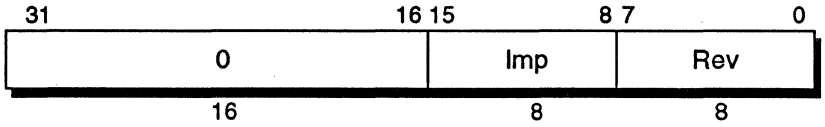


Figure 5-2 Implementation/Revision Register

Table 5-2 FCR0 Fields

Field	Description
Imp	Implementation number (0x0B)
Rev	Revision number in the form of *y.x*
0	Reserved. Returns zeroes when read.

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

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

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

Control/Status Register (FCR31)

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

Figure 5-3 shows the format of the *Control/Status* register, and Table 5-3 describes the *Control/Status* register fields. Figure 5-4 shows the *Control/Status* register *Cause, Flag,* and *Enable* fields.

Control/Status Register (FCR31)

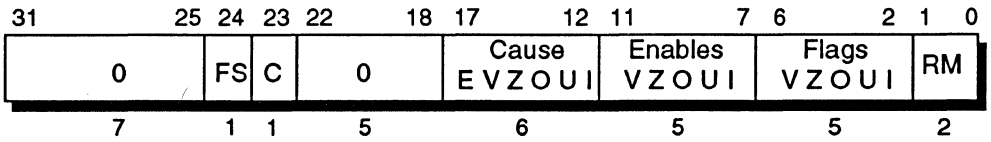


Figure 5-3 FP Control/Status Register Bit Assignments

Table 5-3 Control/Status Register Fields

Field	Description
FS	When set, denormalized results can be flushed instead of causing an unimplemented operation exception.
C	Condition bit. See description of *Control/Status* register *Condition* bit.
Cause	Cause bits. See Figure 5-4 and the description of *Control/Status* register *Cause, Flag,* and *Enable* bits.
Enables	Enable bits. See Figure 5-4 and the description of *Control/Status* register *Cause, Flag,* and *Enable* bits.
Flags	Flag bits. See Figure 5-4 and the description of *Control/Status* register *Cause, Flag,* and *Enable* bits.
RM	Rounding mode bits. See Table 5-5 and the description of *Control/Status* register *Rounding Mode Control* bits.

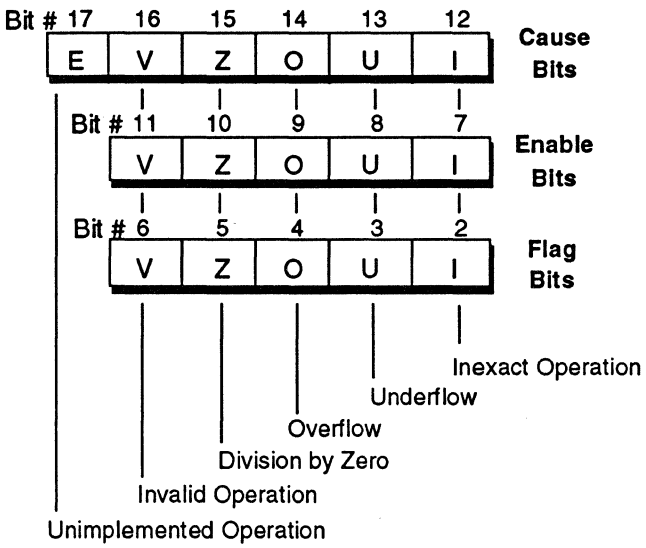


Figure 5-4 Control/Status Register Cause, Flag, and Enable Fields

Accessing the FP Control and Implementation/Revision Registers

The *Control/Status* and the *Implementation/Revision* registers are read by a Move Control From Coprocessor 1 (CFC1) instruction.

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. The *Implementation/Revision* register is a read-only register. There are no pipeline hazards (between any instructions) associated with floating-point control registers.

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

The *FS* bit enables the flushing of denormalized values. When the *FS* bit is set and the Underflow and Inexact *Enable* bits are not set, denormalized results are flushed instead of causing an Unimplemented Operation exception. Results are flushed either to ±0 or ± the minimum normalized value, depending upon the rounding mode (see Table 5-4 below), and the Underflow and Inexact *Flag* and *Cause* bits are set.

Table 5-4 Flush Values of Denormalized Results

Denormalized Result	Flushed Result Rounding Mode			
	RN	Rz	RP	RM
Positive	+0	+0	$+2^{Emin}$	+0
Negative	-0	-0	-0	-2^{Emin}

Control/Status Register Condition Bit

When a floating-point Compare operation takes place, the result is stored at bit 23, the *Condition* bit. 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 FP instructions.

Control/Status Register Cause, Flag, and Enable Fields

Figure 5-4 illustrates the *Cause*, *Flag*, and *Enable* fields of the *Control/Status* register. The *Cause* and *Flag* fields are updated by all conversion, computational (except MOV.fmt), CTC1, reserved, and unimplemented instructions. All other instructions have no affect on these fields.

Cause Bits

Bits 17:12 in the *Control/Status* register contain *Cause* bits, as shown in Figure 5-4, which reflect the results of the most recently executed floating-point instruction. The *Cause* bits are a logical extension of the CP0 *Cause* register; they identify the exceptions raised by the last floating-point operation. If the corresponding *Enable* bit is set at the time of the exception, a floating-point exception and interrupt is raised. If more than one exception occurs on a single instruction, each appropriate bit is set.

The *Cause* bits are updated by most floating-point 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. Within the set of floating-point instructions that update the *Cause* bits, the *Cause* field indicates the exceptions raised by the most-recently-executed instruction.

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

Enable Bits

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

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

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

For a floating-point operation that sets only unenabled *Cause* bits, no floating-point 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

The *Flag* bits are cumulative and indicate the exceptions that were raised by the operations that were executed since the bits 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.

When a floating-point exception is taken, the flag bits are not set by the hardware; floating-point exception software is responsible for setting these bits before invoking a user handler.

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 5-5, these bits specify the rounding mode that CP1 uses for all floating-point operations.

Table 5-5 Rounding Mode Bit Decoding

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 $+\infty$: round to value closest to and not less than the infinitely precise result.
3	RM	Round toward $-\infty$: round to value closest to and not greater than the infinitely precise result.

5.4 Floating-Point Formats

CP1 performs both 32-bit (single-precision) and 64-bit (double-precision) IEEE standard floating-point operations. The 32-bit single-precision format has a 24-bit signed-magnitude fraction field (*f+s*) and an 8-bit exponent (*e*), as shown in Figure 5-5.

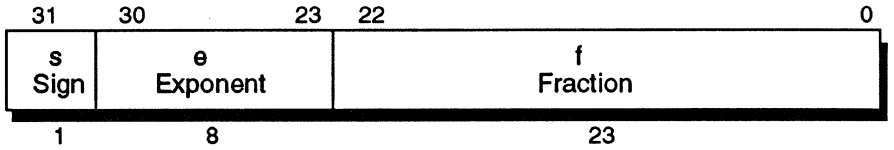


Figure 5-5 Single-Precision Floating-Point Format

The 64-bit double-precision format has a 53-bit signed-magnitude fraction field (*f+s*) and an 11-bit exponent, as shown in Figure 5-6.

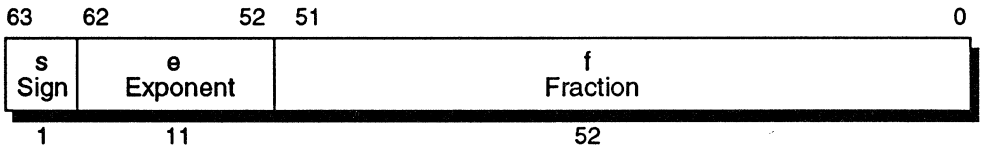


Figure 5-6 Double-Precision Floating-Point Format

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

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

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

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

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

For single- and double-precision formats, the value of a number, v, is determined by the equations shown in Table 5-6.

Table 5-6 Equations for Calculating Values in Single and Double-Precision Floating-Point Format

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 $v = (-1)^s 0$

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 5-7 defines the values for the format parameters; minimum and maximum floating-point values are given in Table 5-8.

Table 5-7 Floating-Point Format Parameter Values

Parameter	Format	
	Single	Double
E_{max}	+127	+1023
E_{min}	–126	–1022
Exponent *bias*	+127	+1023
Exponent width in bits	8	11
Integer bit	hidden	hidden
Fraction width in bits	23†	52†
Format width in bits	32	64

† Excluding the sign bit.

Table 5-8 Minimum and Maximum Floating-Point Values

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

5.5 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 5-7 illustrates binary single fixed-point format and Figure 5-7 illustrates binary long fixed-point format; Table 5-9 lists the binary fixed-point format fields.

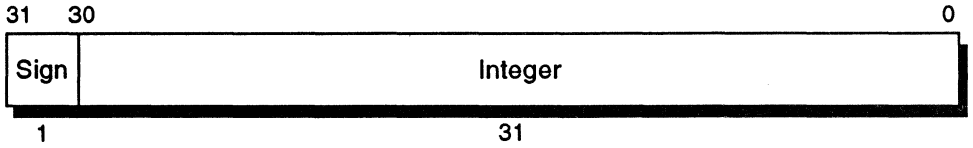


Figure 5-7 Binary Single Fixed-Point Format

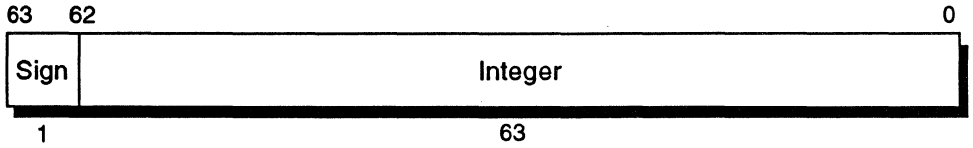


Figure 5-8 Binary Long Fixed-Point Format

Field assignments of the binary fixed-point format are:

Table 5-9 Binary Fixed-Point Format Fields

Field	Description
sign	sign bit
integer	integer value (2's complement)

Floating-Point Exceptions

6

This chapter describes 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.

6.1 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 *Cause* bits are set at the beginning of the operation and a software exception handling routine executes.
- If no trap is taken, an appropriate value is written into the FPU destination register and execution continues.

The FPU supports the five IEEE Standard 754 exceptions:

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

Cause bits, *Enables*, and *Flag* bits (status flags) are used.

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

Figure 6-1 illustrates the *Control/Status* register bits that support exceptions.

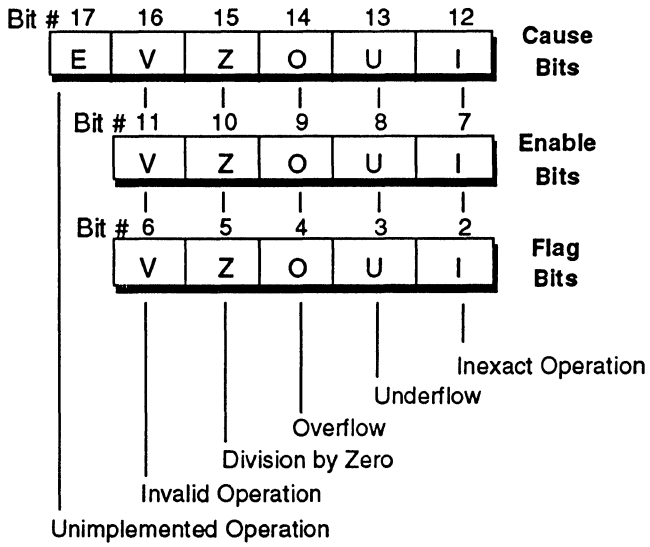


Figure 6-1 Control/Status Register Exception/Flag/Trap/Enable Bits

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, the corresponding *Cause* bit is set. If the corresponding *Enable* bit is not set, the *Flag* bit is also set. If the corresponding *Enable* bit is set, the *Flag* bit is not set and the FPU generates an interrupt to the CPU and the subsequent exception processing allows a trap to be taken.

6.2 Exception Trap Processing

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

6.3 Flags

A *Flag* bit is provided for each IEEE exception. If an IEEE exception occurs with no floating-point exception trap signaled, the *Flag* bit corresponding to the exception is set to a 1.

The *Flag* bit is reset by writing a new value into the *Status* register using the CTC1 instruction.

CTC1 is the only instruction that can clear a *Flag* bit to a 0.

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

Table 6-1 lists the default action taken by the FPU for each of the IEEE exceptions.

Table 6-1 Default FPU Exception Actions

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

†. An underflow result does not cause an Underflow exception in the CPU. If the result of an operation underflows, one of the following occurs: If Underflow and Inexact exceptions are both disabled and the FP *Control/Status* register *FS* bit is set, the Underflow exception is masked and not taken; otherwise an Unimplemented exception is taken.

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 6-2 lists the exception-causing situations and contrasts the behavior of the FPU with the requirements of the IEEE Standard 754.

Table 6-2 FPU Exception-Causing Conditions

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

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

‡. Exponent underflow sets the U and I *Cause* bits if both the U and I *Enable* bits are not set and the *FS* bit is set; otherwise exponent underflow sets the E *Cause* bit.

6.4 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 one of the following occurs:

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

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 (qNaN). 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, when one or both operands is a signaling NaN. A move (MOV) operation is not considered to be an arithmetic operation, but absolute value (ABS) and negate (NEG) are.
- Comparison or a Convert From Floating-point Operation on 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 for routines to handle these cases.

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

Trap Disabled Results: A quiet NaN is delivered to the destination register if no other software trap occurs.

Divide-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 signals an Inexact exception.)

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

Trap Disabled Results: The result, when no trap occurs, is determined by the rounding mode and the sign of the intermediate result (as listed in Table 6-1).

Underflow Exception (U)

Two related events contribute to the Underflow exception:

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

IEEE Standard 754 allows a variety of ways to detect these events, but requires they be detected the same way for all operations.

Tininess can be detected by one of the following methods:

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

The MIPS architecture requires that tininess be detected after rounding.

Loss of accuracy can be detected by one of the following methods:

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

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

Trap Enabled Results: If Underflow or Inexact traps are enabled, or if the *FS* bit is not set, then an Unimplemented exception (E) is generated, and the result register is not modified.

Trap Disabled Results: If Underflow and Inexact traps are not enabled and the *FS* bit is set, the result is determined by the rounding mode and the sign of the intermediate result (as listed in Table 6-1).

Unimplemented Instruction Exception (E)

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

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

- Denormalized operand, except for Compare instruction
- Quiet Not a Number operand, except for Compare instruction
- Denormalized result or Underflow, when either Underflow or Inexact *Enable* bits are set or the *FS* bit is not set.
- Reserved opcodes
- Unimplemented formats
- Operations which are invalid for their format (for instance, CVT.S.S)

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

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

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

Trap Disabled Results: This trap cannot be disabled.

6.5 Saving and Restoring State

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

When state is restored, state information in the *Control/Status* register indicates the exceptions that 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.

† 32 doublewords if the FR bit is set to 1.

6.6 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.

V_R4300 Processor Signal Descriptions

7

This chapter describes the signals used by and in conjunction with the V_R4300 processor. The signals include the System interface, the Clock/Control interface, the Interrupt interface, the Joint Test Action Group (JTAG) interface, and the Initialization interface.

Signals are listed in bold, and low active signals have a trailing asterisk—for instance, the low-active nonmaskable interrupt signal is **NMI***. 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.

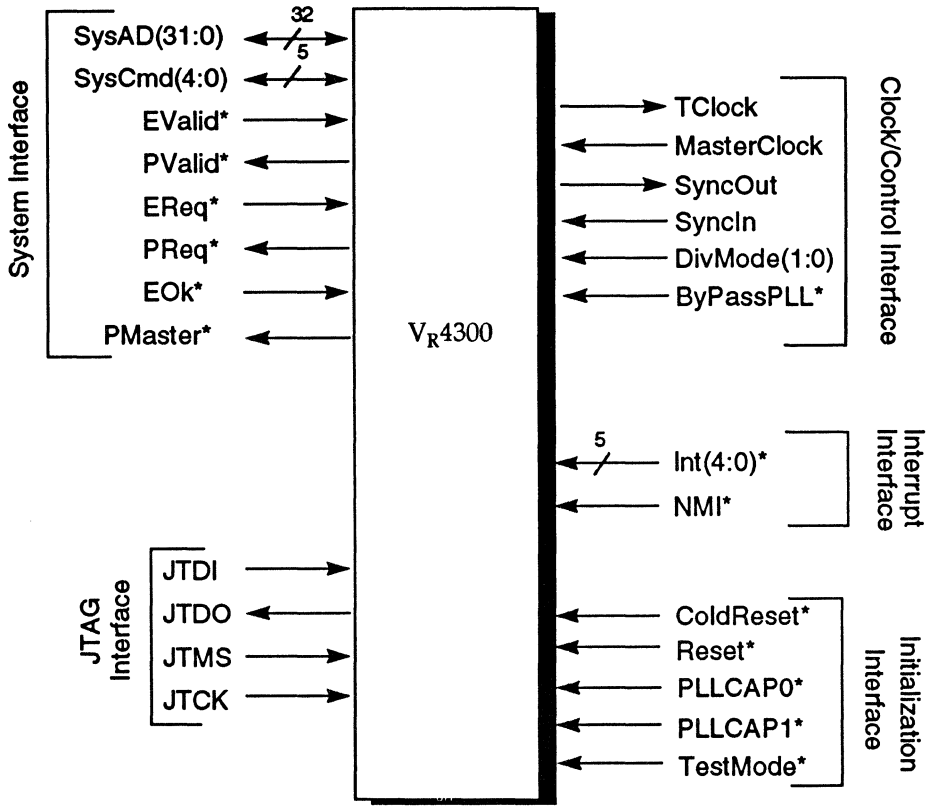


Figure 7-1 V_R4300 Processor Signals

7.1 System Interface Signals

System interface signals provide the connection between the V_R4300 processor and the other components in the system.

Table 8-1 lists the system interface signals.

Table 7-1 System Interface Signals

Name	Definition	Direction	Description
SysAD(31:0)	System address/ data bus	Input/ Output	A 32-bit address and data bus for communication between the processor and an external agent.
SysCmd(4:0)	System command/data identifier	Input/ Output	A 5-bit bus for command and data identifier transmission between the processor and an external agent.
EValid*	Valid input	Input	The external agent asserts **EValid*** when it is driving a valid address or data on the SysAD bus and a valid command or data identifier on the SysCmd bus.
PValid*	Valid output	Output	The processor asserts **PValid*** when it is driving a valid address or data on the SysAD bus and a valid command or data identifier on the SysCmd bus.
EReq*	External request	Input	An external agent asserts **EReq*** to request use of the System interface.
PReq*	Processor request	Output	Processor asserts **PReq*** to request use of the System interface.
EOK*	External agent ready	Input	The external agent asserts **EOK*** to indicate that it can accept a processor request.
PMaster*	Processor master	Output	Indicates the processor is master of the System interface bus.

7.2 Clock/Control Interface Signals

The Clock/Control interface signals make up the interface for clocking and clock synchronization. Table 8-2 lists the Clock/Control interface signals.

Table 7-2 Clock/Control Interface Signals

Name	Definition	Direction	Description
MasterClock	Master clock	Input	Master clock input that establishes the processor operating frequency.
TClock	Transmit clock	Output	Transmit clock at the operational frequency of the System interface.
SyncOut	Synchronization clock out	Output	Synchronization clock output. Must be connected to **SyncIn** through an interconnect that models the interconnect between **MasterOut**, **TClock**, **RClock**, and the external agent.
SyncIn	Synchronization clock in	Input	Synchronization clock input.
BypassPLL*	Bypass the phase-locked loop	Input	Forces **MasterClock** to bypass the PLL and feed directly to the clock buffers, and is used for testing only. This signal is implemented as a non-bonding pad (deasserted by default), and may not be a pin in the final production package. However, it will remain a part of the JTAG scan chain.
DivMode(1:0)	Set the operational frequency of the System interface	Input	Set the **PClock**-to-**MasterClock** ratios. See the description of **DivMode(1:0)** in Chapter 11 for specific settings

7.3 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.

Table 7-3 Initialization Interface Signals

Name	Definition	Direction	Description
Reset*	Soft (Warm) Reset	Input	This signal is asserted synchronously or asynchronously for a soft reset.†
ColdReset*	Cold Reset	Input	This signal indicates to the processor that the +3.3V power supply is stable and the processor should initiate a cold reset sequence, resetting the PLL.
PLLCAP0*	PLL capacitor	Input	A capacitor is connected between PLLCAP0 and the clock VssP to ensure the proper operation of the phase-locked loop.
PLLCAP0*	PLL capacitor	Input	A capacitor is connected between PLLCAP1 and the clock VccP to ensure the proper operation of the phase-locked loop.
TestMode*	Cache test mode	Input	Use for cache testing. This signal must be connected to Vcc during normal operation. This pin is part of the JTAG scan chain.

†. A soft (or warm) reset restarts processor, but does not affect clocks; it preserves the processor internal state. A description of soft reset is given in Chapter 8.

7.4 Interrupt Interface Signals

The Interrupt interface signals make up the interface used by external agents to interrupt the V_R4300 processor. Table 8-3 lists the Interrupt interface signals.

Table 7-4 Interrupt Interface Signals

Name	Definition	Direction	Description
Int*(4:0)	Interrupt	Input	Five general processor interrupts, bit-wise ORed with bits 4:0 of the *Interrupt* register, and visible as bits 14:10 of the *Cause* register
NMI*	Nonmaskable interrupt	Input	Nonmaskable interrupt, ORed with bit 6 of the interrupt register.

7.5 JTAG Interface Signals

The JTAG interface signals make up the interface that provides the JTAG boundary scan mechanism. Table 8-4 lists the JTAG interface signals.

Table 7-5 JTAG Interface Signals

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

Initialization Interface

8

This chapter describes the V_R4300 Initialization interface, and the processor modes. This includes the reset signal description and types, and initialization sequence, with signals and timing dependencies, and the user-selectable V_R4300 processor modes.

Signal names are listed in bold letters—for instance the signal **MasterClock** indicates the processor clock. Low-active signals are indicated by a trailing asterisk, such as **ColdReset***, the power-on/cold reset signal.

8.1 Functional Overview

The V_R4300 processor has the following two types of resets; they use the **ColdReset*** and **Reset*** input signals.

- **Cold Reset** is asserted after the power supply is stable and then restarts all clocks. 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.

After reset, the processor is bus master and drives the **SysAD** bus.

Changes from the V_R4000

The V_R4300 processor differs from the V_R4000 in that only sixteen cycles of **Reset*** assertion are required. In addition, there is no boot-mode interface in the V_R4300 processor; all available modes are directly controlled by pins on the package (for instance, **DivMode** is available for clock definition) or by software through the *Config* register (for instance, Endianness, and Data Rate).

Changes from the V_R4200

The V_R4300 also differs also from the V_R4200 processor in setting up the *Config* register mode bits. What are external pins on V_R4200 are software-controlled *Config* register bits on the V_R4300, set to a default value with the deassertion of Cold Reset and unaffected by a Warm Reset or NMI exception. Upon a Cold Reset, the default modes of the V_R4300 are BigEndian and Data Rate = D. Since instruction fetches are a single word wide, the processor can fetch instructions even from a Little Endian system memory and the first instructions could reset the *BE* bit in the *Config* register to the system endianness mode. The *EP* field of the *Config* register, which specifies the data rate, should also be set for those systems that only support the slow mode, before any write instruction is executed.

System Coordination

Care must be taken to coordinate system reset with other system elements. In general, bus errors immediately before, during, or after a reset may result in unpredicted behavior. Also, a small amount of processor state is guaranteed as stable after a reset of the V_R4300 processor, so extreme care must be taken to correctly initialize the processor through software.

The operation of each type of reset is described in sections that follow. Refer to Figures 8-1 and 8-2 later in this chapter for timing diagrams of the cold and warm resets.

8.2 Reset Signal Description

This section describes the two reset signals, **ColdReset*** and **Reset***.

ColdReset*: the **ColdReset*** signal must be asserted† (low) to reset the processor. **TClock** begins to cycle and is synchronized with the deasserted edge (high) of **ColdReset***. **ColdReset*** can be asserted and deasserted asynchronously with the rising edge of **MasterClock**.

Reset*: the **Reset*** signal is asserted synchronously to initiate a warm reset. The **Reset*** signal must be deasserted synchronously with **MasterClock**.

† *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 **BigEndian** signal is true when it is asserted high.

Cold Reset

A cold reset is used to completely reset the processor, including processor clocks. Information about saving processor states is given below.

Once power to the processor is established, the **ColdReset*** signal is asserted for 64,000 **MasterClock** cycles to ensure time for the processor clocks to lock to the input **MasterClock**. **ColdReset*** can be asserted and deasserted asynchronously with the rising edge of **MasterClock**.

Reset* must be asserted whenever **ColdReset*** is asserted; **Reset*** must remain asserted for 16 cycles after the deassertion of **ColdReset***.

The mode pins (**BypassPLL*** and **TestMode***) are latched one cycle after deassertion of **ColdReset***.

Upon reset, the processor becomes bus master and drives the **SysAD** bus. After **Reset*** is deasserted, the processor branches to the Reset exception vector and begins executing the Cold Reset exception code.

State bits saved during a Cold Reset are listed in Chapter 4.

Warm Reset

A warm reset is used to reset the processor without affecting the clocks; in other words, a warm reset is a logic reset.

Assertion of the **Reset*** signal resets the processor without disrupting the clocks, and allows the processor to retain as much of its state as possible for debugging. (For information about saving processor states, see the description of the Soft Reset exception in Chapter 4.) Since a warm reset takes effect immediately upon assertion of the **Reset*** signal, multicycle operations such as a cache miss or a floating-point instruction may be aborted with the result of some loss of data.

A warm reset is started by assertion of the **Reset*** pin. **Reset*** must be asserted for a minimum of 16 cycles, and must be asserted and deasserted synchronously with **MasterClock**. In general, data in the processor is preserved for debugging purposes.

Upon reset, the processor becomes bus master and drives the **SysAD** bus. After **Reset*** is deasserted, the processor branches to the Reset exception vector and begins executing the reset exception code.

If **Reset*** is asserted in the middle of a **SysAD** transaction, care must be taken to reset all external agents to avoid **SysAD** bus contention.

Figures 8-1 and 8-2 show the timing diagrams for the cold and warm resets.

Cold Reset

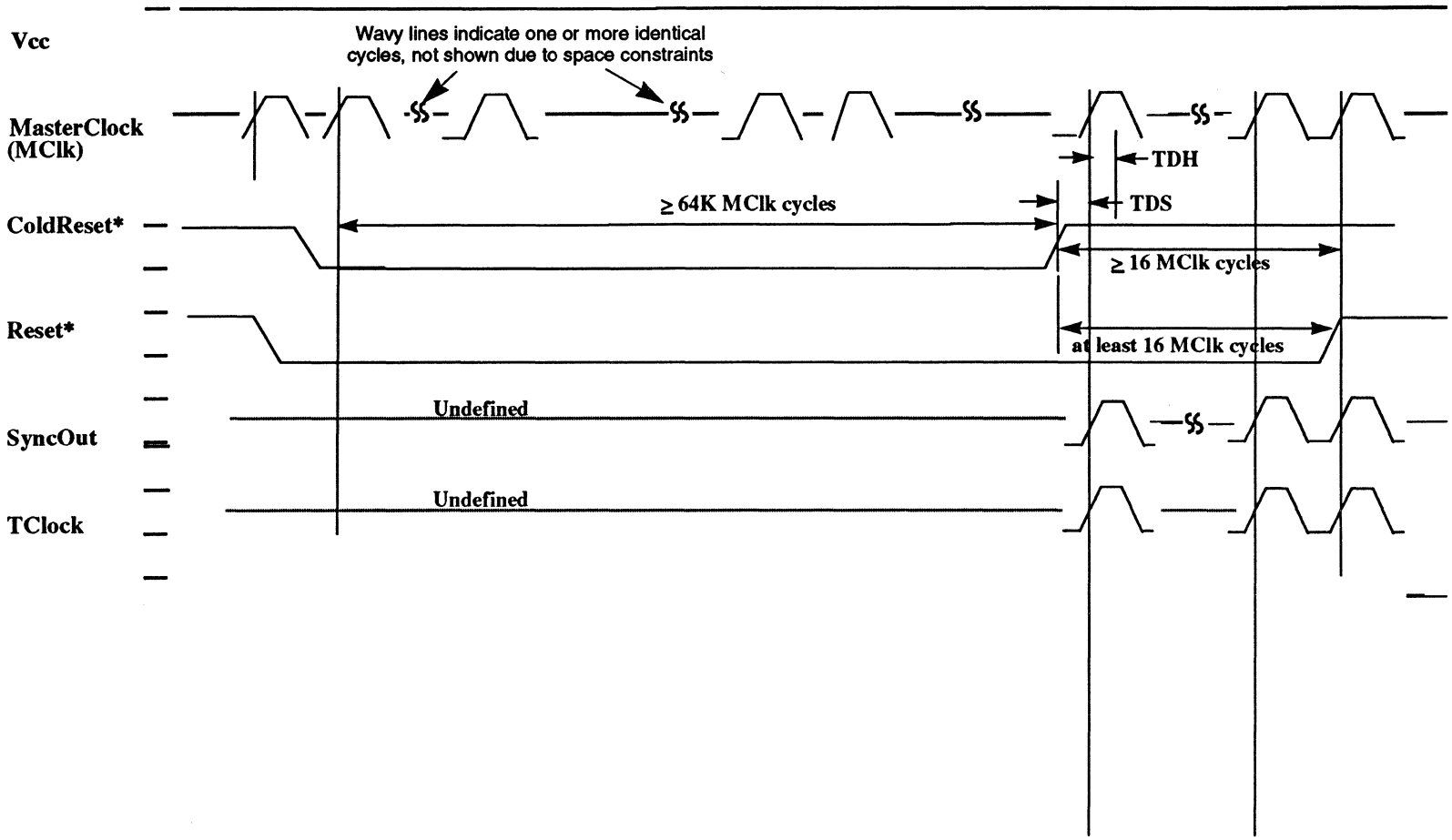


Figure 8-1 Cold Reset

Warm Reset

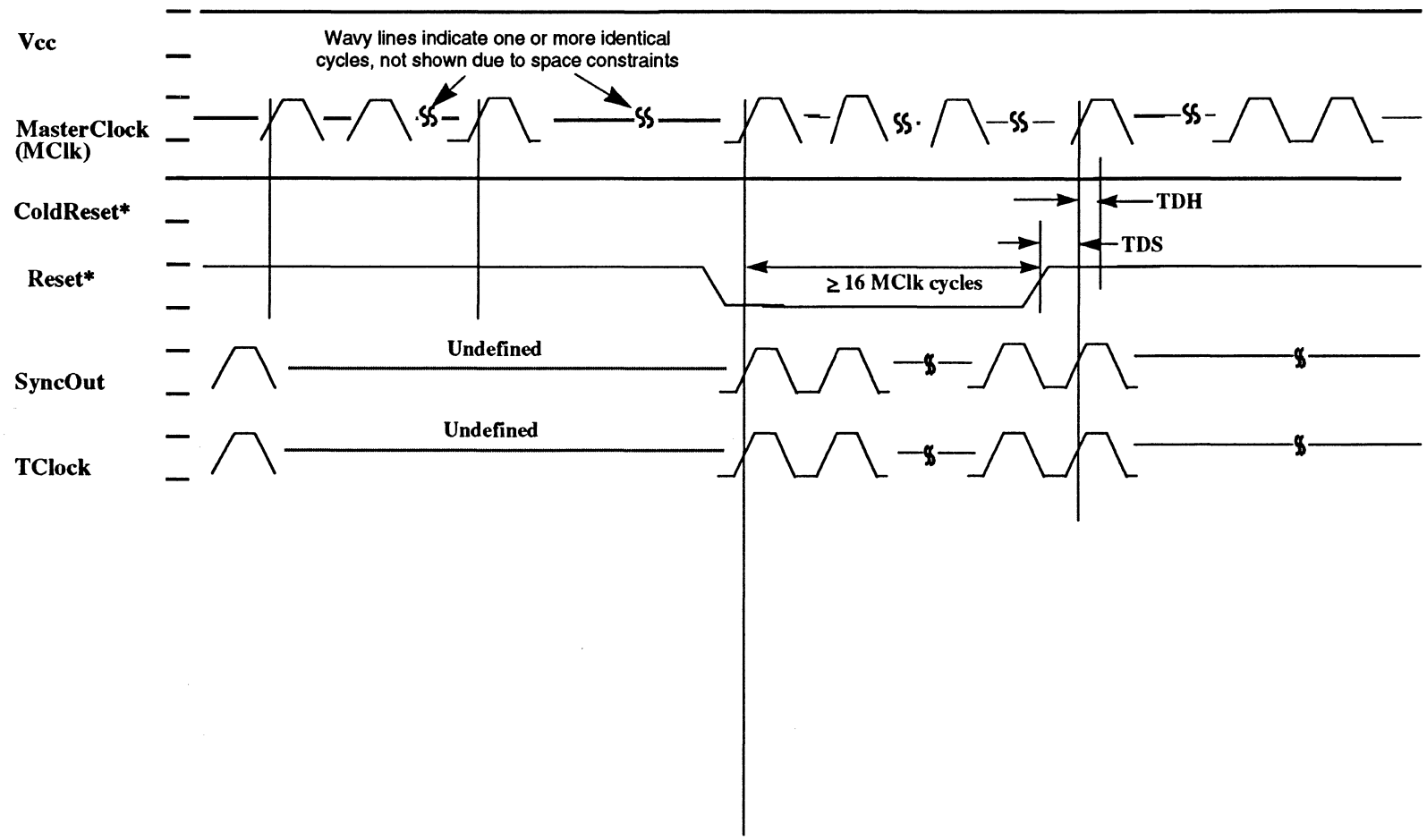


Figure 8-2 Warm Reset

8.3 V_R4300 Processor Modes

The V_R4300 processor supports several user-selectable modes. All modes except **BypassPLL** and **DivMode** ratios are set/reset by writing to the *Status* and *Config* registers.

Power Modes

The V_R4300 supports three power modes: normal power, reduced power, and power-off. This section describes these three modes.

Normal Power Mode

Normally the V_R4300 processor clock (**PClock**) operates at a multiple of the **MasterClock** speed (3, 2, 1.5 or 1, as designated by the **DivMode** setting). The System Interface Clock (**SClock**) operates at the same frequency as **MasterClock.**

Reduced Power Mode

The user may set the processor to reduced power mode through a move to the *Status* register that sets the *RP* bit. In RP mode, the processor stalls the pipeline and goes into a quiescent state—the store buffers are empty, all write-back cycles are completed, and all cache misses are resolved. The processor then slows **PClock** to one of the following frequencies:

3/4, 1/2, 1.5/4, or 1/4 of the **MasterClock** speed.

SClock and **TClock** are also one of the following divisions of their normal frequency: 3/4, 1/2, 1.5/4, or 1/4 of the **MasterClock** speed. Clocks switch frequency within 16 **MasterClock** cycles after the MTC0 instruction is executed.

Default setting for this mode is normal (undivided) clock speed, and the processor returns to this normal state after a Cold Reset.

Software must guarantee the proper operation of the system upon setting or clearing the *RP* bit. Software does this by first writing new values to any registers in external agents that must be changed to accommodate the change in frequency (for instance, DRAM refresh counters). Software then sets the System interface to an inactive state (for instance, by an uncached read, which empties the write buffer upon completion of the read). Only then can software attempt to execute the instruction to set or clear the *RP*

bit. Finally, software must make certain that at least eight instructions immediately preceding and following the Move to Coprocessor register do not cause a cache miss, TLB miss, or exception of any kind.

Power Off Mode

Before entering power off mode, the system retains as much information as possible by writing out to memory the Program Counter, along with the contents of the CP0, general, and floating-point registers. Dirty data cache lines are also written out to memory.

Privilege Modes

The V_R4300 processor supports three modes of system privilege: kernel, supervisor, and user extended addressing. This section describes these three modes.

Kernel Extended Addressing

If the *KX* bit in the *Status* register is set, it enables MIPS III opcodes in Kernel mode and causes TLB misses on kernel addresses to use the Extended TLB Refill exception vector.

Supervisor Extended Addressing

If the *SX* bit in the *Status* register is set, it enables MIPS III opcodes in Supervisor mode and causes TLB misses on supervisor addresses to use the Extended TLB Refill exception vector.

User Extended Addressing

If the *UX* bit in the *Status* register is set, it enables MIPS III opcodes in User mode and causes TLB misses on user addresses to use the Extended TLB Refill exception vector. If the bit is clear, it enables MIPS II opcodes and 32-bit address translation.

Floating-Point Registers

By setting the *FR* bit in the *Status* register, the user accesses the full set of 32 64-bit floating point registers as defined in the MIPS III ISA. When reset, the processor accesses the registers as defined in the MIPS II architecture.

System Endianness

By setting the *BE* bit in the *Config* register, the state of system endianness is provided to the processor. It is recommended that the *BE* bit be changed directly after power up, during processor initialization, and before any nonword memory access operations.

This mode is set to big-endian on Cold Reset.

Reverse Endianess

When the *RE* bit in the *Status* register is set, endianess as seen by user software is reversed from its previous state.

Instruction Trace Support

The V_R4300 processor allows the user to track branches or jumps by setting the *ITS* bit in the *Status* register. ITS mode can be disabled by resetting the *ITS* bit.

Bootstrap Exception Vector

This bit is used when diagnostic tests cause exceptions to occur prior to verifying proper operation of the cache and main memory system.

When set, the Bootstrap Exception Vector (*BEV*) bit in the *Status* register causes the TLB refill exception vector to be relocated to a virtual address of 0xFFFF FFFF BFC0 0200 and the general exception vector relocated to address 0xFFFF FFFF BFC0 0380.

When *BEV* is cleared, these vectors are located at 0xFFFF FFFF 8000 0000 (TLB refill) and 0xFFFF FFFF 8000 0180 (general).

Interrupt Enable

When this bit is clear, interrupts are not allowed, with the exception of reset and the nonmaskable interrupt.

Clock Interface

9

This chapter describes the clock signals ("clocks") used in the V_R4300 processor.

The subject matter includes basic system clocks, system timing parameters, operating the V_R4300 processor in reduced power (RP) mode, connecting clocks to a phase-locked system, and connecting clocks to a system without phase locking.

9.1 Signal Terminology

The following terminology is used in this chapter (and 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*).

Figures 9-1 and 9-2 illustrate these terms.

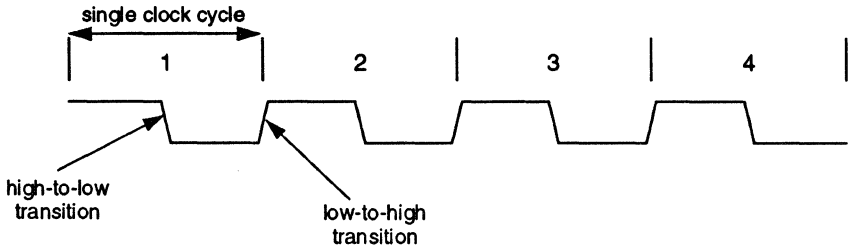


Figure 9-1 Signal Transitions

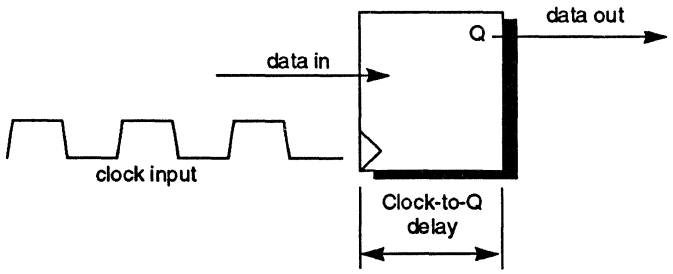


Figure 9-2 Clock-to-Q Delay

9.2 Basic System Clocks

The various clock signals used in the V_R4300 processor are described below, starting with **MasterClock**, upon which the processor bases all internal and external clocking.

The clocks on the V_R4300 processor are controlled by an on-processor Phase-locked Loop (PLL) circuit. This circuit keeps the V_R4300 processor's internal clock edges aligned with the clock edges of the **MasterClock** signal, which itself acts as the master system clock.

Inside the V_R4300 processor, the **MasterClock** signal can be multiplied by a factor set by the **DivMode(1:0)** inputs to the processor. All internal clocks are then derived from this clock. The V_R4300 processor has two primary internal clocks, the pipeline (also referred to as *processor*) clock, **PClock**, and the system interface clock, **SClock**. While other internal clock edges may be generated, they are transparent to the user. **SClock** has the same frequency and phase as **MasterClock**.

The processor also provides an external clock for use by system designers and other users of the V_R4300 processor. This clock is the transmit clock signal, **TClock**. **TClock** is aligned with **SClock**, and has the same frequency.

MasterClock

The processor bases all internal and external clocking on the single **MasterClock** input signal. **MasterClock** specifications are shown in Figure 9-3.

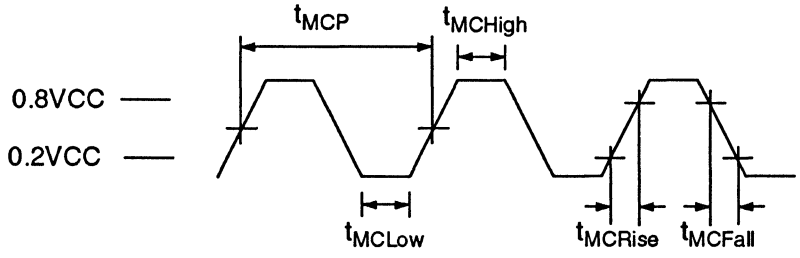


Figure 9-3 MasterClock

SyncIn/SyncOut

The processor generates **SyncOut** at the same frequency as **MasterClock** and aligns the output of the **SyncIn** input buffer with **MasterClock**. **TClock** is generated at the same frequency as **MasterClock** and aligned with **SyncOut**.

SyncOut must be connected to **SyncIn** either directly, or through an external buffer. The processor can compensate for both output driver and input buffer delays (and, when necessary, delay caused by an external buffer) when aligning **SyncIn** with **MasterClock**. Figure 9-9 gives an illustration of **SyncOut** connected to **SyncIn** through an external buffer.

PClock

The pipeline (or *processor*) clock, **PClock**, can be 1, 1.5, 2 or 3 times the **MasterClock** frequency. This multiplication factor is determined by **DivMode(1:0)** pins, which are static signal inputs to V_R4300. In reduced power mode—when the *RP* bit in the *Status* register is set—**PClock** is driven at one-half of the **MasterClock** frequency, which is one-fourth the normal **PClock** speed.

All internal registers and latches use **PClock**.

SClock

The system interface clock, **SClock**, is the same as the **MasterClock** frequency. **SClock** is always derived from **PClock**; thus, when **PClock** is in reduced power mode, **SClock** frequency is also reduced by a factor of four. The V_R4300 processor drives its outputs on this clock edge.

The first rising edge of **SClock**, after **ColdReset*** is deasserted, is aligned with the first rising edge of **MasterClock**.

TClock

TClock (transmit clock) clocks the output registers of an external agent, and can be a global system clock for any other logic in the external agent. Specifications for **TClock** are shown in Figure 9-4.

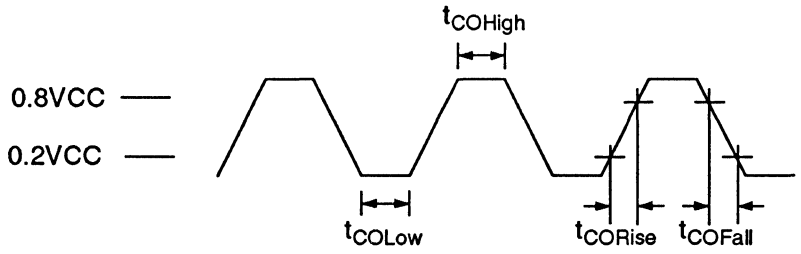


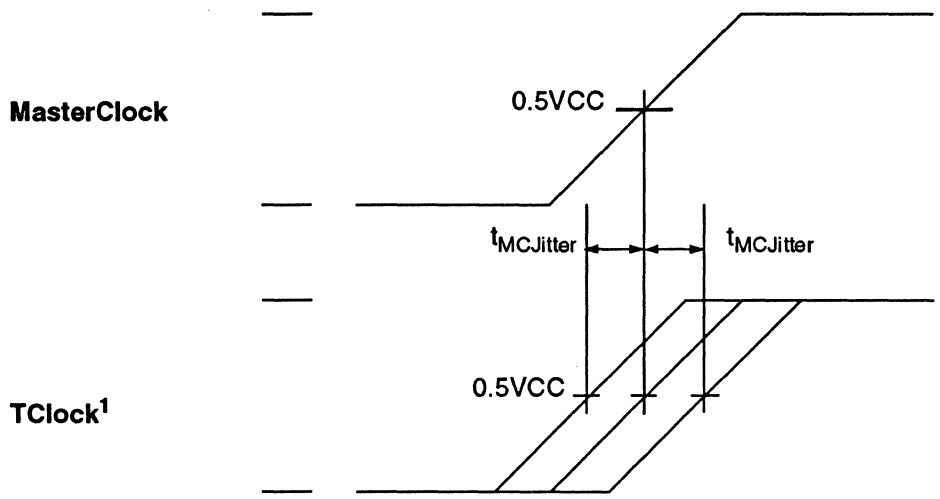
Figure 9-4 TClock

TClock is the same frequency as **SClock**. When **SyncIn** is shorted to **SyncOut**, the edges of **TClock** align precisely with the edges of **SClock** and **MasterClock**.

When a delay is added between **SyncIn** and **SyncOut**, the **TClock** at the pins leads **SClock** (and thus **MasterClock**) by the same amount of delay.

If the delay between **SyncIn** and **SyncOut** is matched to an external delay between **TClock** at the processor and **TClock** at the external logic, the **TClock** at the external logic aligns to **SClock** and **MasterClock**.

Figure 9-5 shows the clock jitter tolerances for **TClock**.



With **SyncOut** shorted to **SyncIn** by the shortest path, the 50% point of **TClock** lines up with the 50% point of **MasterClock.**

Note: The **SyncIn/SyncOut** path and **TClock** must have the same capacitive loading to match the use of **TClock** with the **MasterClock** edges.

Figure 9-5 Clock Jitter

PClock-to-SClock Division

Figure 9-6 shows the clocks for a PClock-to-SClock division by 2.

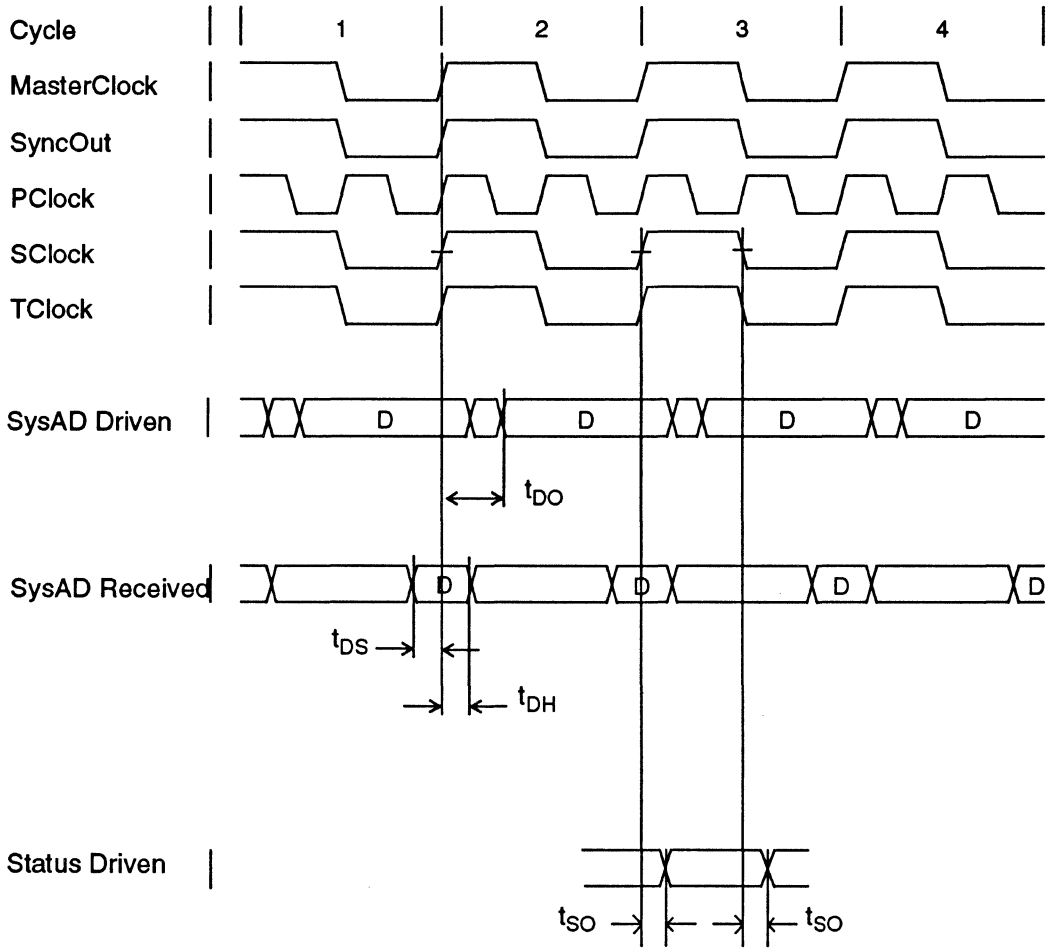


Figure 9-6 Processor Clock, PClock-to-SClock Divisor of 2

Phase-Locked Loop (PLL)

The V_R4300 clocks are controlled by a Phase-locked Loop circuit (PLL). The PLL can be disabled by a **BypassPLL*** pin. In bypass mode, the internal **PClock** and **SClock** are derived by dividing the clock driven at the **MasterClock** pin of the processor. This mode can be used for test purposes, as well as in systems whose **MasterClock** is running too slowly for the phase-locked loop to synchronize.

9.3 Operating the V_R4300 Processor in Reduced Power Mode

The V_R4300 processor normally operates in a mode where the processor clock (**PClock**) operates at a multiple of the **MasterClock** speed (either 3, 2, 1.5, or 1 times the **MasterClock** speed, as selected by **DivMode(1:0)** pins). The System interface clock (**SClock**) operates at the same frequency as **MasterClock**.

The user may initiate Reduced Power mode by executing a move to the *Status* register that sets the *RP* bit. Upon setting this mode, the processor modifies its clocking scheme to slow the PClock to a divisor of four of its normal frequency (that is, 3/4, 1/2 1.5/4 or 1/4 of the **MasterClock** speed). **SClock** and **TClock** are then also derived as one fourth of their normal frequency, (1/4 of the **MasterClock** speed). The clocks switch frequency within 16 **MasterClock** cycles after the move to the *Status* register is executed.

Reduced Power mode allows the user to selectively reduce power when the system is not being heavily used. This feature reduces the power consumed by the processor chip to 25% of its normal value.

The default of this mode is *normal clocking*. The chip returns to normal clocking after a Cold Reset.

Software must be careful to execute a code sequence that will guarantee the proper operation of the system upon setting or clearing the *RP* bit. Software must first write any registers on the external agents that must be changed to accommodate the change in frequency (for instance, DRAM refresh counters). Next, software must guarantee that the system interface is in an inactive state. This can be accomplished by executing an uncached read, which will guarantee the flush buffer is empty upon completion of the read. Only then can software attempt to execute the instruction which sets or clears the *RP* bit. Finally, software should guarantee that at least the eight instructions immediately preceding and following the Move to Coprocessor register does not cause a cache miss, TLB miss, or exception of any kind.

9.4 Connecting Clocks to a Phase-Locked System

When the processor is used in a phase-locked system, the external agent must phase lock its operation to a common **MasterClock**. In such a system, the delivery of data and data sampling have common characteristics, even if the components have different delay values. For example, *transmission time* (the amount of time a signal takes to move from one component to another along a trace on the board) between any two components A and B of a phase-locked system can be calculated from the following equation:

Transmission Time = (SClock period) – (t_{DO} for A) – (t_{DS} for B) –
(Clock Jitter for A Max) – (Clock Jitter for B Max)

Figure 9-7 shows a block-level diagram of a phase-locked system using the V_R4300 processor.

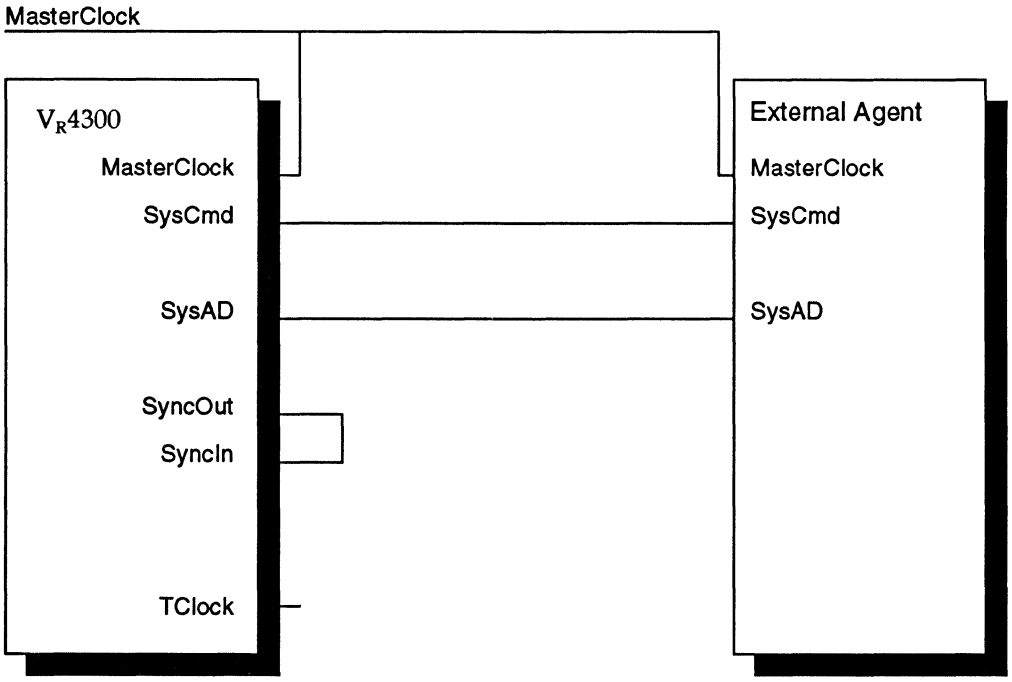


Figure 9-7 The V_R4300 Processor Phase-Locked System

9.5 Connecting Clocks to a System without Phase Locking

When the V_R4300 processor is used in a system in which the external agent cannot lock its phase to a common **MasterClock**, the output clock **TClock** can clock the remainder of the system. Two clocking methodologies are described in this section: connecting to a gate-array device or connecting to discrete CMOS logic devices.

Connecting to a Gate-Array Device

When connecting to a gate-array device, **TClock** is used within the gate array.

Figure 9-8 is a block diagram of a system without phase lock, using the V_R4300 processor with an external agent implemented as a gate array.

These sampling registers should be immediately followed by staging registers clocked by an internally buffered version of **TClock**. This buffered version of **TClock** should be the global system clock for the logic inside the gate array and the clock for all registers that drive processor inputs.

Staging registers place a constraint on the sum of the clock-to-Q delay of the sample registers and the setup time of the staging registers inside the gate arrays, as shown in the following equation:

Clock-to-Q Delay + Setup of Staging Register
– (Maximum Delay Mismatch for Internal Clock Buffers on TClock) } ð 0.25 (TClock period)

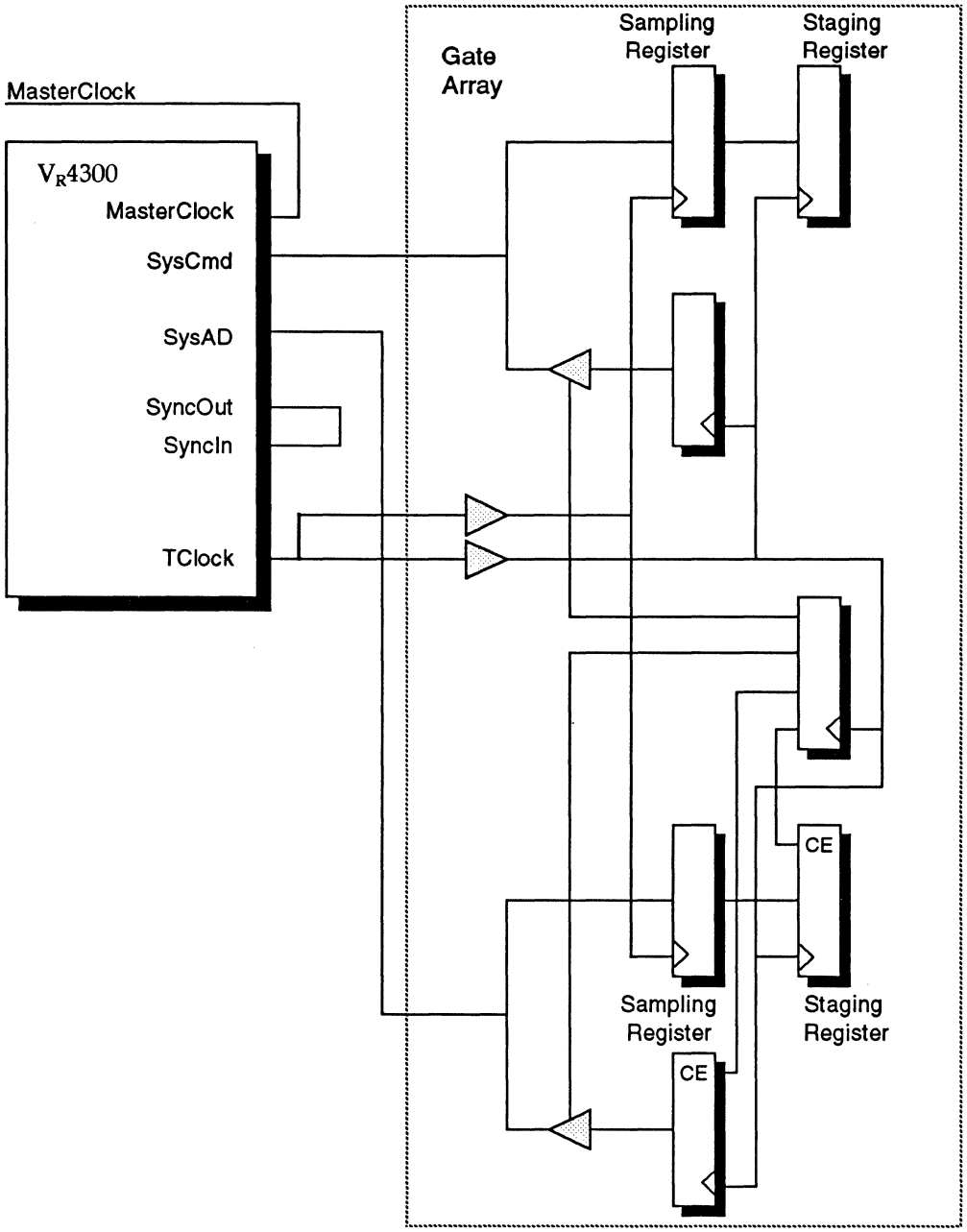


Figure 9-8 Gate-Array System without Phase Lock, using the V_R4300 Processor

In a system without phase lock, the transmission time for a signal *from* the processor *to* an external agent composed of gate arrays can be calculated from the following equation:

Transmission Time = (75 percent of TClock period) – (t_{DO} for V_R4300)
+ (Minimum External Clock Buffer Delay)
– (External Sample Register Setup Time)
– (Maximum Clock Jitter for V_R4300 Internal Clocks)

The transmission time for a signal *from* an external agent composed of gate arrays *to* the processor in a system without phase lock can be calculated from the following equation:

Transmission Time = (TClock period) – (t_{DS} for V_R4300)
– (Maximum External Clock Buffer Delay)
– (Maximum External Output Register Clock-to-Q Delay)
– (Maximum Clock Jitter for TClock)
– (Maximum Clock Jitter for V_R4300 Internal Clocks)

Connecting to a CMOS Logic System

The processor uses matched delay clock buffers to generate aligned clocks to external CMOS logic. A matched delay clock buffer is inserted in the **SyncOut/SyncIn** alignment path of the processor, skewing **SyncOut**, **MasterOut**, and **TClock** to lead **MasterClock** by the buffer delay amount, while leaving **PClock** aligned with **MasterClock**.

The remaining matched delay clock buffers are available to generate a buffered version of **TClock** aligned with **MasterClock**. Alignment error of this buffered **TClock** is the sum of the maximum delay mismatch of the matched delay clock buffers, and the maximum clock jitter of **TClock**.

As the global system clock for the discrete logic that forms the external agent, the buffered version of **TClock** clocks registers that sample processor outputs, as well as clocking the registers that drive the processor inputs.

The transmission time for a signal from the processor to an external agent composed of discrete CMOS logic devices can be calculated from the following equation:

Transmission Time = (TClock period) – (t_{DO} for V_R4300)
– (External Sample Register Setup Time)
– (Maximum External Clock Buffer Delay Mismatch)
– (Maximum Clock Jitter for V_R4300 Internal Clocks)
– (Maximum Clock Jitter for TClock)

Figure 9-9 is a block diagram of a system without phase lock, employing the V_R4300 processor and an external agent composed of both a gate array and discrete CMOS logic devices.

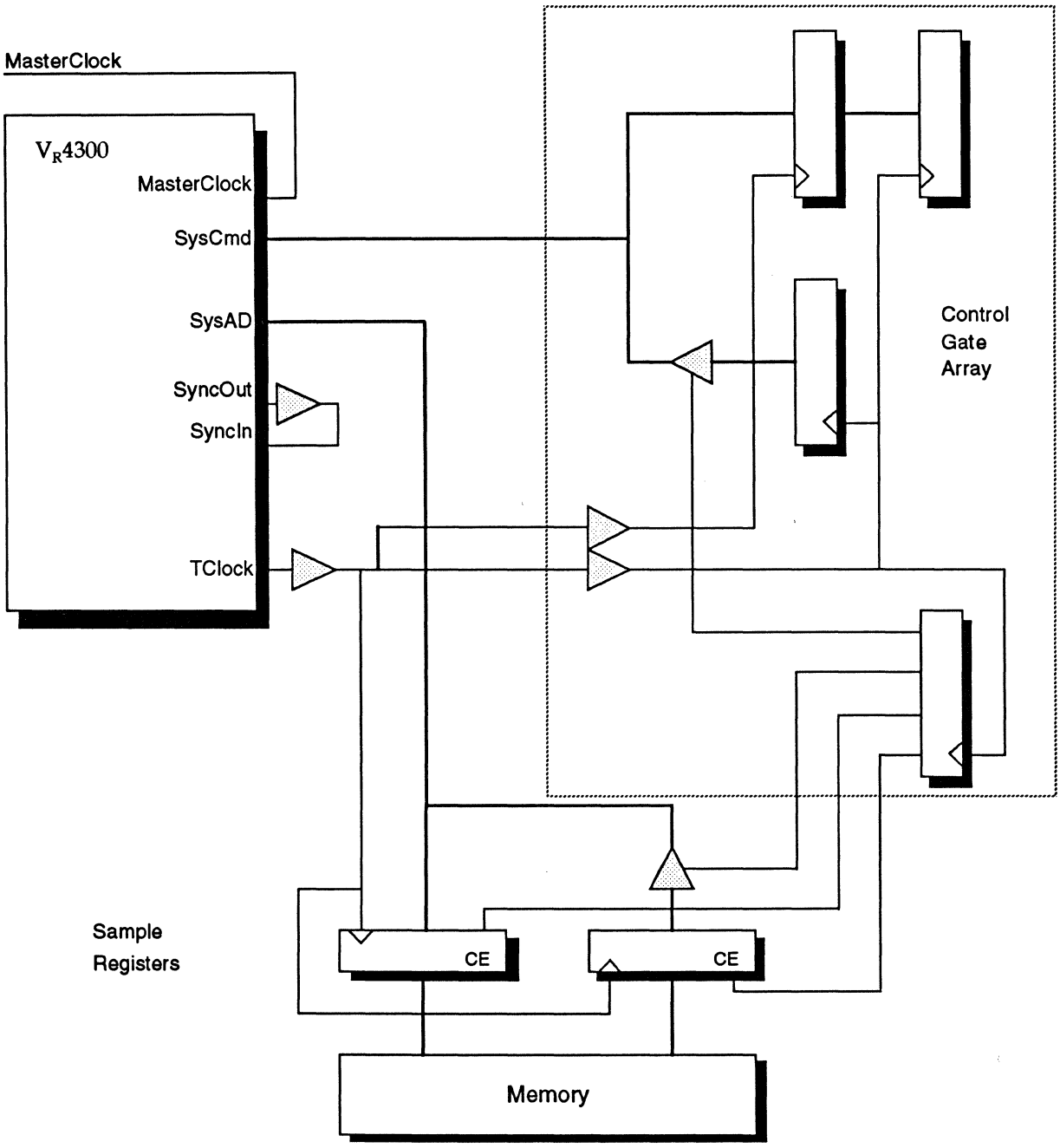


Figure 9-9 Gate Array and CMOS System without Phase Lock, using the V_R4300 Processor

The transmission time for a signal from an external agent composed of discrete CMOS logic devices can be calculated from the following equation:

Transmission Time = (TClock period) – (t_{DS} for V_R4300)
– (Maximum External Output Register Clock-to-Q Delay)
– (Maximum External Clock Buffer Delay Mismatch)
– (Maximum Clock Jitter for V_R4300 Internal Clocks)
– (Maximum Clock Jitter for TClock)

In this clocking methodology, the hold time of data driven from the processor to an external sampling register is a critical parameter. To guarantee hold time, the minimum output delay of the processor, t_{DM}, must be greater than the sum of:

minimum hold time for the external sampling register
+ maximum clock jitter for V_R4300 internal clocks
+ maximum clock jitter for TClock
+ maximum delay mismatch of the external clock buffers
+ transmission time

Cache Organization and Operation

10

This chapter describes in detail the cache memory: its place in the V_R4300 memory organization, and individual organization of the caches.

This chapter uses the following terminology:

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

These terms are used interchangeably throughout this book.

10.1 Memory Organization

Figure 10-1 shows the V_R4300 system memory hierarchy. In the logical memory hierarchy, both primary and secondary 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 10-1 has the capacity to hold more data than the block above it. For instance, physical main memory has a larger capacity than the caches. 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.

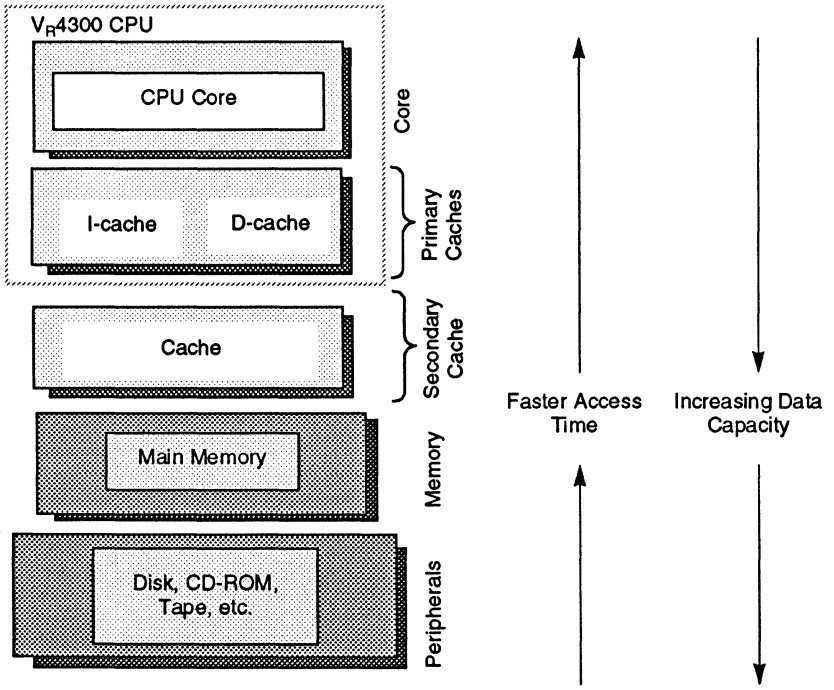


Figure 10-1 Logical Hierarchy of Memory

The V_R4300 processor has two on-chip caches: one holds instructions (the instruction cache), the other holds data (the data cache). The instruction and data caches can be read in one **PClock** cycle.

Data writes are pipelined and can complete at a rate of one per **PClock** cycle. In the first stage of the cycle, the store address is translated and the tag is checked; in the second stage, the data is written into the data RAM.

10.2 Cache Organization

This section describes the organization of the on-chip data and instruction caches. Figure 10-2 provides a block diagram of the V_R4300 cache and memory model.

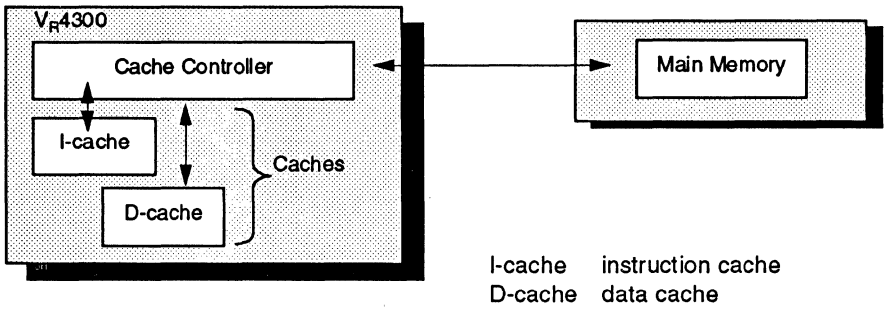


Figure 10-2 V_R4300 Cache Support

Cache Sizes

The V_R4300 instruction cache is 16 Kbytes; the data cache is 8 Kbytes.

Cache Line Lengths

A *cache line* is the smallest unit of information that can be fetched from main memory for the cache, and that is represented by a single tag.†

The line size for the instruction cache is 8 words (32 bytes) and the line size for the data cache is 4 words (16 bytes).

† Cache tags are described in the following sections.

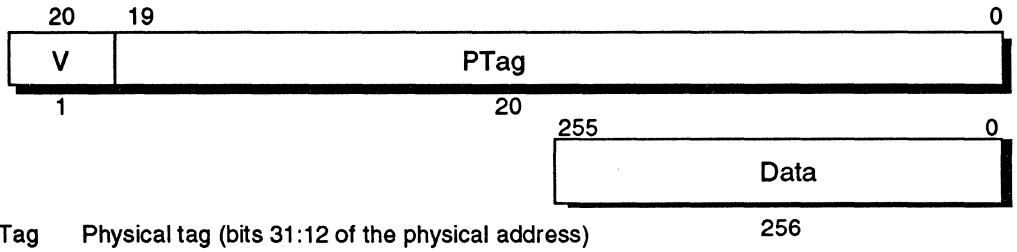
Organization of the Instruction Cache (I-Cache)

Each line of I-cache data (although it is actually an instruction, it is referred to as data to distinguish it from its tag) has an associated 21-bit tag that contains a 20-bit physical address and a single *Valid* bit.

The V_R4300 processor I-cache has the following characteristics:

- direct-mapped
- indexed with a virtual address
- checked with a physical tag
- organized with an 8-word (32-byte) cache line.

Figure 10-3 shows the format of an 8-word (32-byte) I-cache line.



PTag Physical tag (bits 31:12 of the physical address)
V Valid bit
Data Cache data

Figure 10-3 V_R4300 8-Word I-Cache Line Format

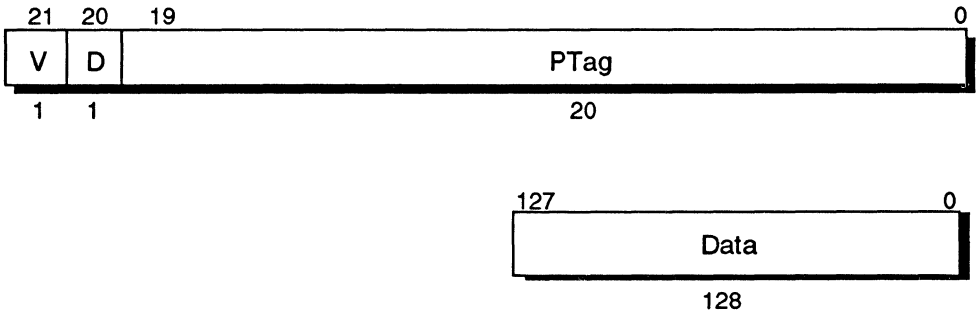
Organization of the Data Cache (D-Cache)

Each line of D-cache data has an associated 22-bit tag that contains a 20-bit physical address, a *Valid* bit and a *Dirty* bit.

The V_R4300 processor D-cache has the following characteristics:

- write-back
- direct-mapped
- indexed with a virtual address
- checked with a physical tag
- organized with a 4-word (16-byte) cache line.

Figure 10-4 shows the format of a 4-word (16-byte) D-cache line.



V	Valid bit
D	Dirty bit
PTag	Physical tag (bits 31:12 of the physical address)
Data	D-cache data

Figure 10-4 V_R4300 4-Word Data Cache Line Format

Accessing the Caches

Figure 10-5 shows the virtual address (VA) index into the caches. The number of virtual address bits needed to index the the instruction and data caches depends on the cache size.

For example, VA(12:4) is needed to access an 8-Kbyte page in the data cache with its 4-word line: VA(12) addresses an 8 Kbyte space and VA(4) provides quadword resolution.

Similarly, VA(13:5) accesses a 16 Kbyte page in the instruction cache with its 8-word line: VA(5) provides octalword resolution and VA(13) addresses a 16 Kbyte space.

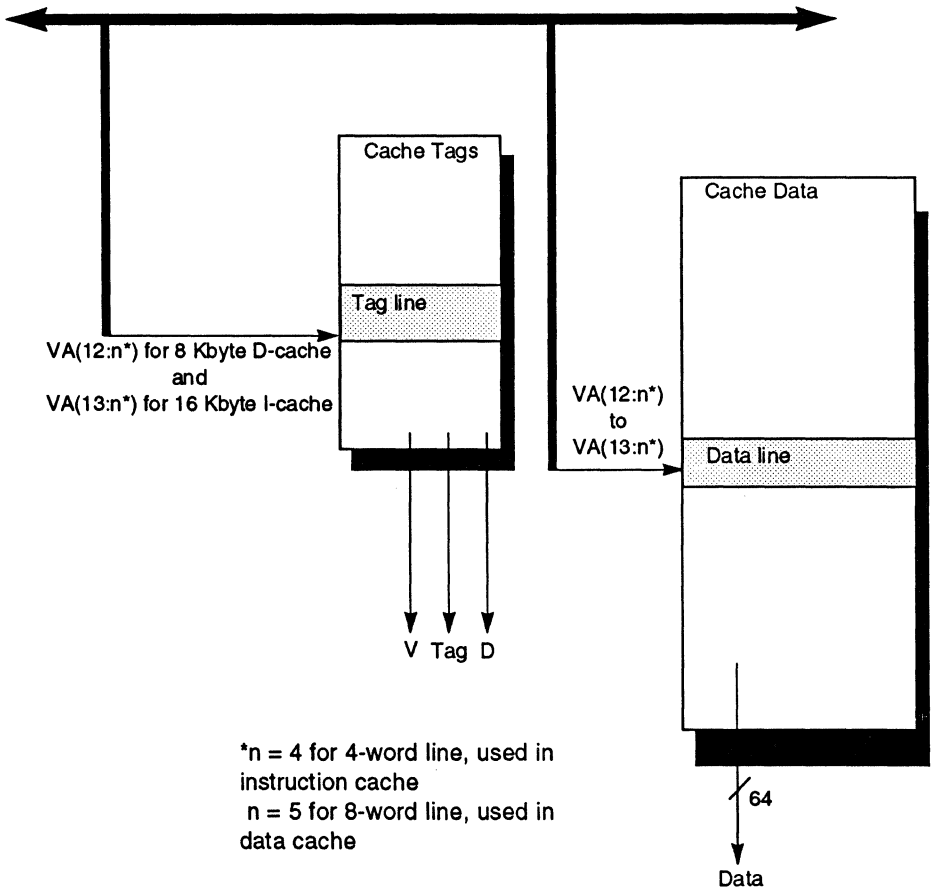


Figure 10-5 Cache Data and Tag Organization

10.3 Cache Operations

As described earlier, 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 appropriate cache.
2. The cache controller checks to see if this instruction or data is present in the cache.
   - If the instruction/data is present, the processor retrieves it. This is called a cache *hit*.
   - If the instruction/data is not present in the cache, the cache controller must retrieve it from memory. This is called a cache *miss*.
3. The processor retrieves the instruction/data from the cache and operation continues.

It is possible for the same data to be in two places simultaneously: main memory and cache. This data is kept consistent through the use of a *write-back* methodology; that is, modified data is not written back to memory until the cache line is to be replaced.

The V_R4300 processor supports all MIPS V_R4400 PC, V_R4000 PC, and V_R4200 processor cache operations.

Instruction and data cache line replacement operations are listed in described in the following sections.

Table 10-1 Cache Operations

Name	Caches	Operation
Index Invalidate	Instruction	Sets the cache state of cache block to invalid.
Index WriteBack Invalidate	Data	Examines cache state, if Valid Dirty, then that block is written back to main memory. Then the cache block is set to invalid.
Index Load Tag	Instruction & Data	Read the tag for the cache block at the specified index and place it into TagLo.
Index Store Tag	Instruction & Data	Write the tag for the cache block at the specified index from the TagLo register.
Create Dirty Exclusive	Data	If the cache does not contain the specified address, and the block is Valid Dirty the block will be written back to main memory. Then the tag will be set to the specified physical address and will be marked valid.
Hit Invalidate	Instruction & Data	If the cache block contains the specified address, cache block will be marked invalid.
Hit WriteBack Invalidate	Data	If the cache block contains the specified address, and it is Valid Dirty, the data will be written back to main memory. Then, the cache block is marked invalid.
Fill	Instruction	Fill the Instruction cache block from main memory.
Hit WriteBack	Data	If the cache block contains the specified address, and it is marked Valid Dirty, the block will be written back to main memory, and marked Valid Clean.
Hit WriteBack	Instruction	If the cache block contains the specified address, the block will be written back to main memory unconditionally.

Cache Write Policy

The V_R4300 processor manages its data cache by using a write-back policy; that is, it stores write data into the cache, instead of writing it directly to memory.† Some time later this data is independently written into memory. In the V_R4300 implementation, a modified cache line is not written back to memory until the cache line is to be replaced either in the course of satisfying a cache miss, or during the execution of a write-back CACHE instruction.

When the processor writes a cache line back to memory, it does not ordinarily retain a copy of the cache line, and the state of the cache line is changed to invalid.

Data Cache Line Replacement

Since the data cache uses a write-back methodology, a cache line load is issued to main memory on a load or store miss, as described below. After the data from memory is written to the data cache, the pipeline resumes execution.

The line replacement sequence is based on a "Critical Data Word First" scheme. The processor restarts its pipeline as soon as the memory supplies the desired word in the first doubleword of a block transfer. This sequence is summarized as follows:

1. Move the data physical address to the processor pads. At the same time, move the dirty cache line to the flush buffer, if needed.
2. Wait for a **PClock** cycle aligned on an **SClock** boundary.
3. Read the data from memory, receiving the desired doubleword first.
4. Receive remaining doubleword. For all loads, move the data to target register. For byte, halfword and word stores, it is necessary to do a read followed by a write procedure—retrieve (read) the 64-bit data, write new data to this retrieved data, then write the 64-bit data to cache. As this is being done, interlock the data cache to prevent it from being accessed by any subsequent instruction that tries to access this particular cache line.

† An alternative to this is a *write-through* cache, in which information is written simultaneously to cache and memory.

The data cache miss penalty in number of **PClock** cycles is:

```
    3
+ (Clock Aignment)
+ (Memory Access & Transfer Time
      for Critical Doubleword)
+  1
```

Rules for replacement on data load and data store misses are given below.

Data Load Miss

If the missed line is not dirty, it is replaced with the requested line.

If the missed line is dirty, it is moved to the flush buffer. The requested line replaces the missed line, and the data in the flush buffer is written to memory.

Data Store Miss

If the missed line is not dirty, the requested line is merged with the store data and written to memory.

If the missed line is dirty, it is moved to the flush buffer. The requested line is merged with the store data and written to cache, and data in the flush buffer is written to memory.

The data cache miss penalties, in number of **SClock** cycles, are given in Table 10-2.

Table 10-2 Data Cache Refill Penalty Cycle Count

Number of SClock Cycles	Action
1	Transfer address to flush buffer and wait for the pipeline start signal
0 to 1	Synchronize with **SClock** and transfer address to internal SysAD bus
1	Transfer to external SysAD bus
M	Time needed to access memory, measured in **PClock** cycles
2	Transfer the cache line from memory to the SysAD bus
1	Transfer the cache line from the external to internal bus
0	Restart the DC stage

Instruction Cache Line Replacement

For an instruction cache miss, refill is done using sequential ordering, starting from the first word of the retrieved cache line.

During an instruction cache miss, a memory read is issued. The requested line is returned from memory and written to the instruction cache. At this time the pipeline resumes execution, and the instruction cache is reaccessed.

The instruction cache miss penalty in number of **PClocks** is:

```
    4
+ (Clock Alignment)
+ (Memory Access & Transfer Time for Entire Cache Line)
+ 1
```

The replacement sequence for an instruction cache miss is:

1. Move the instruction physical address to the processor pads.
2. Wait for a **PClock** cycle, aligned with an **SClock** boundary, to occur.
3. Read the line from memory and write it out to the instruction cache array.
4. Restart the processor pipe.

The instruction cache miss penalties, in number of **SCycles**, is given in Table 10-3.

Table 10-3 Instruction Cache Refill Penalty Cycle Count

Number of SCycles	Action
1	Transfer address to flush buffer and wait for the pipeline start signal
0 to 1	Synchronize with **SClock** and transfer address to internal SysAD bus
1	Transfer to external SysAD bus
M	Time needed to access memory, measured in **PClock** cycles
8	Transfer the cache line from memory to the SysAD bus
1	Transfer the cache line from the external to internal bus
0	Restart the RF stage

10.4 Cache States

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

- **Dirty:** a cache line containing data that has changed since it was loaded from memory.
- **Clean:** a cache line that contains data that has not changed since it was loaded from memory.
- **Invalid:** a cache line that does not contain valid information must be marked invalid, and cannot be used. For example, after a Soft Reset, software sets all cache lines to invalid. A cache line in any other state than invalid is assumed to contain valid information.†

The data cache supports three cache states:

- invalid
- valid clean
- valid dirty

The instruction cache supports two cache states:

- invalid
- valid

The state of a valid cache line may be modified when the processor executes a CACHE operation.

† Cold or Soft Reset does not set the cache state to invalid. The invalidation of caches is left to software.

10.5 Cache State Transition Diagrams

The following section describes the cache state diagrams for the data and instruction caches. These state diagrams do not cover the initial state of the system, since the initial state is system-dependent.

Data Cache State Transition

The following diagram illustrates the data cache state transition sequence. A load or store operation may include one or more of the atomic read and/or write operations shown in the state diagram below, which may cause cache state transitions.

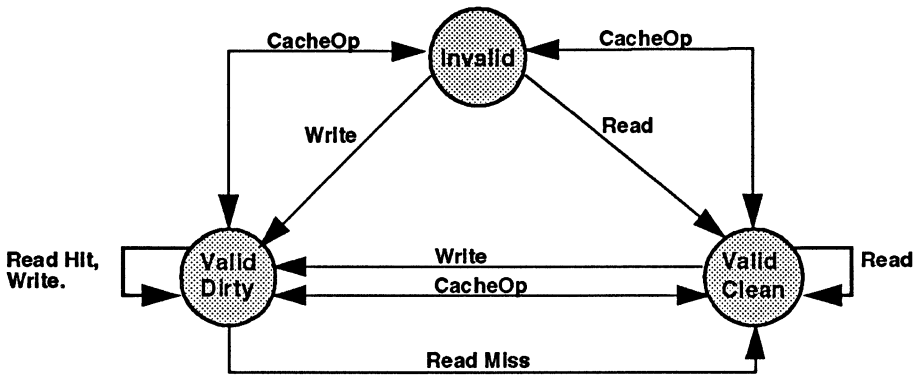


Figure 10-6 Data Cache State Diagram

Instruction Cache State Transition

The following diagram illustrates the instruction cache state transition sequence.

- Read(1) indicates a read operation from memory to cache, inducing a cache state transition.
- Read(2) indicates a read operation from cache to the processor, which induces no cache state transition.

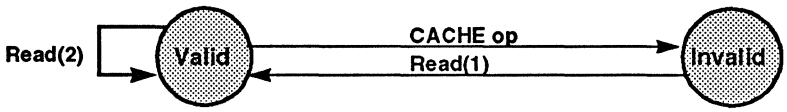


Figure 10-7 Instruction Cache State Diagram

10.6 Manipulation of the Caches by an External Agent

The V_R4300 does not provide any mechanisms for an external agent to examine and manipulate the state and contents of the caches.

System Interface

11

The System interface allows the processor to access external resources needed to satisfy cache misses and uncached operations, while permitting an external agent access to some of the processor internal resources.

This chapter describes the System interface from the point of view of both the processor and the external agent. The V_R4300 uses a subset of the System interface contained on the V_R4000.

11.1 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. System events include: a load that misses in the instruction cache; a load that misses in the data cache; a store that misses in the data cache; an uncached load or store; actions resulting from the execution of cache instructions.
- *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.
- *Block* indicates any transfer larger than a doubleword across the System interface.
- *Single* indicates any transfer of a doubleword or less across the System interface.
- *Fetch* refers to the retrieval of information from the instruction cache.
- *Sequence* refers to a sequential series of requests that the processor generates to service a system event.
- *Timing* refers to the cycle-by-cycle signal transitions that occur on the processor's System interface pins during a processor or external request.
- *Load* refers to the retrieval of information from the data cache.

11.2 System Interface Description

The V_R4300 processor has a 32-bit address/data interface. The System interface consists of:

- 32-bit address and data bus, **SysAD**
- 5-bit command bus, **SysCmd**
- six handshake signals:
  - **EValid*, PValid***
  - **EReq*, PReq***
  - **PMaster*, EOK***

System Interface Signals

Following is a detailed list of the System interface signals, with their descriptions. (I) indicates the signal is a processor input; (O) indicates the signal is output from the processor.

SysAD(31:0) (I/O) Address and data transfer bus, multiplexed between the processor and an external agent.

SysCmd(4:0) (I/O) Used for command and data identifier transmission between the processor and external agent.

EValid* (I) During the cycle it is asserted, **EValid*** indicates an external agent is driving a valid address or valid data on the SysAD bus, and a valid command or data identifier on the SysCmd bus.

PValid* (O) During the cycle it is asserted, **PValid***, indicates the processor is driving a valid address or valid data on the SysAD bus, and a valid command or data identifier on the SysCmd bus.

EReq* (I) Indicates an external agent is requesting System interface bus ownership.

PReq* (O) Indicates the processor is requesting System interface bus ownership. Also, when the processor experiences a protocol error (the processor detects an external agent has violated the SysAD bus protocol), the processor continuously toggles **PReq***.

PMaster* (O) Indicates the processor is the master of the system interface bus.

EOK*	(I) Indicates an external agent is capable of accepting a processor request.
Int(4:0)*	(I) General processor interrupts, visible as bits 14 through 10 of the *Cause* register.
NMI*	(I) Nonmaskable interrupt.
Reset*	(I) When asserted, initiates and maintains a Warm keset in the processor.
TClock	(O) Transmit clock, set to the operational frequency of the System interface. **TClock** has the same frequency and phase as **MasterClock.**
MasterClock	(I) Master clock input, set to the operational frequency of the system interface.
SyncOut	(O) Synchronization clock output.
SyncIn:	(I) Synchronization clock input.
ColdReset*	(I) When asserted, this signal indicates to the V_R4300 processor that the +3.3 volt power supply is stable and the V_R4300 chip should initiate a Cold Reset sequence. The assertion of **ColdReset*** resets the phase-locked loop (PLL). This signal is asynchronous.
JTDI	(I) JTAG serial data in.
JTDO	(O) JTAG serial data out.
JTMS	(I) JTAG command signal that indicates the Serial Data In is command data.
JTCK	(I) JTAG serial clock input.
BypassPLL*	(I) This signal forces **MasterClock** to bypass the PLL and connect directly to the clock buffers.†
TestMode*	(I) This signal is used for direct testing of the cache. The signal must be deasserted (connected to VCC) for normal operation.‡

† This signal should be used for testing only, and is presently be implemented as a nonbonding pad (default state is deasserted) and may not be present on the production package. However this pin will remain a part of the JTag scan chain.

‡ This signal is implemented as a nonbonding pad (default state is deasserted) and may not be present on the production package. However this pin will remain part of the JTag scan chain.

DivMode(1:0) (I) These signals are an encoding of the **PClock**-to-**MasterClock** ratios. **TClock** (system interface clock) is the same frequency as **MasterClock**. As an example, the encoding of **DivMode** for 40 MHz **MasterClock** is shown in Table 11-1 below:

Table 11-1 DivMode Settings and Frequencies

DiveMode(1:0)	MasterClock	TClock	PClock	Ratio
00	40MHz	40MHz	40MHz	1:1
01	40MHz	40MHz	60MHz	1.5:1
10	40MHz	40MHz	80MHz	2:1
11	40MHz	40MHz	120MHz	3:1

The SysAD bus and the SysCmd bus are driven by the processor to issue a processor request when it is master, as indicated by the assertion of **PMaster***. The SysAD bus and the SysCmd bus are driven by an external agent to issue an external response when the processor is a slave, as indicated by deassertion of **PMaster***.

The processor uses the System interface to access external resources such as cache misses and uncached operations.

System Events

System events include:

- a fetch that misses in the instruction cache
- a load that misses in the data cache
- a store that misses in the data cache
- an uncached load or store
- actions resulting from the execution of cache instructions

When a system event occurs, the processor issues a request or a series of requests through the system interface to access some external resource to service that event. The system interface must be connected to an external agent that coordinates access to system resources.

Processor requests include both read and write requests:

- a read request supplies an address to an external agent
- a write request supplies an address and a word or block of data to be written to an external agent

Processor read requests that have been issued, but for which data has not yet been returned, are said to be *pending*. The processor will not issue another request while a read is already pending. A processor read request is said to be *complete* after the last transfer of response data has been received from an external agent. A processor write request is said to be complete after the last word of data has been transmitted.

External requests include both read responses and write requests:

- a read response supplies a block or single transfer of data from an external agent in response to a read request
- a write request supplies an address and a word of data to be written to a processor resource

When an external agent receives a read request, it accesses the specified resource and returns the requested data through a read response, which may be returned any time after the read request and at any data rate.

By default, the processor is the master of the system interface. An external agent becomes master of the system interface either through arbitration, or by default after a processor read request. The external agent returns mastership to the processor after the external request completes and/or after the processor read request has been serviced.

System Event Sequences and the SysAD Bus Protocol

The following sections detail the sequence and timing of processor and external requests.

NOTE: The following sections describe the SysAD bus *protocol;* the V_R4300 processor always meets the conditions of this protocol. The V_R4300 processor is capable of receiving sequences of transactions on the bus at full protocol speed and of receiving data on every cycle. At a minimum, the design of external agents must meet the requirements of this protocol, and would ideally take full advantage of the maximum speed of the V_R4300 processor.

Fetch Miss

When the processor misses in the instruction cache on a fetch, it obtains a cache line of instructions from an external agent. The processor issues a read request for the cache line and waits for an external agent to provide the data in response to this read request.

Load Miss

When the processor misses in the data cache on a load, it obtains a cache line of data from an external agent. The processor issues a read request for the cache line and waits for an external agent to provide the data in response to this read request. If the cache data which the incoming line will replace contains valid dirty data, this data is written to memory. The read completes before the write of the dirty cast-out data.

Store Miss

When the processor misses in the data cache on a store, it issues a read request to bring a cache line of data into the cache, where it is then updated with the store data. If the cache data which the incoming line will replace contains valid dirty data, the data is written to memory. The read completes before the write of the dirty cast-out data.

To guarantee that cached data written by a store is consistent with main memory, the corresponding cache line must be explicitly flushed from the cache using a cache operation.

Uncached Load or Store

When the processor performs an uncached load, it issues a read request and waits for a single transfer of read response data from an external agent.

When the processor performs an uncached store, it issues a write request and provides a single transfer of data to the external agent.

The processor does not consolidate data on uncached writes. For example, writes of two contiguous halfwords takes two write cycles, they are never grouped into a single word write.

Cache Instructions

The V_R4300 processor provides a number of cache instructions for use in maintaining the state and contents of the caches. Cache operations supported in the V_R4300 processor are described in Chapter 10.

Byte Ordering

The System interface byte order is set by the *BigEndian* (*BE*) bit in the *Config* register. The byte order is big-endian when *BE* is high, and little-endian when *BE* is low. The *RE* (reverse-endian) bit in the *Status* register can be set by software to reverse the byte order available in User mode.

Physical Addresses

Physical addresses are driven on **SysAD(31:0)** during address cycles.

Interface Buses

Figure 11-1 shows the primary communication paths for the System interface: a 32-bit address and data bus, **SysAD(31:0)**, and a 5-bit command bus, **SysCmd(4:0)**. 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.

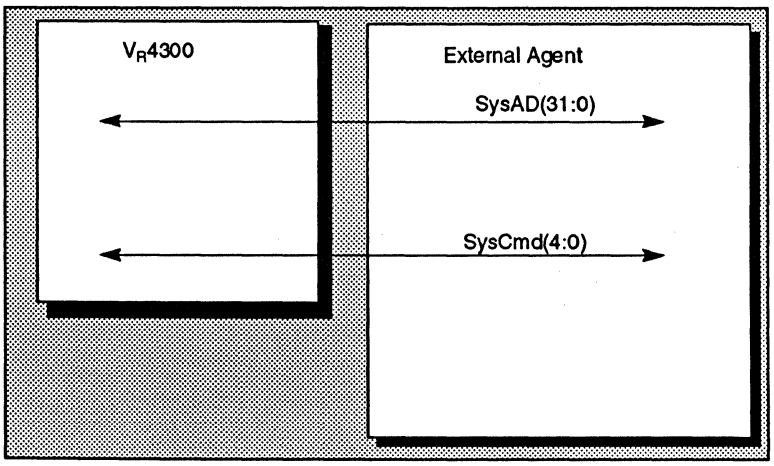


Figure 11-1 System Interface Buses

Address and Data Cycles

The **SysCmd** bus identifies the contents of the **SysAD** bus during any cycle in which it is valid. 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*. 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.

When the V_R4300 processor is driving the **SysAD** and **SysCmd** buses, the System interface is in *master state*. When the external agent is driving the **SysAD** and **SysCmd** buses, the System interface is in *slave state*.

- When the processor is master, it asserts the **PValid*** signal when the **SysAD** and **SysCmd** buses are valid.
- When the processor is slave, an external agent asserts the **EValid*** signal when the **SysAD** and **SysCmd** buses are valid.

The **SysCmd** bus identifies the contents of the **SysAD** bus during valid cycles.

- During address cycles [**SysCmd(4)** = 0], the remainder of the **SysCmd** bus, **SysCmd(3:0)**, contains a *System interface command*, described later in this chapter.
- During data cycles [**SysCmd(4)** = 1], the remainder of the **SysCmd** bus, **SysCmd(3:0)**, contains a *data identifier*, described later in this chapter.

11.3 System Interface Protocols

Figure 11-2 shows the register-to-register operation of the System interface. That is, processor outputs come directly from output registers and begin to change with the rising edge of **SClock.**†

Processor inputs are fed directly to input registers that latch these input signals with the rising edge of **SClock**. This allows the System interface to run at the highest possible clock frequency.

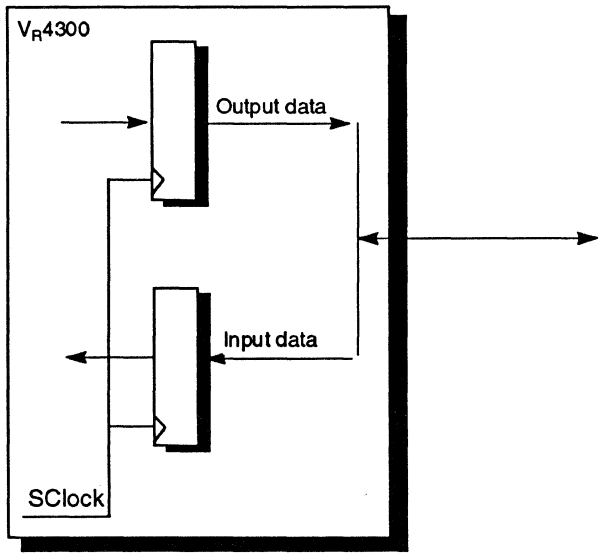


Figure 11-2 System Interface Register-to-Register Operation

Master and Slave States

When the V_R4300 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 asserts the signal **PValid*** whenever the **SysAD** and **SysCmd** buses are valid.

In slave state, the external agent asserts the signal **EValid*** whenever the **SysAD** and **SysCmd** buses are valid.

† **SClock** is an internal clock used by the processor to sample data at the System interface and to clock data into the processor System interface output registers.

Moving from Master to Slave State

The processor is the default master of the system interface. An external agent becomes master of the system interface through arbitration, or by default after a processor read request. The external agent returns mastership to the processor after an external request completes.

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 (uncompelled change to slave state).

The following sections describe these two actions.

External Arbitration

The System interface must be in slave state for the external agent to issue an external request through the System interface. The transition from master state to slave state is arbitrated by the processor using the System interface handshake signals **EReq*** and **PMaster***. This transition is described by the following procedure:

1. An external agent signals that it wishes to issue an external request by asserting **EReq***.
2. When the processor is ready to accept an external request, it releases the System interface from master to slave state by negating **PMaster***.
3. The System interface returns to master state as soon as the issue of the external request is complete.

Uncompelled Change to Slave State

An *uncompelled* change to slave state is the transition of the System interface from master state to slave state, initiated by the processor itself when a processor read request is pending. **PMaster*** is negated automatically after a read request. An uncompelled change to slave state occurs either during or some number of cycles after the issue cycle of a read request.

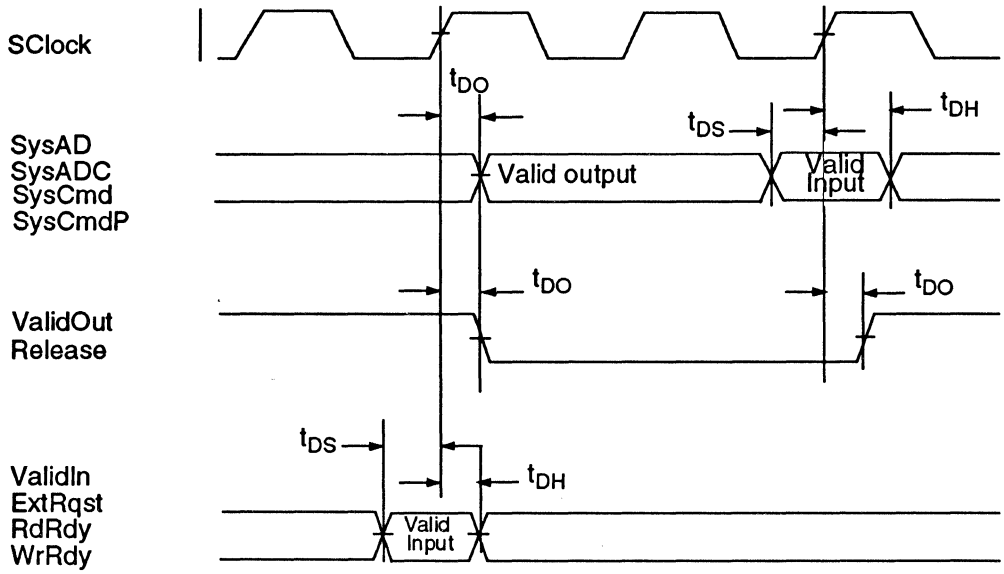
The uncompelled release latency depends on the state of the cache. After an uncompelled change to slave state, the processor returns to master state at the end of the next external request. This can be a read response, or some other type of external request.

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 an external request without arbitrating for the System interface; that is, without asserting **EReq***.

After the external request, the System interface returns to master state.

Signal Timing

The System interface protocol describes the cycle-by-cycle signal transitions that occur on the pins of the system interface to realize requests between the processor and an external agent. Figure 11-3 shows the timing relationships between System interface signal edges.



Note: These waveforms only describe edge-to-edge timing relationships.

Figure 11-3 System Interface Edge Timing Relationships

The Timing Summary section below describes the minimum and maximum timing values of each signal. The sections that follow describe the timing requirements for various bus cycles.

Timing Summary

In the following timing diagrams, gray-scale signals indicate values that are either Unknown or Don't Cares, within the specification limits. They may be any value as long they do not violate any bus value or timing specification. The timing diagrams illustrate cycles using the following signals:

- **PMaster***
- **EValid***
- **PValid***
- **EOK***
- **EReq***
- **PReq***

PMaster* (O) Indicates the processor is the master of the system interface bus.

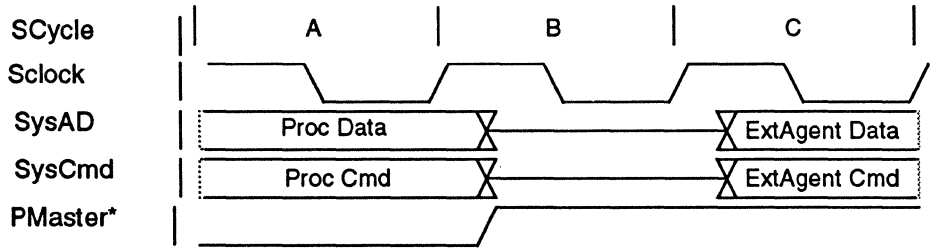


Figure 11-4 Sample Cycle with **PMaster*** *Asserted, Then Deasserted*

A Processor drives SysAD and SysCmd buses (processor is master).

B **PMaster*** is deasserted. SysAD and SysCmd buses are set to a tri-state (no bus master).

C External agent drives SysAD and SysCmd buses (external agent is master).

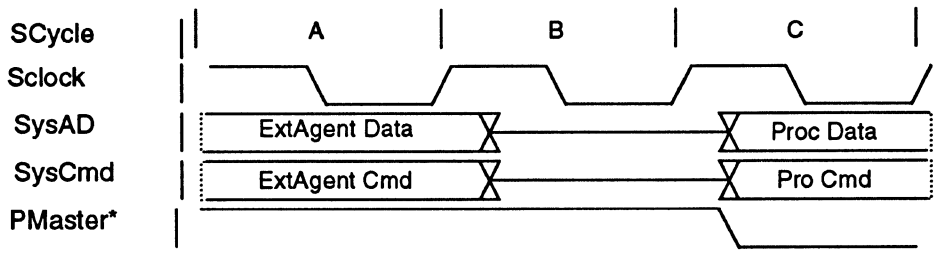


Figure 11-5 Sample Cycle with **PMaster*** *Asserted*

A External agent drives SysAD and SysCmd buses (external agent is master).

B SysAD and SysCmd buses are set to a tri-state (no bus master).

C **PMaster** is asserted. Processor drives SysAD and SysCmd buses (processor is master).

EValid* (I), **PValid*** (O) During a cycle in which either signal is asserted, the signal indicates a new valid address or valid data is on the SysAD bus, and a new valid command or data identifier is on the SysCmd bus. **EValid*** indicates an external agent is driving new SysAD and SysCmd values. **PValid*** indicates the processor is driving new SysAD and SysCmd values.

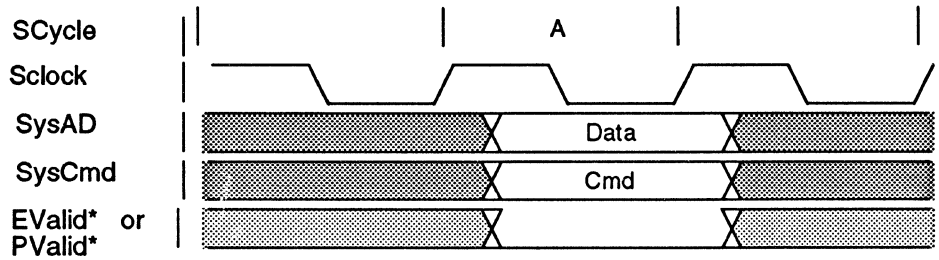


Figure 11-6 Sample Cycle with **PValid*** *and* **EValid***

A: New SysAD and SysCmd values.

Each cycle either of these signals remains asserted indicates there is a new SysAD and SysCmd value.

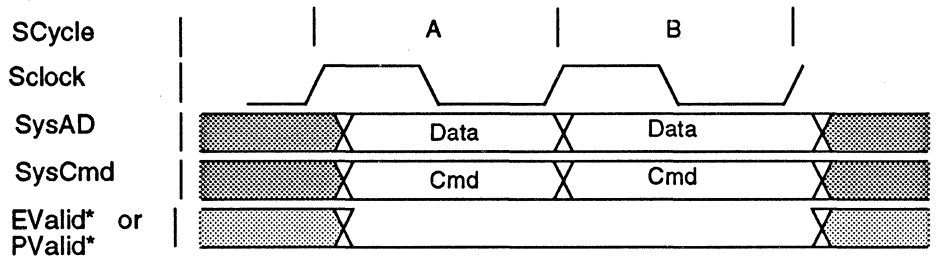


Figure 11-7 Sample Cycles with Multiple **PValid*** *and* **EValid***

A New SysAD and SysCmd value.

B Another new SysAd and SysCmd value.

EOK* (I) Indicates an external agent accepts a processor request. An external agent has excepted the processor read/write command if and only if the following has occurred:

A **EOK*** is active.

B The processor asserts **PValid*** and drives a read or write command. **EOK*** is asserted and the external agent accepts the processor command.

Once the external agent has accepted a processor write command, the agent must be able to accept the entire data size at the programmed data rate immediately following this command.

The external agent may provide read response data to the processor at any rate.

Deasserting EOK may kill a processor read/write request in progress. If this occurs, the external agent must ignore command and data from the processor in the following cycle.

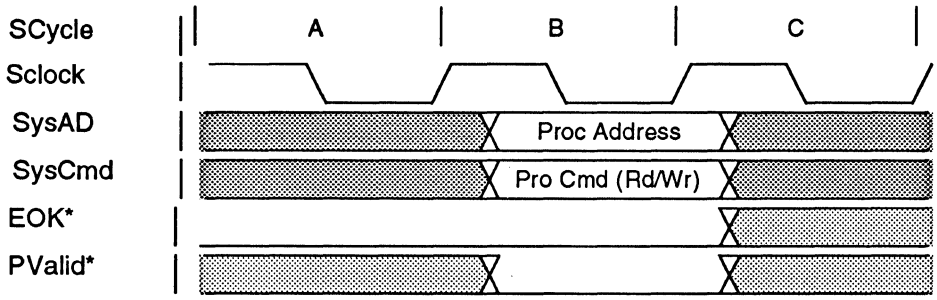


Figure 11-8 Sample Cycle with **EOK*** *Asserted*

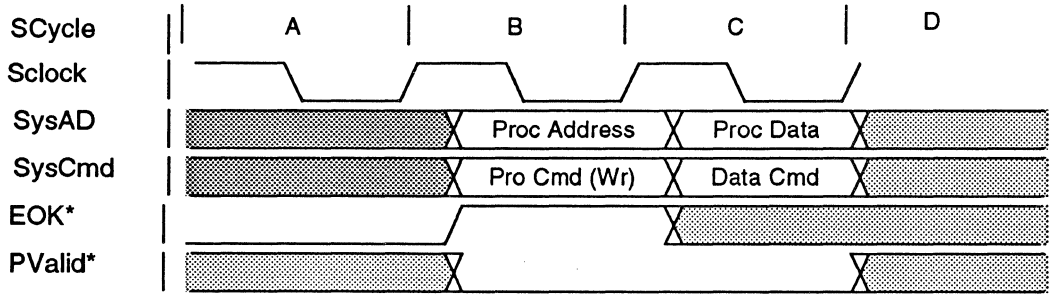


Figure 11-9 Sample Cycle with **EOK*** *Asserted, Then Deasserted*

A EOK* is active.

B Processor asserts **PValid*** and drives a read or write command. **EOK*** is deasserted (external agent has killed the processor's command).

C The external agent must ignore any SysAD and SysCmd data from the processor.

D The external agent *does not* ignore any SysAD and SysCmd data from the processor.

EReq* (I) Indicates an external agent is requesting bus ownership of the System interface. To gain mastership of the bus, an external agent must arbitrate with the processor as follows:

A External agent asserts **EReq***

B Wait for **PMaster*** to be deasserted (1 to N cycles).

C External agent drives SysAD and SysCmd buses. The external agent is guaranteed to maintain mastership of the bus as long as **EReq*** is asserted.

If at any time **EReq*** is deasserted, the external agent must go back to step A and re-arbitrate for the bus.

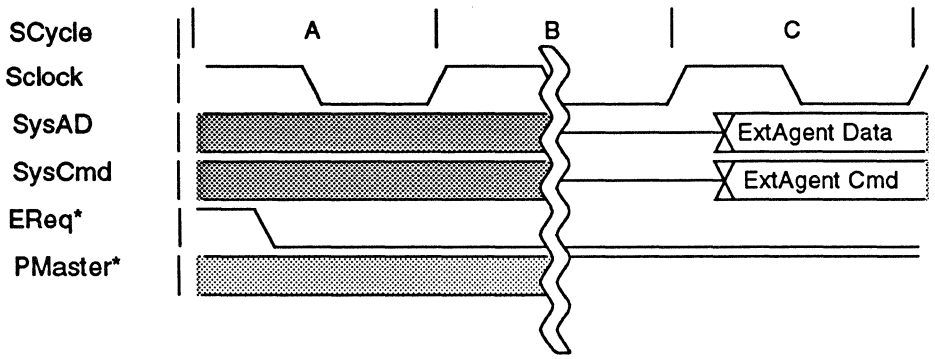


Figure 11-10 Sample Cycle with **EReq*** *Asserted*

From the time that **EReq*** is asserted, the external agent is guaranteed to gain mastership of the bus after at most one processor request. However, if **EOK*** is being deasserted, the external agent will gain mastership of the bus without having to accept any processor requests.

The external agent relinquishes bus mastership by deasserting **EReq*** as shown below:

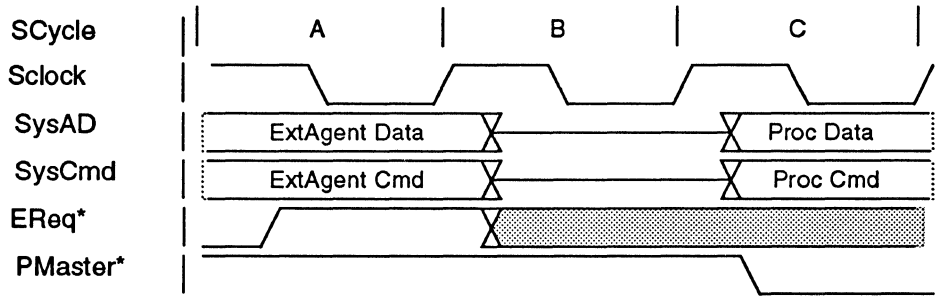


Figure 11-11 Sample Cycle with Deassertion of **EReq***

A External agent deasserts **EReq*** and external agent drives the bus.

B Bus is set to a tristate.

C Processor regains mastership of bus.

Except for a processor read request (see below), assertion of **EReq*** is the only way the external agent gets and maintains bus mastership.

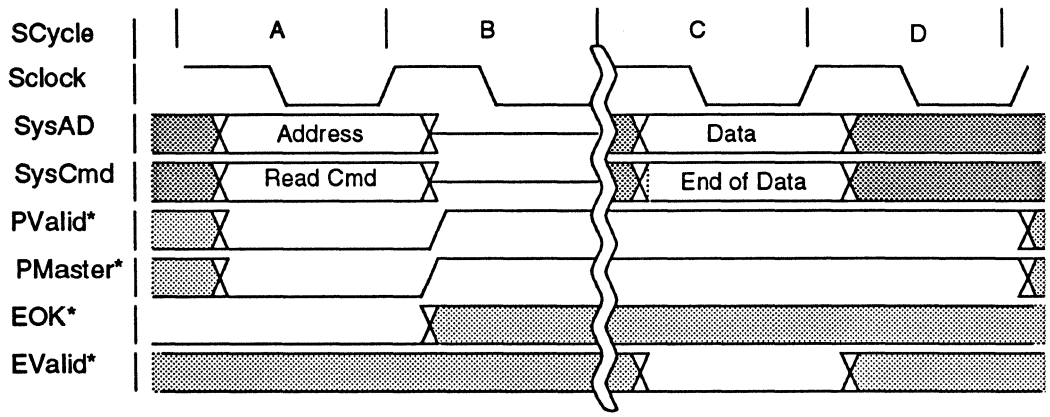


Figure 11-12 Sample Cycle with Assertion of **EReq***

A — Processor drives a valid read command and an external agent accepts it.

B — **PMaster*** is deasserted and the bus is set to a tristate.

C — External agent drives last of requested data. During all cycles between B and C the external agent is guaranteed mastership of the bus.

PReq* (O) — Indicates the processor is requesting the bus. When the processor is in slave state and has a read/write request to issue, it asserts **PReq***.

PReq* is also used to indicate when the V_R4300 processor has detected a protocol error. When a protocol error is detected, the processor's system interface hangs and **PReq*** toggles.

Arbitration

The processor is the default master of the bus. It relinquishes ownership of the bus either when an external agent requests and is granted the system interface, or until the processor issues a read request. The transition from processor master to processor slave state is arbitrated by the processor, using the System interface handshake signals **EReq*** and **PMaster***.

When a processor read request is pending, the processor transitions to slave state by deasserting **PMaster***, allowing an external agent to return the read response data. The processor remains in slave state until the external agent issues an *End Of Data* read response, whereupon the processor reassumes mastership, signalled by the assertion of **PMaster***. Note that an external agent is able to retain mastership of the bus after an End Of Data read response if the external agent arbitrates for mastership using **EReq***.

When the processor is master, an external agent acquires control of the system interface by asserting **EReq***, and waiting for the processor to deassert **PMaster***. The processor is ready to enter slave state when it deasserts **PMaster***. The external agent must go through a three-step arbitration process (see the **EReq*** cycle in the Timing Summary) before driving the bus. Once the external agent has become master through **EReq*** arbitration, it can remain master as long as it continues to assert **EReq***. The System interface returns to master state (with the processor driving the bus) two cycles after **EReq*** is deasserted. Figure 11-13 illustrates an arbitration for external requests.

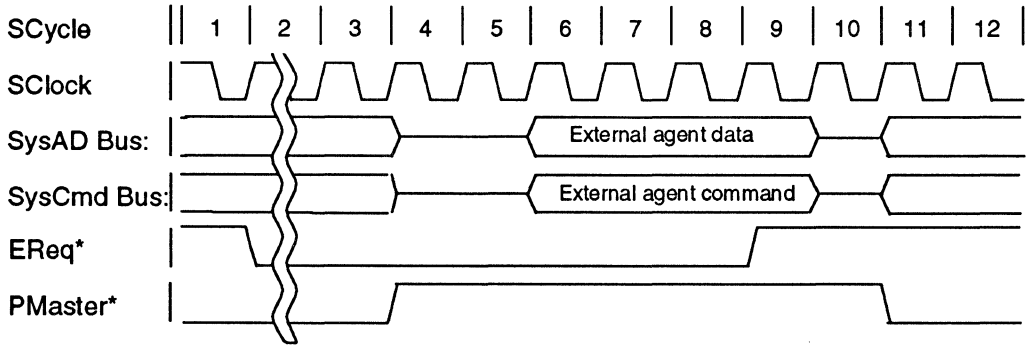


Figure 11-13 External Request Arbitration

When an external agent is master, it may always respond to a read with data. If the external agent has become master by **EReq***, it may issue transactions at will; that is, the processor must always accept any command or data on the bus at any time. There is no means for the processor to hold off the external agent once the external agent is master. However the processor can request the bus by asserting **PReq***, and the external agent may or may not honor the request depending upon system priorities.

If the processor is in slave state and needs the bus, it may assert the **PReq*** to reclaim mastership of the bus. Thereafter, when the processor sees **EReq*** deasserted, it resumes bus ownership, asserts the **PMaster*** line, and issues its own command. The processor becomes master and drives the bus two cycles after **EReq*** is deasserted.

An illustration of a processor request for bus mastership and the release of the bus by the external agent is illustrated in Figure 11-14.

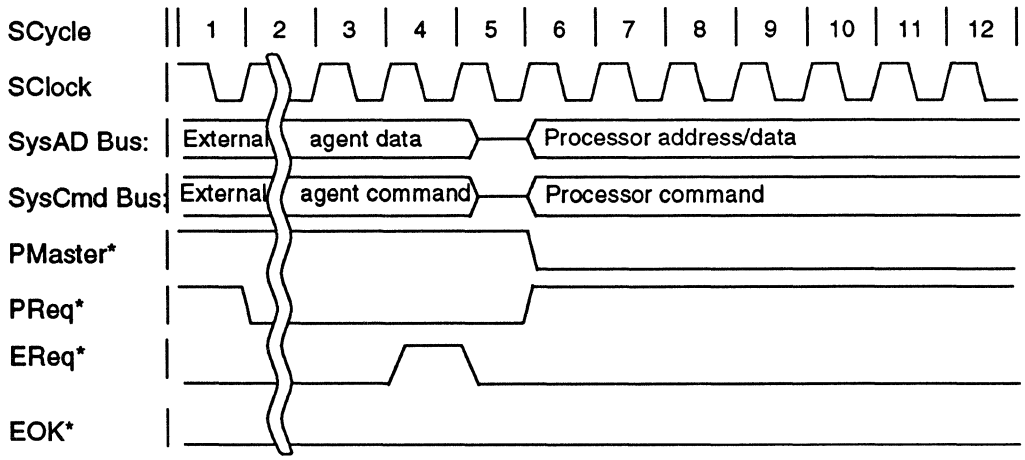


Figure 11-14 Processor Request For Bus Arbitration And External Agent Release

Upon assertion of **Reset*** or **ColdReset***, the processor becomes bus master and the external agent must become slave.

This protocol guarantees that either the processor or an external agent is always bus master. The master should never tristate the bus, except when giving up ownership of the bus under the rules of the protocol.

Issuing Commands

When the processor is master of the bus and wishes to issue a command, it cannot successfully issue the command until the external agent signals that it is ready to accept it. This readiness is indicated by assertion of the **EOK*** signal. Being master, the processor may place the command on the bus and continually reissue it while waiting for **EOK*** to be asserted; however, the command is not considered issued until **EOK*** has been asserted for two consecutive cycle (see the Timing Summary for **EOK*** earlier in this chapter).

If the **EOK*** signal is asserted in one cycle and then deasserted in the next, during which time a command is issued, that command is considered killed and must be retried. When a command is killed in this way, the processor begins to execute the read/write command. This action must be ignored by the external agent. If a write command is killed, the data cycle following this killed transaction must be ignored. If a read is killed, the processor releases the bus one cycle after and (assuming no **EReq***) regains mastership two cycles later. This allows the processor to retry the transaction.

Processor Write Request

A processor write request is issued by the following:

- driving a write command on the SysCmd bus
- driving a write address on the SysAD bus
- asserting **PValid*** for one cycle
- driving the appropriate number of data identifiers on the SysCmd bus
- driving data on the SysAD bus
- asserting **PValid***

For 1- to 4-byte writes, a single data cycle is used. 5-, 6- and 7-byte writes are broken up into two address/data transactions; one 4 bytes in size, the next handling the remaining 1, 2, or 3 bytes.

For all transactions larger than 7 bytes (e.g. 8, 16, 32), 4 bytes are sent on each data cycle until the appropriate number of bytes has been transferred. The final data cycle is tagged as end of data (EOD) on the command bus.

To be fully compliant with all implementations of this protocol, an external agent should be able to receive write data over any number of cycles with any number of idle cycles between any two data cycles. However, for the V_R4300 processor implementation, data begins to arrive on the cycle immediately following the write issue cycle, and continues to arrive at a programed data rate thereafter. The processor drives data at the rate specified by the data rate configuration signals (see the section describing Data Rate Control, later in this chapter).

Writes may be cancelled and retried with the **EOK*** signal (see the section earlier, Issuing Commands).

Figure 11-15 illustrates the bus transactions for a 4-word data cache block store.

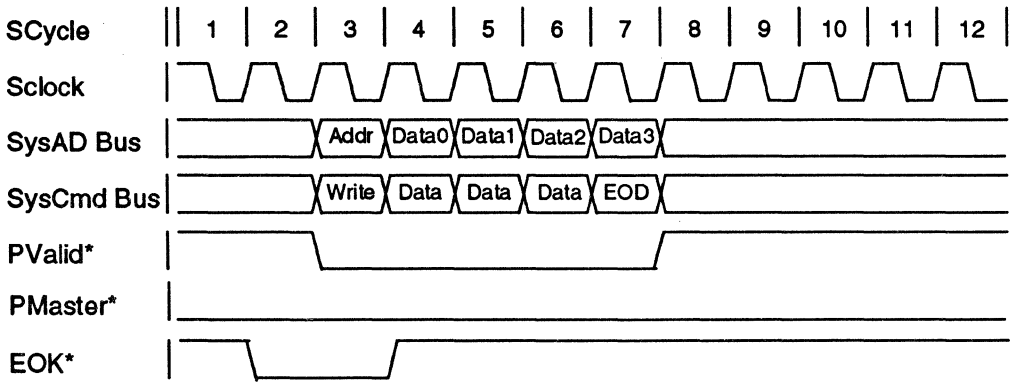


Figure 11-15 Processor Block Write Request With D Data Rate

illustrates a write request which is cancelled by the deassertion of **EOK*** during the address cycle of the second write, and which is retried when **EOK*** is asserted again.

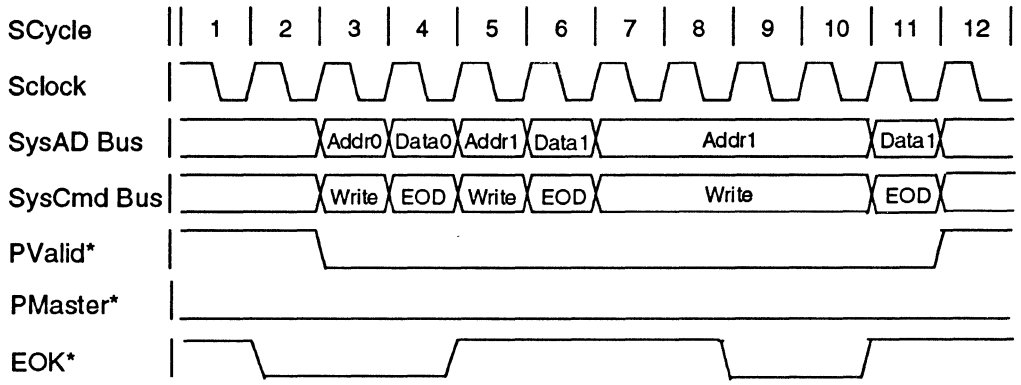


Figure 11-16 Processor Single Write Request Followed By A Cancelled And Retried Write Request

Processor Read Request

A processor read request is issued by the following:

- driving a read command on the SysCmd bus
- driving a read address on the SysAD bus
- asserting **PValid***

Only one processor read request may be pending at a time. The processor must wait for an external read response before starting a subsequent read.

The processor moves to slave state after the issue cycle of the read request, by deasserting the **PMaster*** signal. An external agent may then return the requested data through a read response. The external agent, which is now bus master, may issue any number of writes before sending the read response data.

An example of a processor read request and an uncompelled change to slave state occurring as the read request is issued is illustrated in Figure 11-17.

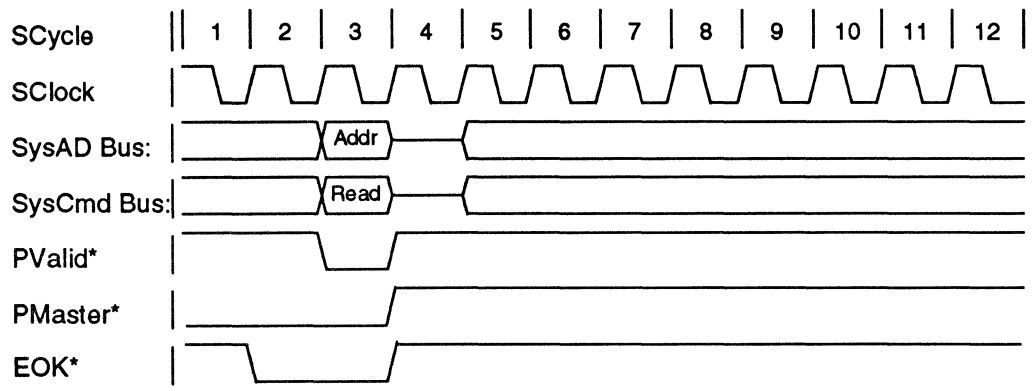


Figure 11-17 Processor Read Request

External Write Request

External write requests are similar to a processor single write except that the signal **EValid*** is asserted instead of the signal **PValid***. An external write request consists of the following:

- an external agent driving a write command on the SysCmd bus and a write address on the SysAD bus
- asserting **EValid*** for one cycle
- driving a data identifier on the SysCmd bus and data on the SysAD bus
- asserting **EValid*** for one cycle.

The data identifier associated with the data cycle must contain a last data cycle indication. Note that the external agent must gain and maintain bus mastership during these transactions (see **EReq*** in the Timing Summary, earlier in this chapter).

An external write request example with the processor initially in master state is illustrated in Figure 11-18.

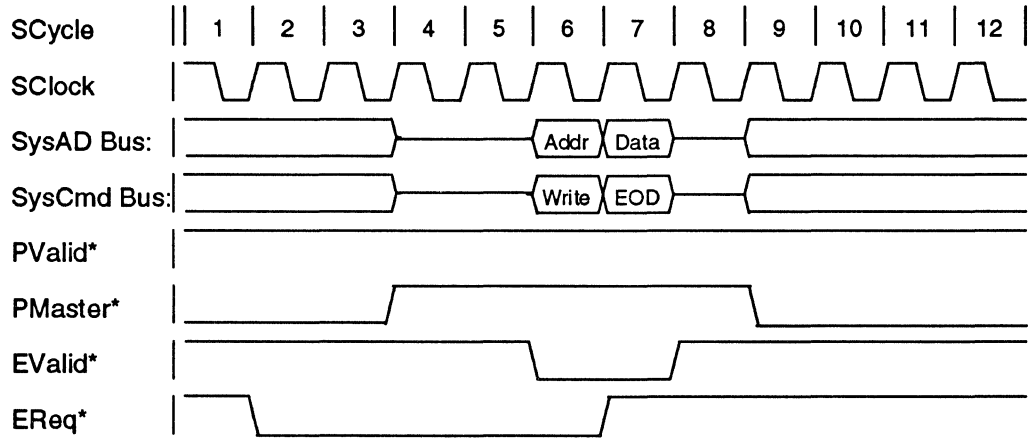


Figure 11-18 External Write Request

An example of a read response for a processor single word read request that is interrupted by an external agent write request is illustrated in Figure 11-22. External writes can not occur in the middle of a data response block; they can, however, occur before the first data response of the data block or after the last EOD response, but it can not occur between them.

NOTE: The only writable resources are processor interrupts. An external write to any address is treated as a write to the processor interrupts.

External Read Response

An external agent returns data to the processor in response to a processor read request by waiting for the processor to move to slave state. The external agent then returns the data through either a single data cycle or a series of data cycles sufficient to transmit the requested data. After the last data cycle is issued, the read response is complete and the processor becomes master (assuming **EReq*** was not asserted).

If, at the end of the read response cycles, **EReq*** has been asserted, the processor remains in slave state until the external agent relinquishes the bus. When the processor is in slave state and needs access to the SysAD bus, it asserts **PReq*** and waits until **EReq*** is deasserted.

The data identifier associated with a data cycle may indicate that the data transmitted during that cycle is erroneous; however, an external agent must return a block of data of the correct size regardless of this erroneous data cycle indication. If a read response includes one or more erroneous data cycles, the processor takes a bus error.

Read response data must only be delivered to the processor when a processor read request is pending. The state of the processor is undefined if a read response is presented to it when no processor read is pending.

An example of a processor single read request followed by a read response is illustrated in Figure 11-19.

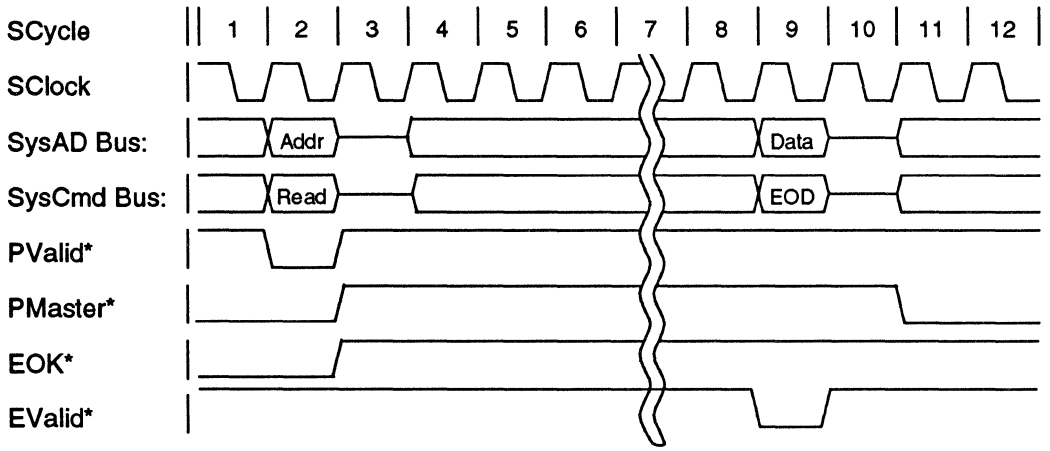


Figure 11-19 Single Read Request Followed By Read Response

A read response example for a processor block read with the system interface already in slave state is illustrated in Figure 11-20.

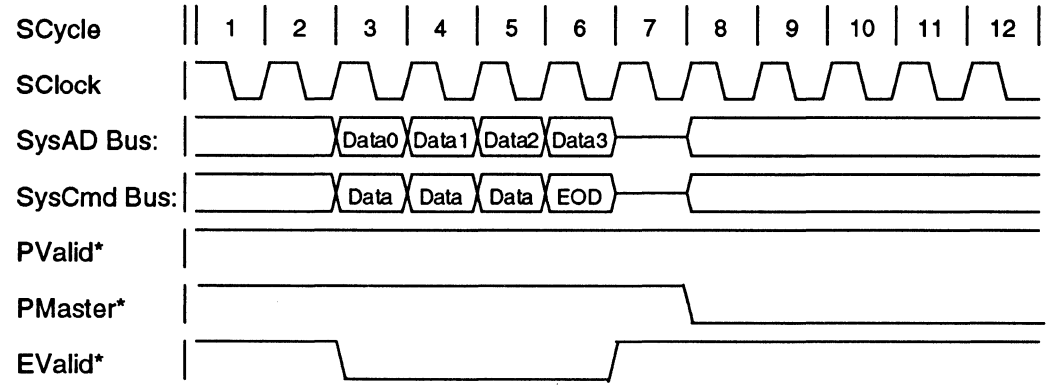


Figure 11-20 Block Read Response, System Interface Already In Slave State

A read response example for a processor single read request followed by an external agent write request is illustrated in Figure 11-21.

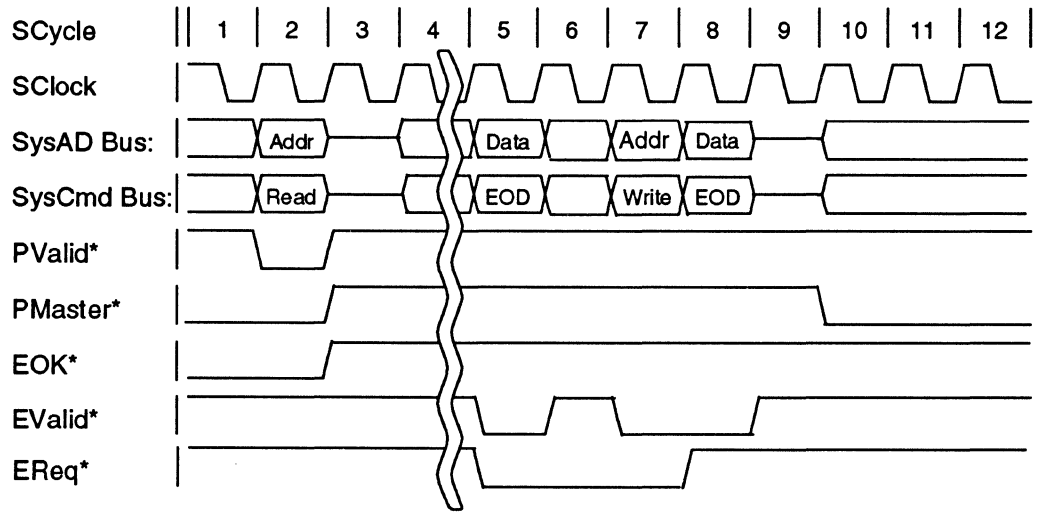


Figure 11-21 Single Read Request Followed By External Write Request (External Agent Keeps Bus)

An example of a read response for a processor single word read request that is interrupted by an external agent write request is illustrated in Figure 11-22. Cycle 5 is the data for the external write request in cycle 4. Cycle 7 is the read response data.

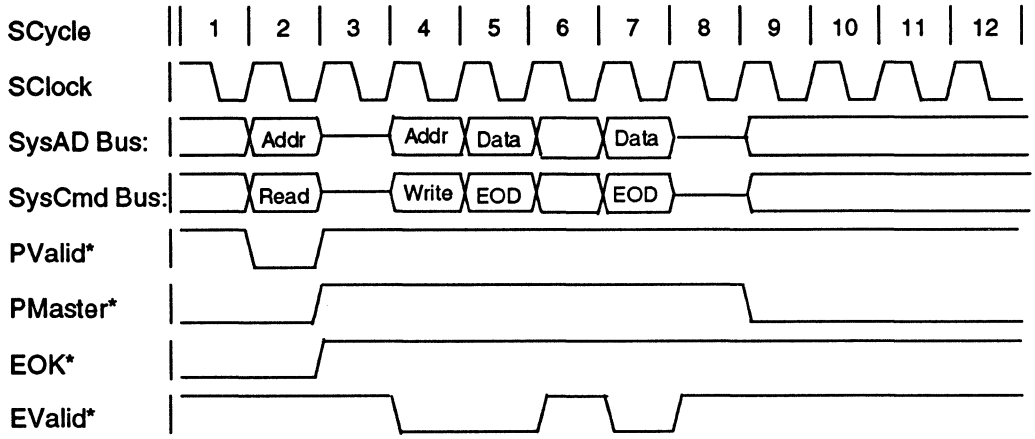


Figure 11-22 External Write Followed By External Read Response, System Interface In Slave State

Flow Control

EOK* may be used by an external agent to control the flow of processor read and write requests; while EOK* is deasserted the processor will repeat the current address cycle until an external agent signals it is ready, by asserting EOK*. There is a one cycle delay from the assertion of EOK* to the state in which the Read/Write command becomes valid. EOK* must be asserted for two consecutive cycles for the command issue completion. Examples of EOK* use are given in Figures 11-23 and 11-24.

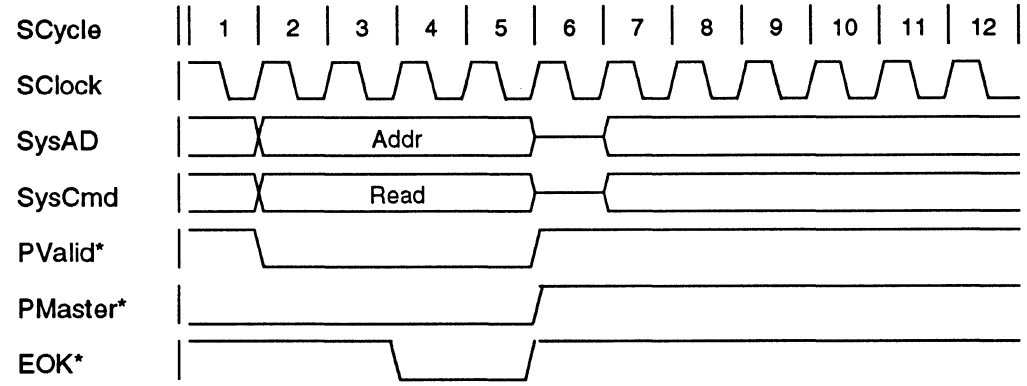


Figure 11-23 Delayed Processor Read Request

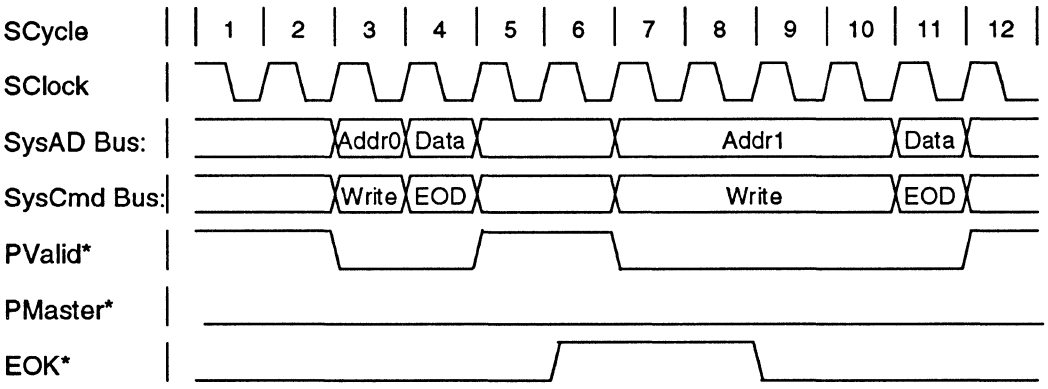


Figure 11-24 Two Processor Write Requests, Second Write Delayed

Data Rate Control

The System interface supports a maximum data rate of one word per cycle, and an external agent may deliver data to the processor at this maximum data rate. The rate at which data is delivered to the processor can be controlled by the external agent by driving data and asserting **EValid*** only when it wants data to be available.

The processor interprets cycles as valid data cycles when **EValid*** is asserted and the SysCmd bus contains a data identifier. The processor continues to accept data until the end of data (EOD) indicator is received.

The rate at which the processor transmits data to an external agent is programmed in the *EP* field in *Config* register. Data patterns are specified using the letters **D** and **x** (**D** indicates a data cycle and **x** indicates an unused, or idle, cycle). A data pattern is specified as a sequence of data and unused cycles that will be repeated to provide the appropriate number of data cycles for a given transfer. For example, a data pattern of **DDxx** indicates a data rate of two words every four cycles.

V_R4300 supports two data rates, **D** and **Dxx**. During a cycle indicated by an **x**, the processor continues to hold the same data as the previous cycle.

A processor block write request for two words with Dxx pattern is illustrated in Figure 11-25; this transaction results from a store doubleword instruction.

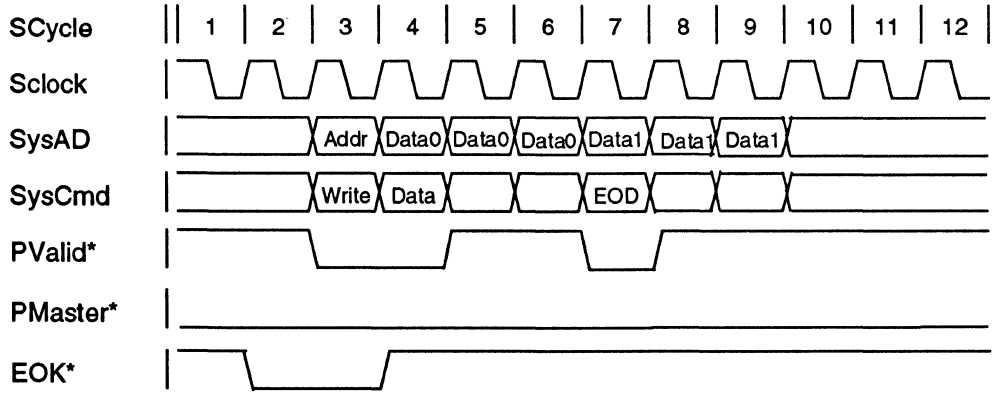


Figure 11-25 Processor Block Write Request With Dxx Data Rate

Consecutive SysAD Bus Transactions

The following figures (Figures 11-26 to 11-29) illustrate the minimum cycles required between consecutive bus transactions.

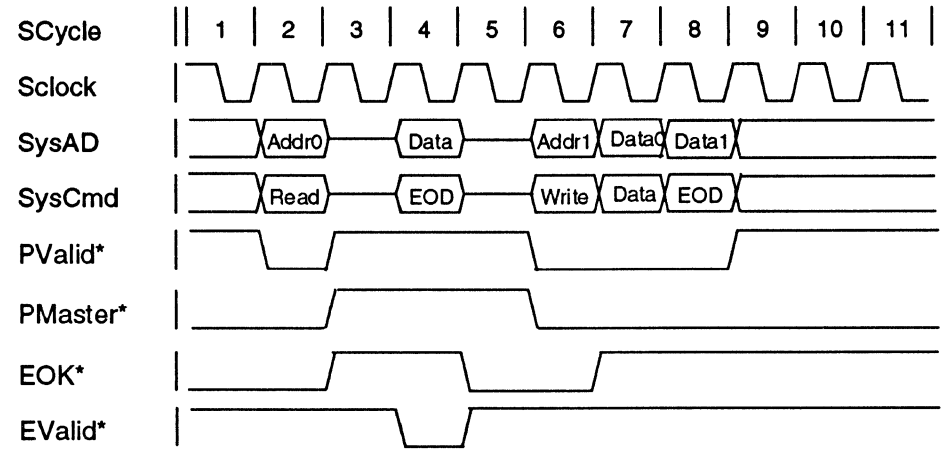


Figure 11-26 Processor Single Word Read Followed By Block Write Request

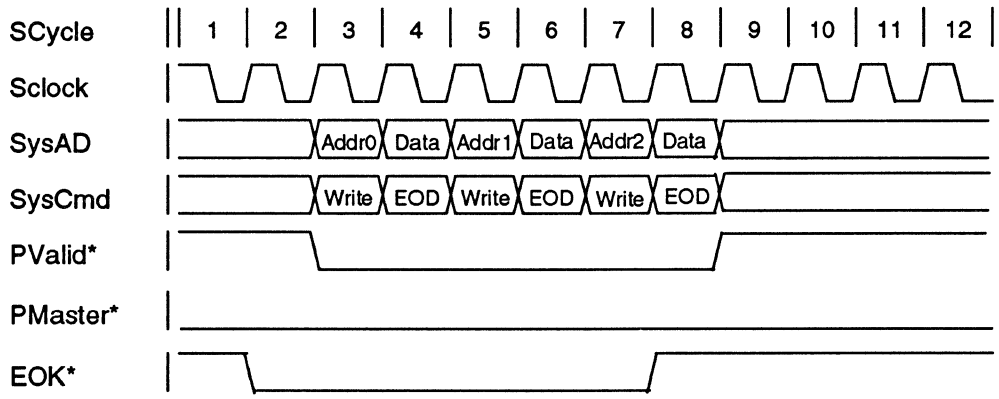


Figure 11-27 Consecutive Processor Single Word Write Requests With D Data Rate

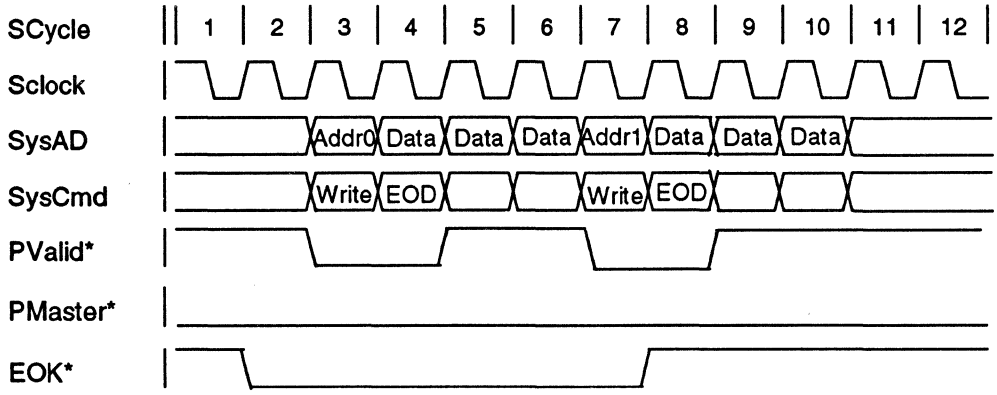


Figure 11-28 Consecutive Processor Single Word Write Requests With Dxx Data Rate

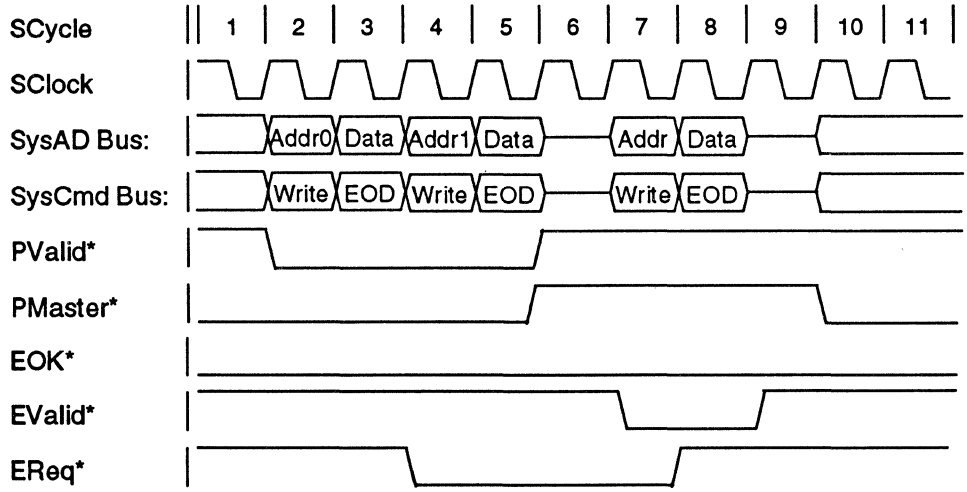


Figure 11-29 Consecutive Processor Write Requests Followed By External Write Request

Block Read Maximum Rate

Maximum block reads can occur with the following data rate:

```
AxD...DxAxD...D (1 cycle between D and A)
```

where **A** is the address, **D** is data (four words, or DDDD, in the data cache miss, and 8 words, or DDDDDDDD, in an instruction cache miss), and **x** is an idle cycle.

Back-to-Back Instruction Cache Misses

With a PClock to SClock ratio of 2:1, back-to-back instruction cache misses can be refilled with the following data rate:

```
AxDDDDDDDDxxxAxDDDDDDDD (3 cycles between D and A)
```

That is, the address is followed by an idle cycle, the instruction is executed, three idle cycles occur, followed by the next address. This pattern is valid for the case in which two sequential instructions miss in the instruction cache, each instruction residing on a different cache line.

Running completely in uncached space (every instruction is uncached and a cache miss) results in a similar data pattern:

```
AxDxxxAxD (3 cycles between D and A)
```

Back-to-Back Uncached Loads

With a PClock to SClock ratio of 2:1, back-to-back uncached doubleword loads have the following data rate:

```
AxDDxxxAxDD (3 cycles between D and A)
```

That is, the address is followed by an idle cycle, a doubleword of data, three idle cycles, and the next address.

Back-to-back uncached word loads have the following data rate:

```
AxDxxxAxD (3 cycles between D and A)
```

Starvation and Deadlock Avoidance.

Careful use of the **EReq*** and **PReq*** signals allows a system to avoid starvation and deadlock situations.

Whenever an external agent needs the bus, it can request the bus by asserting **EReq***. The external agent is guaranteed to gain mastership of the bus after accepting at most one read/write request from the processor. If the external agent also deasserts **EOK***, it is guaranteed to gain mastership of the bus without accepting any read/write request from the processor.

The processor asserts **PReq*** to gain ownership of the bus. The external agent can allow the processor to gain bus mastership, perform one read/write request and then relinquish mastership by the following sequence of actions:

1. deassert **EReq***
2. assert **EReq***
3. arbitrate for the bus while asserting **EOK***

The minimum deassertion of **EReq*** can be one cycle in length.

shows an external agent relinquishing the bus to allow a single read/write request from the processor. The external agent must be ready to accept this request by keeping **EOK*** asserted, otherwise the read/write request is held off or killed and the processor relinquishes bus mastership without extending a request. This could lead to starvation of the processor.

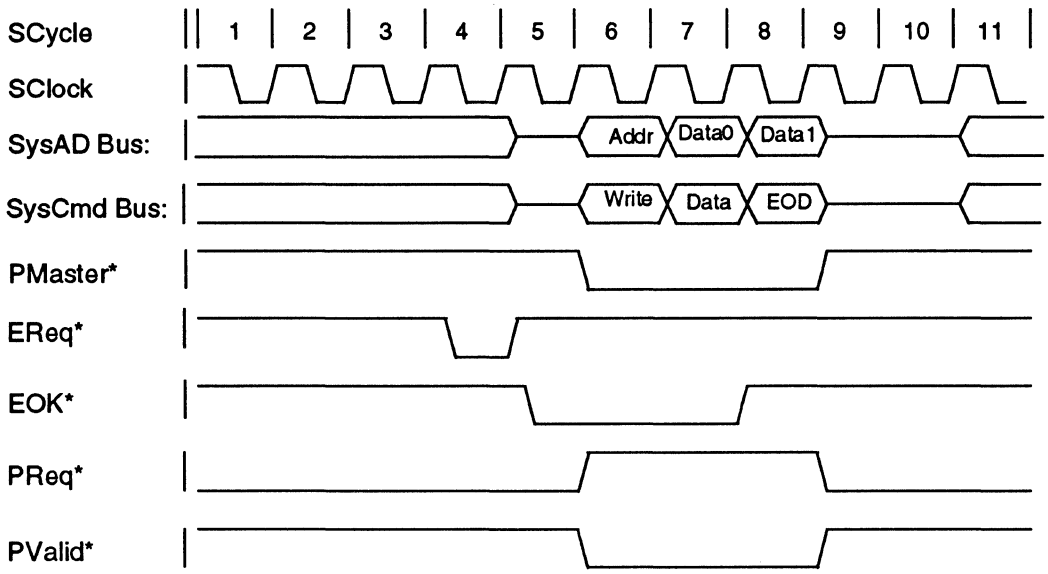


Figure 11-30 External Agent Gives Up Bus for One Processor Request

Multiple Drivers on the SysAD Bus

In most applications, the SysAD bus is a point-to-point connection between the processor and a bidirectional, registered transceiver located in an external agent. In this application, the SysAD bus has two possible drivers: the processor and the external agent.

However, an application may add additional drivers and receivers to the SysAD bus, allowing transmissions over the SysAD bus that bypass the processor. To accomplish this, the external agent(s) must coordinate its use of the SysAD bus by using arbitration handshake signals such as **EReq***, **PMaster*** and **PReq***.

To implement an independent transmission on the SysAD bus that does not involve the processor, the system executes the following sequence of actions:

1. The external agent(s) requests the SysAD bus by asserting **EReq***.
2. The processor releases the System interface to slave state.
3. The external agent(s) allows independent transmission over the SysAD bus, making certain the **EValid*** input to the processor is not asserted while the transmission occurs.
4. When the transmission is complete, the external agent(s) deasserts **EReq*** to return the system interface to master state.

To implement multiple drivers, separate **Valid** lines are required for non-processor chips to communicate.

Signal Codes

System interface commands and data identifiers are encoded in five bits on the SysCmd bus and transmitted between the processor and external agent during address and data cycles.

- When **SysCmd(4)** is a 0, the current cycle is an address cycle and **SysCmd(3:0)** contains a command.
- When **SysCmd(4)** is a 1, the current cycle is a data cycle and **SysCmd(3:0)** identifies data.

For commands and data identifiers associated with external requests, all bits and fields have a value or a suggested value.

For System interface commands and data identifiers associated with processor requests, reserved bits and reserved fields in the command or data identifier are undefined, except where noted.

For all System interface commands, the SysCmd bus specifies the system interface request type. The encoding of **SysCmd(4)** for system interface commands is Table 11-2.

Table 11-2 Encoding of System Interface Commands in **SysCmd(4)**

SysCmd(4)	Command
0	Address Cycle
1	Data Cycle

For address requests, the remainder of the SysCmd bus specifies the attributes of the address request, as follows:

- **SysCmd(3)** encodes the address request type.
- **SysCmd(2:0)** indicates the size of the address requests.

The encoding of **SysCmd(3:2)** for address requests is shown in Table 11-3.

Table 11-3 Encoding of SysCmd(3) and SysCmd(2) for Address Cycle

SysCmd(3)	Command	SysCmd(2)	Request Size
0	Read Request	0	Single data
1	Write Request	1	Block data

The encoding of **SysCmd(1:0)** for block or single address requests is shown in Tables 11-4 and 11-5, respectively.

Table 11-4 Encoding of **SysCmd(1:0)** *for Block Address Requests*

SysCmd(1:0)	Block Size
0	Two words.
1	Four words.
2	Eight words.
3	Reserved.

Table 11-5 Encoding of **SysCmd(1:0)** *for Single Address Requests*

SysCmd(1:0)	Data size.
0	One byte valid (byte)
1	Two bytes valid (halfword).
2	Three bytes valid (tribyte).
3	Four bytes valid (single word)

The encoding of **SysCmd(3:0)** for processor data identifiers is described in Table 11-6. The encoding of **SysCmd(3:0)** for external data identifiers is illustrated in Table 11-7.

Table 11-6 Encoding of SysCmd(3:0) for Processor Data Identifiers

SysCmd(3)	Last Data Element Indication
0	Last data element
1	Not the last data element
SysCmd(2)	**Reserved**
SysCmd(1)	***Reserved for*: Good Data Indication**
	Processor drives 0 (Data is error free)
SysCmd(0)	***Reserved for*: Data Checking Enable**
	Processor drives 1 (Disable data checking)

Table 11-7 Encoding of **SysCmd(3:0)** *for External Data Identifiers*

SysCmd(3)	Last Data Element Indication
0	Last data element
1	Not the last data element
SysCmd(2)	**Response Data Indication**
0	Data is response data
1	Data is not response data
SysCmd(1)	***Reserved for*: Good Data Indication**
0	Data is error free
1	Data is erroneous
SysCmd(0)	***Reserved for*: Data Checking Enable**
	Processor ignores this field (Suggested drive of 1, disable data checking)

NOTE: External read requests for processor resources are not supported in the V_R4300 processor.

11.4 Physical Addresses

Physical addresses are driven on all 32 bits (bits 31 through 0) of the SysAD bus during address cycles. Addresses associated with single read and write requests are aligned for the size of the data element; specifically, for single word requests, the low order two bits of the address are zero, for halfword requests, the low order bit of the address is zero. For byte and tribyte requests, the address provided is a byte address.

External agents returning read response data must support subblock ordering. Addresses associated with block read requests are aligned to the word of the desired data. The order in which data is returned in response to a processor block read request is:

- the word containing the addressed data word is returned first
- the remaining word(s) in the block are returned next, sequentially

Block writes are always block aligned.

JTAG Interface

12

The V_R4300 processor provides a boundary-scan interface that is compatible with Joint Test Action Group (JTAG) specifications, using the industry-standard JTAG protocol (IEEE Standard 1149.1/D6).

This chapter describes that interface, including descriptions of boundary scanning, the pins and signals used by the interface, and the Test Access Port (TAP).

12.1 What Boundary Scanning Is

With the evolution of ever-denser integrated circuits (ICs), surface-mounted devices, double-sided component mounting on printed-circuit boards (PCBs), and buried vias, in-circuit tests that depend upon making physical contact with internal board and chip connections have become more and more difficult to use. The greater complexity of ICs has also meant that tests to fully exercise these chips have become much larger and more difficult to write.

One solution to this difficulty has been the development of *boundary-scan* circuits. A boundary-scan circuit is a series of shift register cells placed between each pin and the internal circuitry of the IC to which the pin is connected, as shown in Figure 12-1. Normally, these boundary-scan cells are bypassed; when the IC enters test mode, however, the scan cells can be directed by the test program to pass data along the shift register path and perform various diagnostic tests. To accomplish this, the tests use the four signals described in the next section: **JTDI, JTDO, JTMS,** and **JTCK.**

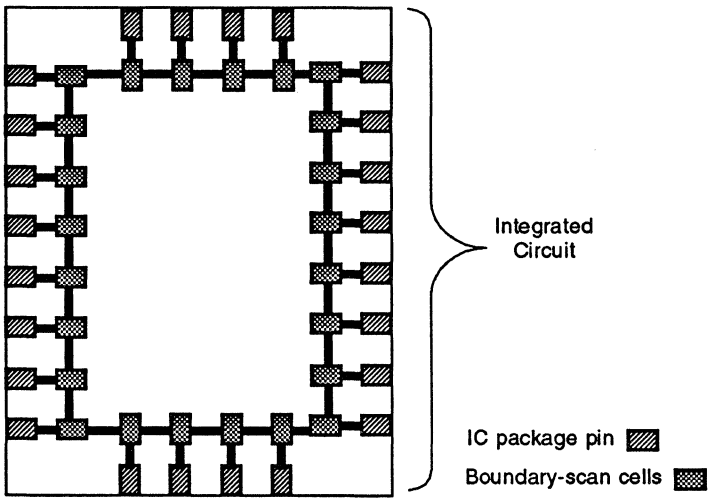


Figure 12-1 JTAG Boundary-scan Cells

12.2 Signal Summary

The JTAG interface signals are listed below and shown in Figure 12-2.

JTDI	JTAG serial data in
JTDO	JTAG serial data out
JTMS	JTAG test mode select
JTCK	JTAG serial clock input

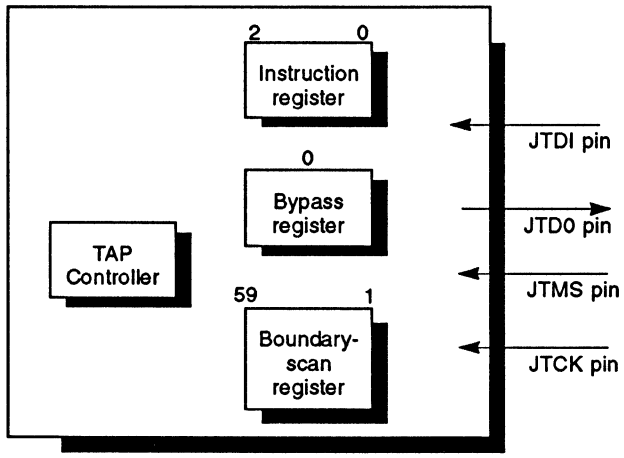


Figure 12-2 JTAG Interface Signals and Registers

The JTAG boundary-scan mechanism (referred to in this chapter as *JTAG mechanism*) allows testing of the connections between the processor, the printed circuit board to which it is attached, and the other components on the circuit board.

The JTAG mechanism does not provide any capability for testing the processor itself.

12.3 JTAG Controller and Registers

The processor contains the following JTAG controller and registers:

- *Instruction* register
- *Boundary-scan* register
- *Bypass* register
- Test Access Port (TAP) controller

The processor executes the standard JTAG EXTEST operation associated with External Test functionality testing.

The basic operation of JTAG is for the TAP controller state machine to monitor the JTMS input signal. When it occurs, the TAP controller determines the test functionality to be implemented. This includes either loading the JTAG instruction register (IR), or beginning a serial data scan through a data register (DR), listed in Table 12-1. As the data is scanned in, the state of the JTMS pin signals each new data word, and indicates the end of the data stream. The data register to be selected is determined by the contents of the *Instruction* register.

Instruction Register

The JTAG *Instruction* register includes three shift register-based cells; this register is used to select the test to be performed and/or the test data register to be accessed. As listed in Table 12-1, this encoding selects either the *Boundary-scan* register or the *Bypass* register.

Table 12-1 JTAG Instruction Register Bit Encoding

MSB..... LSB	Data Register
0 0 0	*Boundary-scan register (external test only)*
x x 1	*Bypass register*
x 1 x	*Bypass register*
1 x x	*Bypass register*
0 1 1	*Set Cache_Test_Mode sticky bit*

The *Instruction* register has two stages: shift register, and parallel output latch. Figure 12-3 shows the format of the *Instruction* register.

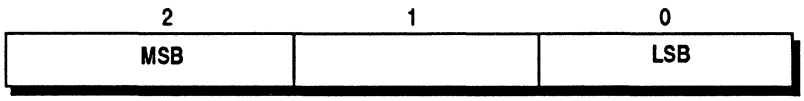


Figure 12-3 Instruction Register

Bypass Register

The *Bypass* register is 1 bit wide. When the TAP controller is in the Shift-DR (Bypass) state, the data on the **JTDI** pin is shifted into the *Bypass* register, and the *Bypass* register output shifts to the **JTDO** output pin.

In essence, the *Bypass* register is a short-circuit which allows bypassing of board-level devices, in the serial boundary-scan chain, which are not required for a specific test. The logical location of the *Bypass* register in the boundary-scan chain is shown in Figure 12-4. Use of the *Bypass* register speeds up access to boundary-scan registers in those ICs that remain active in the board-level test datapath.

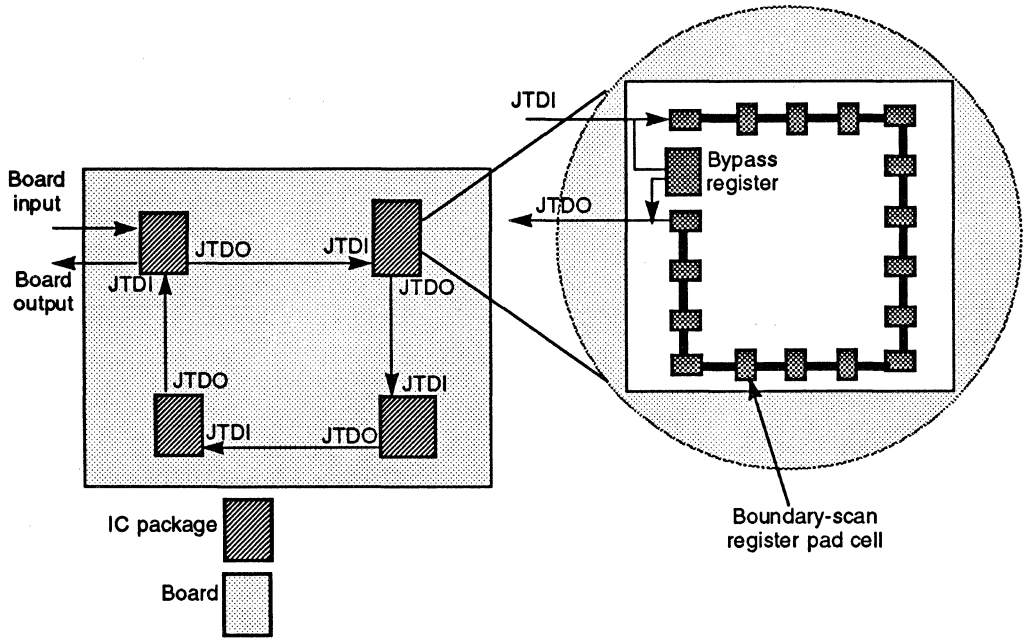


Figure 12-4 Bypass Register Operation

Boundary-Scan Register

The *Boundary Scan* register includes all of the inputs and outputs of the V_R4300 processor, except some clock and phase lock loop signals. The pins of the V_R4300 chip can be configured to drive any arbitrary pattern by scanning into the *Boundary Scan* register from the Shift-DR state. Incoming data to the processor is examined by shifting while in the Capture-DR state with the *Boundary Scan* register enabled.

The *Boundary-scan* register is a single, 58-bit-wide, shift register-based path containing cells connected to all input and output pads on the V_R4300 processor. Figure 12-5 shows the *Boundary-scan* register.

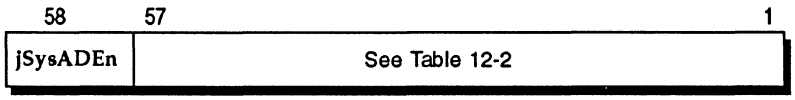


Figure 12-5 Format of the Boundary-scan Register

The most-significant bit, **jSysADEn**, is the JTAG output enable bit for all outputs of the processor. Output is enabled when this bit is set to 1 (default state).

The remaining 57 bits correspond to 57 signal pads of the processor.

At the end of this chapter, Table 12-2 lists the scan order of these 60 scan bits, starting from JTDI and ending with JTDO.

Test Access Port (TAP)

The Test Access Port (TAP) consists of the four signal pins: **JTDI**, **JTDO**, **JTMS**, and **JTCK**. Serial test data and instructions are communicated over these four signal pins, along with control of the test to be executed.

As Figure 12-6 shows, data is serially scanned into one of the three registers (*Instruction* register, *Bypass* register, or the *Boundary-scan* register) from the **JTDI** pin, or it is scanned from one of these three registers onto the **JTDO** pin.

The **JTDI** input feeds the least-significant bit (LSB) of the selected register, whereas the most-significant bit (MSB) of the selected register appears on the **JTDO** output.

The **JTMS** input controls the state transitions of the main TAP controller state machine.

The **JTCK** input is a dedicated test clock that allows serial JTAG data to be shifted synchronously, independent of any chip-specific or system clocks.

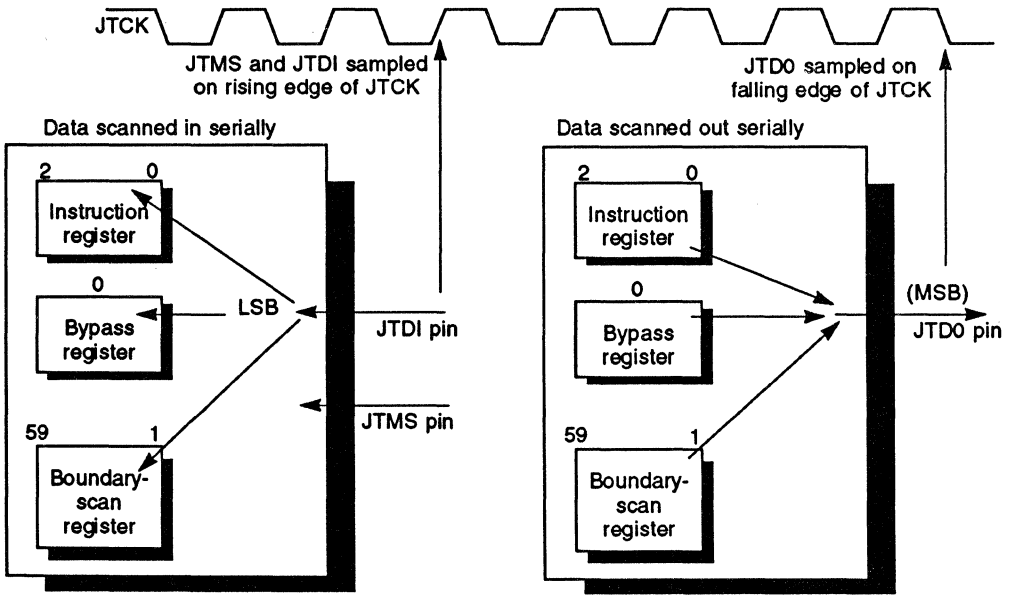


Figure 12-6 JTAG Test Access Port

Data on the **JTDI** and **JTMS** pins is sampled on the rising edge of the **JTCK** input clock signal. Data on the **JTDO** pin changes on the falling edge of the **JTCK** clock signal.

TAP Controller

The processor implements the 16-state TAP controller as defined in the IEEE JTAG specification.

Controller Reset

The TAP controller state machine can be put into Reset state by one of the following:

- assertion of the **ColdReset*** signal (low) resets the TAP controller
- keeping the **JTMS** input signal asserted through five consecutive rising edges of **JTCK** input sends the TAP controller state machine into its Reset state.

In either case, keeping **JTMS** asserted maintains the Reset state.

Controller States

The TAP controller has four states: Reset, Capture, Shift, and Update. They can reflect either instructions (as in the Shift-IR state) or data (as in the Capture-DR state).

- When the TAP controller is in the Reset state, the value 0x7 is loaded into the parallel output latch, selecting the *Bypass* register as default. The three most significant bits of the *Boundary-scan* register are cleared to 0, disabling the outputs.
- When the TAP controller is in the Capture-IR state, the value 0x4 is loaded into the shift register stage.
- When the TAP controller is in the Capture-DR (Boundary-scan) state, the data currently on the processor input and I/O pins is latched into the *Boundary-scan* register. In this state, the *Boundary-scan* register bits corresponding to output pins are arbitrary and cannot be checked during the scan out process.
- When the TAP controller is in the Shift-IR state, data is loaded serially into the shift register stage of the *Instruction* register from the **JTDI** input pin, and the MSB of the *Instruction* register's shift register stage is shifted onto the **JTDO** pin.
- When the TAP controller is in the Shift-DR (Boundary-scan) state, data is serially shifted into the *Boundary-scan* register from the **JTDI** pin, and the contents of the *Boundary-scan* register are serially shifted onto the **JTDO** pin.

- When the TAP controller is in the Update-IR state, the current data in the shift register stage is loaded into the parallel output latch.
- When the TAP controller is in the Update-DR (Boundary-scan) state, data in the *Boundary-scan* register is latched into the register parallel output latch. Bits corresponding to output pins, and those I/O pins whose outputs are enabled (by the three MSBs of the *Boundary-scan* register), are loaded onto the processor pins.

Table 12-2 shows the boundary scan order of the processor signals.

Table 12-2 Scan Order of V_R4300 Processor Pins

1. [JTDI]	2. SysAD<4>	3. SysAD<3>	4. SysAD<2>	5. SysAD<1>
6. SysAD<0>	7. PReq*	8. SysAD<31>	9. PValid*	10. SysAD<30>
11. EOK*	12. SysAD<29>	13. SysAD<28>	14. SysAD<27>	15. Int*<2>
16. NMI*	17. SysAD<26>	18. PMaster*	19. SysAD<25>	20. EReq*
21. SysCmd<0>	22. SysCmd<2>	23. Reset*	24. EValid*	25. SysCmd<2>
26. SysCmd<3>	27. ColdReset*	28. SysCmd<4>	29. DivMode<1>	30. SysAD<24>
31. DivMode<0>	32. SysAD<23>	33. Int*<3>	34. SysAD<22>	35. SysAD<21>
36. SysAD<20>	37. TestMode*	38. BypassPLL*	39. TClock	40. SyncOut
41. SysAD<19>	42. SysAD<18>	43. SysAD<17>	44. Int*<4>	45. SysAD<16>
46. SysAD<15>	47. SysAD<14>	48. SysAD<13>	49. SysAD<12>	50. SysAD<11>
51. SysAD<10>	52. Int*<0>	53. SysAD<9>	54. SysAD<8>	55. SysAD<7>
56. SysAD<6>	57. SysAD<5>	58. Int*<1>	59. jSysADEn	60. [JTDO]

12.4 Implementation-Specific Details

This section describes details of JTAG boundary-scan operation that are specific to the processor.

- The **MasterClock, MasterOut, SyncIn,** and **SyncOut** signal pads do not support JTAG.
- The update function occurs on the next rising edge of JTCK after the TAP controller enters the Update-DR state. (This is one-half a JTCK cycle later than the IEEE specification defines; the IEEE specification states that updates occur on the the falling edge of JTCK after the TAP controller enters Update-DR state).
- **MasterClock** must be running for reliable JTAG operation.

V_R4300 Processor Interrupts

13

Four types of interrupts are available on the V_R4300. These are:

- one non-maskable interrupt, NMI
- five external interrupts
- two software interrupts
- one timer interrupt

These are described in this chapter.

13.1 Nonmaskable Interrupt

The non-maskable interrupt is signaled by asserting the **NMI*** pin (low), forcing the processor to branch to the Reset Exception vector. This pin is latched into an internal register by the rising edge of **SClock**, as shown in Figure 13-2. An NMI can also be set by an external write through the **SysAD** bus. On the data cycle, **SysAD(22)** acts as the write enable for **SysAD(6)**, which is the value to be written as the interrupt.

NMI only takes effect when the processor pipeline is running. Thus NMI can be used to recover the processor from a software hang (for example, in an infinite loop) but cannot be used to recover the processor from a hardware hang (for example, no read response from an external agent). NMI cannot cause drive contention on the **SysAD** bus and no reset of external agents is required.

This interrupt cannot be masked.

The **NMI*** pin is latched by the rising edge of **SClock**, however the NMI exception occurs in response to the falling edge of the **NMI*** signal, and is not level-sensitive.

Figure 13-1 shows the internal derivation of the **NMI** signal. The **NMI*** pin is latched into an internal register by the rising edge of **SClock**. Bit 6 of the *Interrupt* register is then ORed with the inverted value of **NMI*** to form the nonmaskable interrupt.

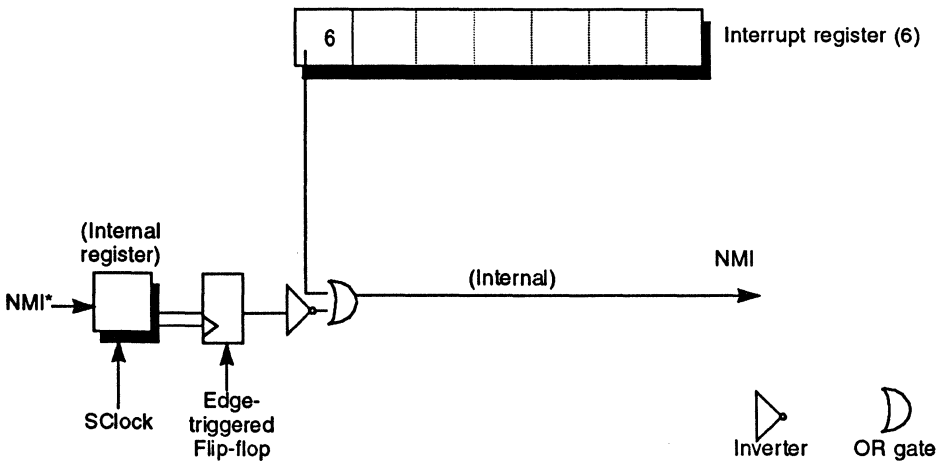


Figure 13-1 V_R4300 Nonmaskable Interrupt Signal

13.2 External Interrupts

External interrupts are set by asserting the external interrupt pins **Int*(4:0)**. They also may be set by an external write through the **SysAD** bus. During the data cycle, **SysAD(20:16)** are the write enables for bits **SysAD(4:0)**, which are the values to be written as interrupts.

These interrupts can be masked with the *IM*, *IE*, and *EXL* fields of the *Status* register.

13.3 Software Interrupt

Software interrupts use bits 1 and 0 of the interrupt pending, *IP*, field in the *Cause* register. These may be written by software, but there is no hardware mechanism to set or clear these bits.

These interrupts are maskable.

13.4 Timer Interrupt

The timer interrupt signal is bit 15 of the *Cause* register, which is bit 7 of the interrupt pending, *IP*, field. The timer interrupt is set whenever the value of the *Count* register equals the value of the *Compare* register.

This interrupt is maskable through the *IM* field of the *Status* register.

13.5 Asserting Interrupts

External writes to the CPU are directed to various internal resources, based on an internal address map of the processor. 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(20:16)** are the write enables for the five individual *Interrupt* register bits and **SysAD(4:0)** are the values to be written into these bits. This allows any subset of the *Interrupt* register to be set or cleared with a single write request. Figure 13-2 shows the mechanics of an external write to the *Interrupt* register, along with the nonmaskable interrupt described earlier.

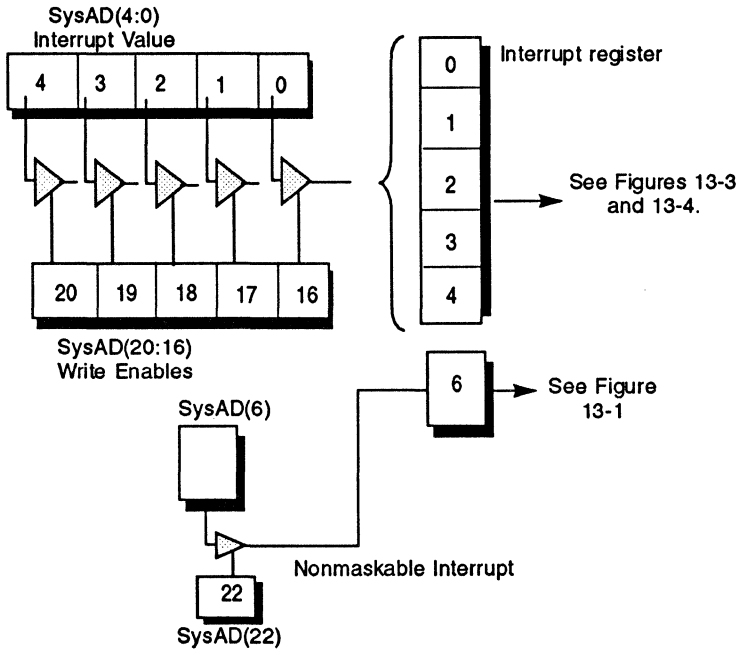


Figure 13-2 Interrupt Register Bits and Enables

Figure 13-3 shows how the V_R4300 hardware interrupts are readable through the *Cause* register.

- The timer interrupt signal, *IP7*, 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.

IP(1:0) of the *Cause* register, which are described in Chapter 5, are software interrupts. There is no hardware mechanism for setting or clearing the software interrupts.

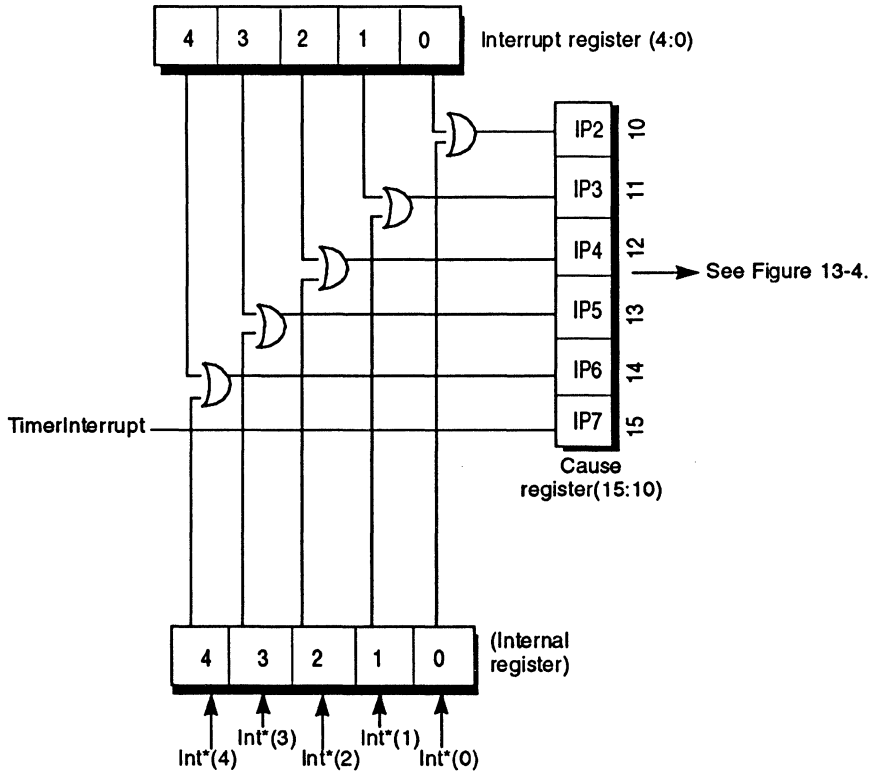


Figure 13-3 V_R4300 Hardware Interrupt Signals

Figure 13-4 shows the masking of the V_R4300 interrupt signals.

- *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 V_R4300 interrupt signal. The *EXL* bit in the *Status* register also enables these interrupts.

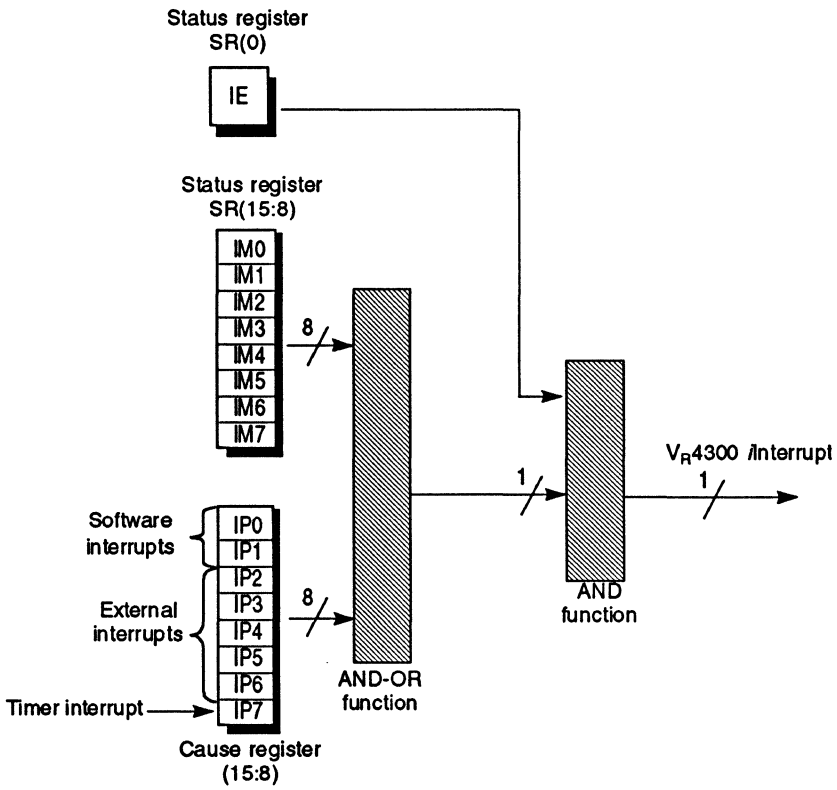


Figure 13-4 Masking of the V_R4300 Interrupts

Power Management

14

One of the objectives of the design of the V_R4300 processor is to minimize power dissipation in order to make the processor suitable for use in battery operated systems, as well as in environments where low power consumption and heat dissipation are desirable. To accomplish this, the V_R4300 has the following pair of complementary feature sets, described in this chapter:

- power reduction features
- power management features

14.1 Power Reduction Features

The V_R4300 processor incorporates the following power reduction features:

- 3.3-volt processor operation
- dynamic logic design
- a write-back cache, which reduces the System interface store traffic
- cache bank partitioning, by using segmented direct-mapped caches which reduce power consumption by enabling only that segment which contains the requested address
- cache prefetching, by making doubleword (two instructions) reads from the I-cache into a buffer; during typical sequential cache accesses the next instruction can be found in this buffer, thus reducing the frequency of I-cache access
- instruction micro-TLB, containing page table entries for the two most recently used instruction pages
- reduced die size, resulting in a reduction of required power.

Some of these features are described in greater detail below.

Dynamic Logic Design

The V_R4300 uses dynamic logic design, rather than static logic design, to reduce the transistor count, and the power dissipated. Processor power management in a dynamic logic design is just as effective as in a static logic design. In static logic design, the clock may be slowed to 0 MHz in reduced power mode, and state is retained in the processor. In dynamic logic design, state is saved off-processor in non-volatile storage (on disk, NVRAM) in power-off mode.

Cache Bank Partitioning

The V_R4300 splits the I-cache into four banks and the D-cache into two banks; only one of the banks in each cache need be powered up in any cycle. This saves power on every cache-access cycle.

Cache bank selection is done automatically by the two most-significant bits of the cache address.

Single Execution Unit

The single execution unit, supporting both floating point and integer instructions in hardware, is designed to reduce power consumption and simplify the hardware requirements while providing an adequate level of performance by maximizing use of each functional element. Separate integer and floating-point registers are provided but integer and floating-point instructions share a single datapath. For V_R4300 target applications, floating point performance is less critical than integer performance.

The execution unit uses a modular design approach to further reduce dynamic power consumption. Control logic is partitioned into small independent blocks responsible for a set of instructions. When relevant instructions are not in the instruction stream, the corresponding control blocks are inactive. Also, when functional elements in the data path are idle, they operate on a constant, carefully selected to minimize power dissipation, instead of on data from the bus.

Reduced Die Size

To reduce the die size (and, by implication, reduce the processor power requirements) the following techniques were used:

- high-density CMOS design rules
- 4-transistor RAM cells in caches
- reduced configurability
- cache and TLB sizes optimized for small die area(16 Kbyte I-cache, 8 Kbyte D-cache, 32-double-entry TLB)
- reduced physical address space (32-bit physical address)
- FPU and integer units are integrated into a single execution unit with shared resources, which reduces the size of the chip, and overall interconnect capacitance
- simplified pipeline and control logic, using a 5-stage pipeline, which results in reduced logic, interconnections, and an overall smaller device size.

14.2 Power Management Features

The chip supports three processor-level modes of operation: normal, reduced power, and power off. These modes allow processor power dissipation to be managed by system logic.

Typically a notebook system has many different levels of system power management. It is the responsibility of system logic to switch the processor between the three available modes in order to reflect the power management state of the system.

Normal Mode

In normal mode, pipeline clock (**PClock**) runs at a multiple of the system clock (**MasterClock**), as defined by the **DivMode(1:0)** pins. System interface clock (**SClock**) runs at one half the **PClock** speed, or at the same frequency as **MasterClock**.

In normal mode, the processor is estimated to dissipate 1.5 watts with a pipeline clock of 80 MHz.

Reduced Power Mode

In reduced power mode, processor clock (**PClock**) runs at one quarter of its normal speed, or one half of the **MasterClock** frequency. **SClock** is also reduced to one quarter of its normal mode frequency. Thus, if **MasterClock** is running at 40 MHz, **PClock** runs at 20 MHz, and **SClock** at 10 MHz.

Typically, chipset logic triggers reduced power mode when no user activity has been sensed after a certain period of time. In reduced power mode, power dissipation is estimated to be one quarter of normal.

Software moves the processor from one mode to the other by setting or resetting the *RP* bit in the *Status* Register. Setting reduced power mode (RP=1) results in the chip reducing the pipeline clock frequency and the System interface clock frequency by a factor of 4. For more information, see Chapter 9.

Software must be careful to execute a code sequence that guarantees the proper operation of the system upon setting or clearing the *RP* bit. Before going to RP mode, the system must first write any registers on any external agents that would be affected by the change in frequency: for instance, DRAM refresh counters. Next, the system must make certain the System interface is in an inactive state by executing an uncached read, which guarantees the write buffer is empty upon completion of the read. Only then can the system set or clear the *RP* bit. Finally, software must make certain that eight instructions immediately preceding and eight following the Move to Coprocessor register do not cause a cache miss, TLB miss, or exception of any kind.

Power Down Mode

In power down mode, the state of the processor is written to non-volatile storage. When the processor is returned to normal mode, all variable registers are restored to their former state.

In order to support system power-down mode, all internal state information necessary for restarting the processor from the point of power-down is read and write accessible. Prior to power-down, this state information must be read and saved off chip into non-volatile memory.

It is the system's responsibility to power-down the chip when the system is in idle state. The Load Link **LLbit** bit is not required to be saved since it is cleared by the cache invalidation during the power-up routine.

Cache content is not saved, and therefore the cache should be invalidated during the power-up routine and flushed during the power-down routine. The V_R4300 chip supports the CACHE and TLB operation instructions which invalidate all cache and TLB locations.

Electrical Specifications

15

This chapter describes the electrical specifications for the V_R4300 processor.

15.1 LVCMOS

The V_R4300 meets and exceeds the JEDEC standard for low-voltage CMOS-compatible VLSI digital circuits (LVCMOS).

15.2 DC Characteristics

Maximum Ratings

NOTE: Operation beyond the limits set forth in Table 15-1 may impair the useful life of the device.)

Table 15-1 Maximum Ratings

Parameter	Symbol	Test Conditions	Minimum	Maximum	Units
Supply Voltage	VCC		3.0	3.6	V
Input Voltage	VIN		-.5[1]	VCC + 0.5	V
Storage Temperature	TST		-65	+150	C
Operating Temperature	TC	Case temperature	0	+85	C

Note: Not more than one output should be shorted at a time. Duration of the short should not exceed 30 seconds.

(1) VIN Min. = -3.0V for pulse width less than 15ns.

Operating Parameters

Table 15-2 Operating Parameters

Parameter	Symbol	Conditions	62.5 MHz		Units
			Minimum	Maximum	
Output HIGH Voltage	VOH	VCC = Min. IOH=-4ma	2.4		V
Clock Output HIGH Voltage[3]	VOHC	VCC = Min. IOH=-4ma	2.7		V
Output LOW Voltage	VOL	VCC = Min. IOL=4ma		.4	V
Input HIGH Voltage[2]	VIH		2	VCC+.5	V
Input LOW Voltage[1,2]	VIL		-.5(1)	.8	V
MasterClock Input HIGH Voltage	VIHC		0.8 VCC	VCC+.5	V
MasterClock Input LOW Voltage	VILC		-.5(1)	0.2 VCC	V
Input Capacitance	CIn			10	pF
Output Capacitance	COut			10	pF
Operating Current	ICC	VCC = 3.0V, TC=0C		0.67	A
Input Leakage	ILeak			10	µA
Input/Output Leakage	IOLeak			20	µA

Note:

(1) VIL Min. = -3.0V for pulse width less than 15 ns.

(2) Except for **MasterClock** input

(3) Applies to **TClock** output

15.3 AC Characteristics

All output timings assume a 50 pf capacitive load. Output timings should be derated where appropriate, as listed in Table 15-3.

MasterClock and Clock Parameters

Table 15-3 MasterClock and Clock Parameters

Parameter	Symbol	Test Conditions	62.5 MHz		Units
			Minimum	Maximum	
MasterClock High	t_{MCHigh}	Transition ≤ 5ns	4		ns
MasterClock Low	t_{MCLow}	Transition ≤ 5ns	4		ns
MasterClock Freq[1]			20	62.5	MHz
MasterClock Period	t_{MCP}		16	50	ns
Clock Jitter	$t_{MCJitter}$			+/-500	ps
MasterClock Rise Time	t_{MCRise}			4	ns
MasterClock Fall Time	t_{MCFall}			4	ns
JTAG Clock Period	$t_{JTAGCKP}$		$4*t_{MCP}$		ns

Note:

(1) Operation of V_R4300 is only guaranteed with the Phase Lock Loop enabled.

System Interface Parameters

Table 15-4 System Interface Parameters

Parameter	Symbol	62.5 MHz		Units
		Minimum	Maximum	
Data Output [1,2,3]	t_{DO}	2	8	ns
Data Setup [3]	t_{DS}	3.5		ns
Data Hold [3]	t_{DH}	1.5		ns
Clock Rise Time[4]	t_{CORise}		4	ns
Clock Fall Time[4]	t_{COFall}		4	ns
Clock High Time[4]	t_{COHigh}	4		ns
Clock Low Time[4]	t_{COLow}	4		ns

Note:

(1) Timings are measured from 1.5V of the SClock to 1.5V of signal.

(2) Capacitive load for all output timings besides Status is 50pf.

(3) Data Output, Data Setup and Data Hold apply to all logic signals driven out of o driven into the V_R4300 on the system interface. Clocks are specified separately.

(4) TCLock.

Capacitive Load Deration

Table 15-5 Capacitive Load Deration

Parameter	Symbol	62.5 MHz		Units
		Minimum	Maximum	
Load Derate	CLD		2	ns/25pF

Packaging

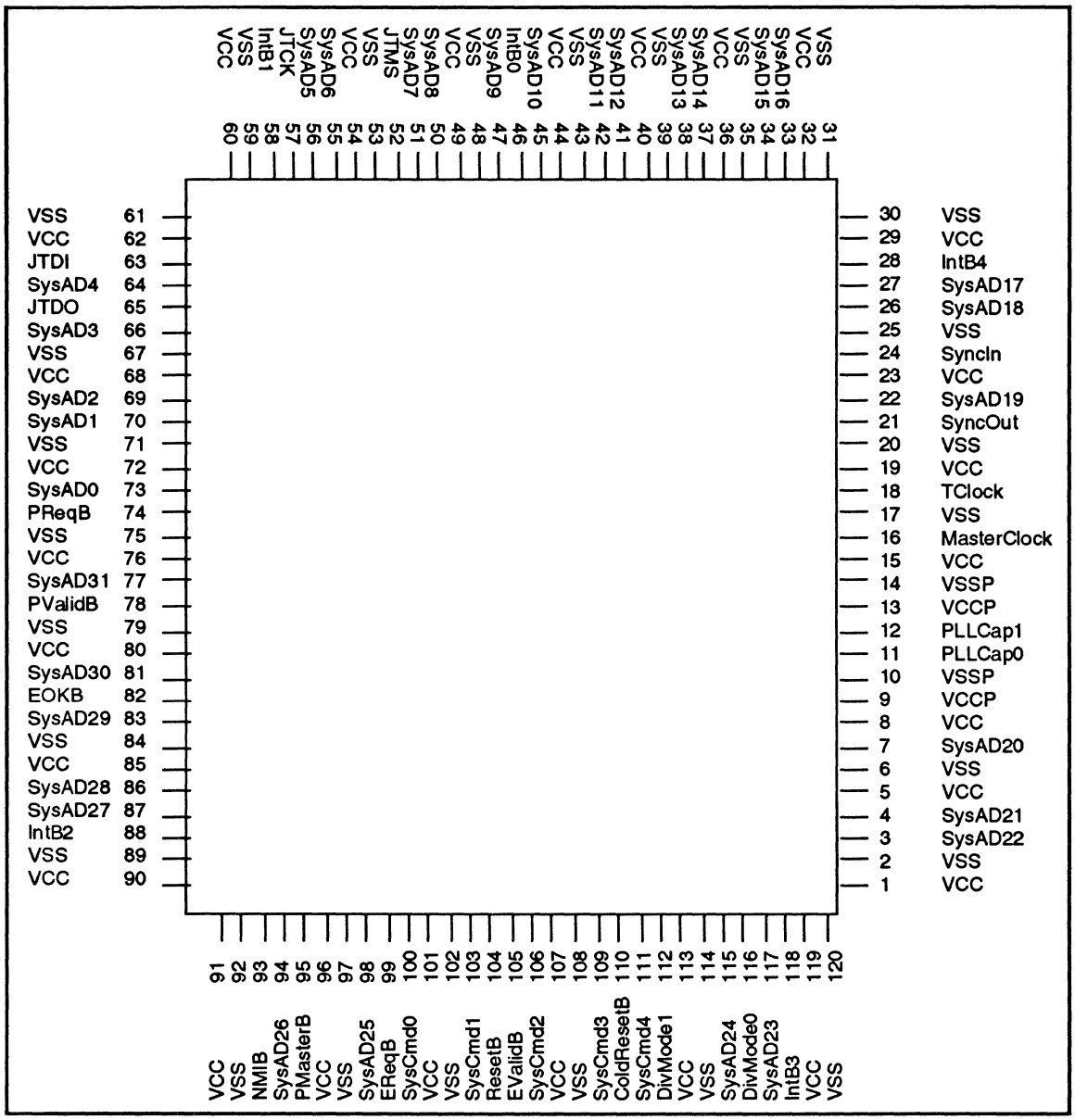
16

This chapter describes the packaging for the V_R4300 processor. Signals with a "B" suffix, such as ColdResetB, are low active. The "B" is synonymous with the asterisk, "*", used elsewhere in this manual.

120-pin PQFP Pin-out

VCC	31	VSS	61	VSS	91	VCC
VSS	32	VCC	62	VCC	92	VSS
SysAD22	33	SysAD16	63	JTDI	93	NMIB
SysAD21	34	SysAD15	64	SysAD4	94	SysAD26
VCC	35	VSS	65	JTDO	95	PMasterB
VSS	36	VCC	66	SysAD3	96	VCC
SysAD20	37	SysAD14	67	VSS	97	VSS
VCC	38	SysAD13	68	VCC	98	SysAD25
VCCP	39	VSS	69	SysAD2	99	EReqB
VSSP	40	VCC	70	SysAD1	100	SysCmd0
PLLCAP0	41	SysAD12	71	VSS	101	VCC
PLLCAP1	42	SysAD11	72	VCC	102	VSS
VCCP	43	VSS	73	SysAD0	103	SysCmd1
VSSP	44	VCC	74	PReqB	104	ResetB
VCC	45	SysAD10	75	VSS	105	EValidB
MasterClock	46	IntB0	76	VCC	106	SysCmd2
VSS	47	SysAD9	77	SysAD31	107	VCC
TClock	48	VSS	78	PValidB	108	VSS
VCC	49	VCC	79	VSS	109	SysCmd3
VSS	50	SysAD8	80	VCC	110	ColdResetB
SyncOut	51	SysAD7	81	SysAD30	111	SysCmd4
SysAD19	52	JTMS	82	EOKB	112	DivMode1
VCC	53	VSS	83	SysAD29	113	VCC
SyncIn	54	VCC	84	VSS	114	VSS
VSS	55	SysAD6	85	VCC	115	SysAD24
SysAD18	56	SysAD5	86	SysAD28	116	DivMode0
SysAD17	57	JTCK	87	SysAD27	117	SysAD23

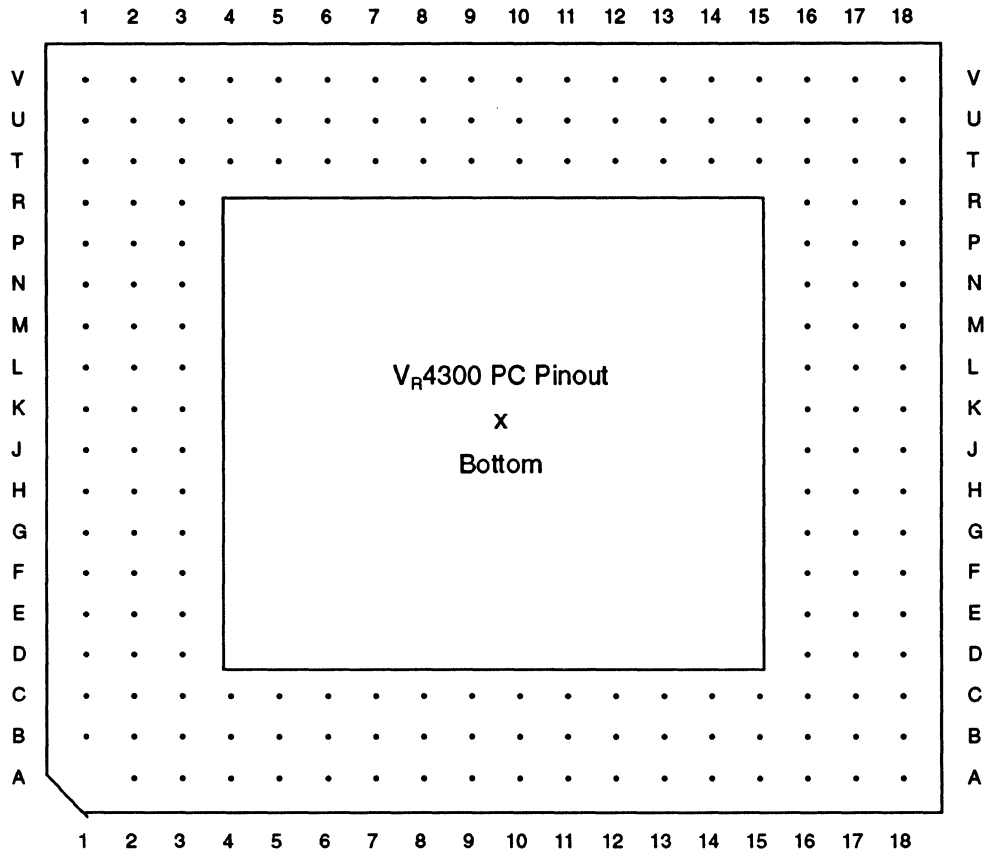
:0 Pin PQFP Physical Pin Location



179-pin PGA Pin-out

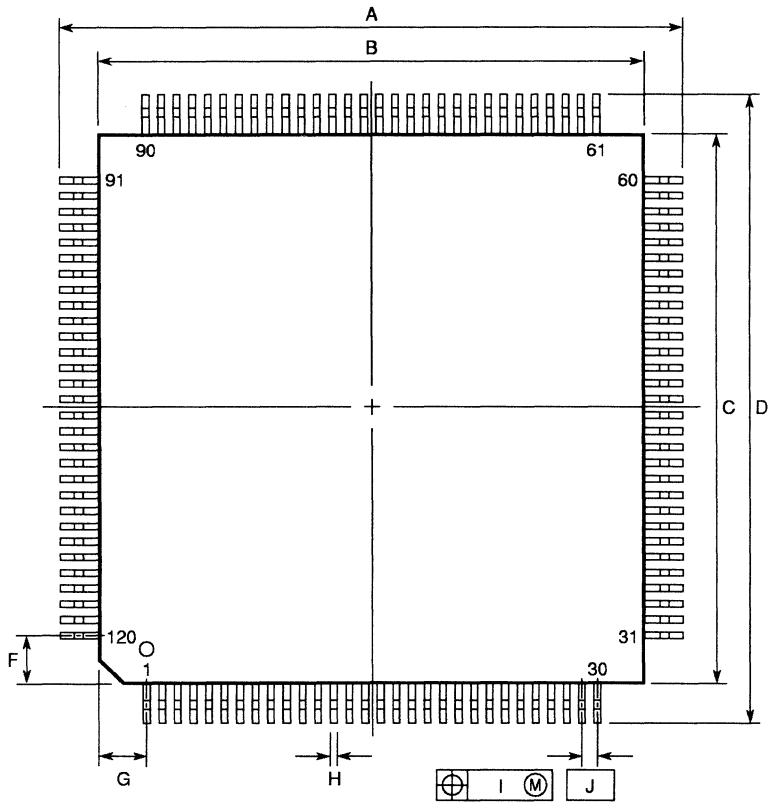
A1	No pin	C10	SysAD9	K1	VCC	T10	SysCmd2
A2	VCC	C11	NC	K2	SysAD31	T11	ColdResetB
A3	VSS	C12	SysAD11	K3	NC	T12	DivMode1
A4	VCC	C13	SysAD13	K16	VSSP	T13	NC
A5	JTCK	C14	SysAD16	K17	VCCP	T14	DivMode0
A6	VSS	C15	NC	K18	VSS	T15	IntB3
A7	VCC	C16	NC	L1	VSS	T16	NC
A8	VSS	C17	NC	L2	NC	T17	NC
A9	VCC	C18	VSS	L3	PValidB	T18	VCC
A10	VSS	D1	VSS	L16	SysAD20	U1	VCC
A11	VCC	D2	JTDI	L17	TestModeB	U2	NC
A12	VSS	D3	NC	L18	VCC	U3	NC
A13	VCC	D16	NC	M1	VCC	U4	NC
A14	VSS	D17	NC	M2	SysAD30	U5	PMasterB
A15	NC	D18	VCC	M3	EOKB	U6	NC
A16	VCC	E1	NC	M16	SysAD21	U7	EReqB
A17	VSS	E2	JTDO	M17	ByPassPLLB	U8	NC
A18	VSS	E3	SysAD4	M18	VSS	U9	EValidB
B1	VSS	E16	NC	N1	VSS	U10	NC
B2	NC	E17	NC	N2	NC	U11	SysCmd3
B3	NC	E18	SysAD17	N3	NC	U12	NC
B4	NC	F1	VCC	N16	SysAD22	U13	SysCmd4
B5	NC	F2	NC	N17	NC	U14	SysAD24
B6	SysAD6	F3	SysAD3	N18	VCC	U15	
B7	NC	F16	IntB4	P1	SysAD29	U16	NC
B8	JTMS	F17	SysAD18	P2	SysAD28	U17	NC
B9	SysAD8	F18	VSS	P3	SysAD27	U18	VSS
B10	IntB0	G1	VSS	P16	NC	V1	VSS
B11	SysAD10	G2	SysAD1	P17	NC	V2	VSS
B12	SysAD12	G3	SysAD2	P18	VSS	V3	VCC
B13	SysAD14	G16	SyncIn	R1	VCC	V4	VSS
B14	SysAD15	G17	NC	R2	IntB2	V5	SysAD25
B15	NC	G18	NC	R3	NC	V6	VCC
B16	ResetB	H1	VCC	R16	NC	V7	VSS
B17	NC	H2	SysAD0	R17	NC	V8	VCC
B18	VCC	H3	NC	R18	VSS	V9	VSS
C1	VCC	H16	SysAD19	T1	VSS	V10	VCC
C2	NC	H17	SyncOut	T2	NC	V11	VSS
C3	NC	H18	VSS	T3	NC	V12	VCC
C4	NC	J1	VSS	T4	NC	V13	VSS
C5	NC	J2	NC	T5	NMIB	V14	NC
C6	IntB1	J3	PReqB	T6	SysAD26	V15	SysAD23
C7	SysAD5	J16	TClock	T7	SysCmd0	V16	VSS
C8	NC	J17	MasterClock	T8	SysCmd1	V17	VCC
C9	SysAD7	J18	VCC	T9	NC	V18	VSS

179-Pin PGA Physical Pin Location

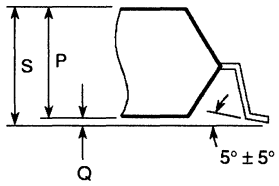


120-Pin Plastic QFP (Quad Flat Package)

Item	Millimeters	Inches
A	32.0 ±0.3	1.260 ±.012
B	28.0 ±0.2	1.102 $^{+.009}_{-.008}$
C	28.0 ±0.2	1.102 $^{+.009}_{-.008}$
D	32.0 ±0.3	1.260 ±.012
F	2.4	.094
G	2.4	.094
H	0.35 ±0.10	.014 $^{+.004}_{-.005}$
I	0.15	.006
J	0.8 (TP)	.031 (TP)
K	2.0 ±0.2	.079 $^{+.009}_{-.008}$
L	0.8 ±0.2	.031 $^{+.009}_{-.008}$
M	0.15 $^{+0.10}_{-0.05}$	.006 $^{+.004}_{-.003}$
N	0.10	.004
P	3.4	.134
Q	0.1 ±0.1	.004 ±.004
S	3.5 max	.138 max



Enlarged detail of lead end



P120GD-80-LBB,MBB

95CL-0585B (

PLL Passive Components

A

The Phase Locked Loop circuit requires several passive components for proper operation, which are connected to **PLLCap0**, **PLLCap1**, **VccP**, and **VssP**, as illustrated in Figure A-1.

In addition, the capacitors for **PLLCap0** (**Cp**) and **PLLCap1** (**Cp**) can be connected to either **VssP** (as shown), **VccP**, or one to **VssP** and one to **VccP**. Note that **C2** and the **Cp** capacitors are only incorporated into the 120-pin PFQP package as surface-mounted chip capacitors.

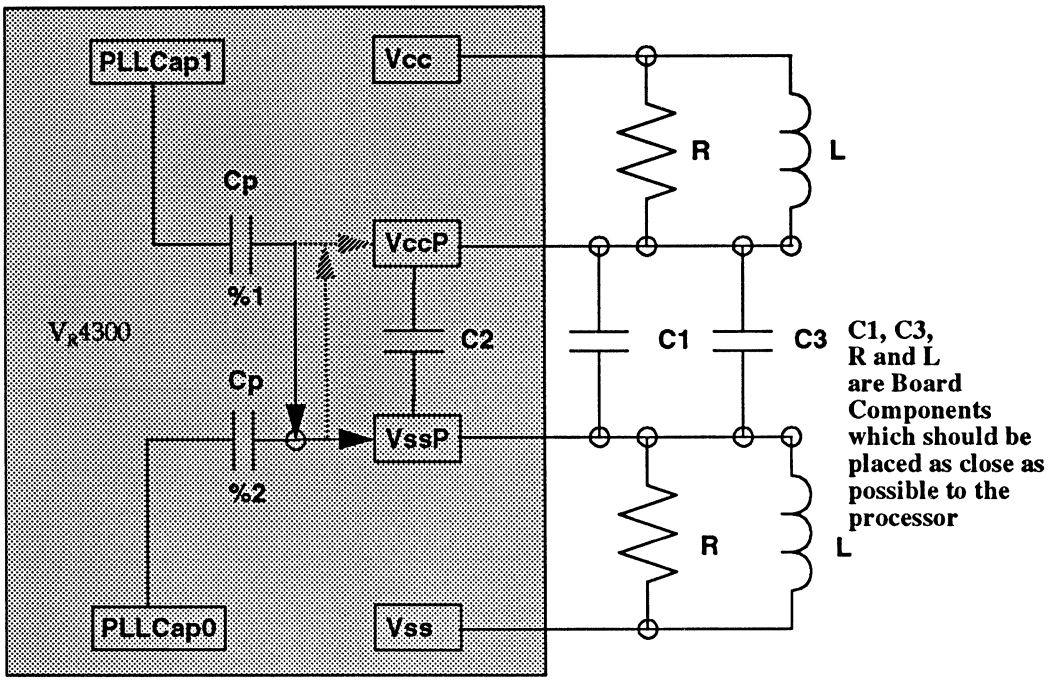


Figure A-1 PLL Passive Components for the V_R4300

Values:

R = 5 ohms

C1 = 1 nF

C2 = 82 nF

C3 = 10 µF

C4 = 470 pF

The inductors (L) can be used as alternatives to the resistors (R) to filter the power supply.

It is essential to isolate the analog power and ground for the PLL circuit (**VccP/VssP**) from the regular power and ground (**Vcc/Vss**).

Figure A-2 shows a top view of the 120-pin PQFP package with chip capacitors mounted on the printed circuit board (PCB).

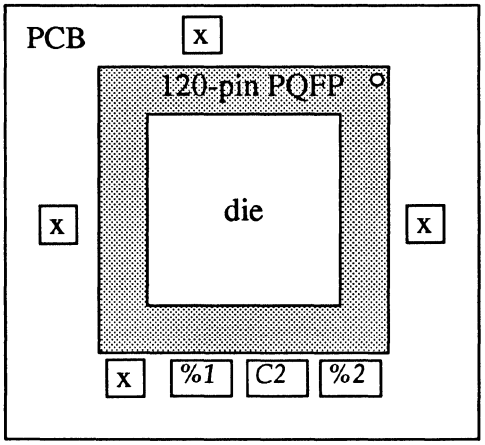


Figure A-2 V_R4300 120-Pin Package

V_R4300 Coprocessor 0 Hazards

B

The contents of the System Coprocessor registers and the TLB affect the operation of the processor in many ways. For instance, an instruction that changes CP0 data also affects subsequent instructions that use the data.

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

There is a *required-data dependence* between an instruction that changes a register or TLB entry (a *writer*) and the next instruction that uses it (a *user*). (A writer can write multiple data items, forming multiple writer/user pairs.) The writer/user instruction pair places a *hazard* on the data if there must be a delay between the time the writer instruction writes the data, and the user instruction can use the data.

In addition to instructions, events can be writers and users of CP0 information. For instance, an exception writes information to CP0 registers and events that occur for every instruction, like an instruction

fetch, use CP0 information. Therefore, when manipulating CP0 contents, the systems programmer must identify hazards and write code that avoids these hazards.

Table B-1 describes how to identify and avoid hazards, listing instructions and events that use CP0 registers and the TLB. This table also tells when written information is available (column 3) and when this latest information can actually be used (column 2). *Exception event writer timing* refers to the instruction identified with the exception; *user event timing* information is the pipestage of each instruction during which the user event uses the data. In the case of a hazard, the number of instructions required between a writer and user is:

available_stage - (use_stage + 1)

To identify a hazard, look for an instruction/event writer/user pair that has a required-data dependence and use the timing information in the table to calculate the delay required between the writer and user. If no delay is required, there is no hazard. If there is a hazard, place enough instructions between the writer and user so that the written information is available or effective when the user needs it.

NOTE: Any instructions inserted between a writer/reader pair with a hazard must not depend on or modify the data creating the hazard (for example NOP instructions may be used).

The following steps are used to determine a hazard delay:

1. Find the pipeline stage of the *writer* instruction in which the result is available. For example, the MTC0 instruction writes a CP0 general register, and the new value is available at stage 7.
2. Find the pipeline stage in which the *user* instruction reads or uses the data item that the writer changes. The TLBWR instruction, for example, uses different registers through different stages; all source register values must be stable by stage 5 and remain unchanged through stage 8.
3. Calculate the number of instructions that must be inserted between the hazardous pair, by using this formula: *available_stage - (use_stage + 1)*. For example, with an MTC0/TLBWR pair, MTC0 data is available at stage 7, and TLBWR data must be stable by stage 5 so the computation is: 7 - (5 + 1) = 1. This means 1 instruction must be inserted between the MTC0 and TLBWR. If the result of the computation is less than or equal to zero, there is no hazard and no instructions are required between the pair.

Table B-1 V_R4300 Coprocessor 0 Data Writer and User Timing

Instruction or Event	CP0 Data Used, Stage Used		CP0 Data Written, Stage Available	
MTC0 / DMTC0			CPR[0,rd]	7γδ
MFC0 / DMFC0	CPR[0,rd]	4βγ		
TLBR	Index, TLB	5-7	PageMask, EntryHi, EntryLo0, EntryLo1	8
TLBWI TLBWR	Index or Random, PageMask, EntryHi, EntryLo0, EntryLo1	5-8	TLB	8
TLBP	PageMask, EntryHi	3-6	Index	7
ERET	EPC or ErrorEPC, TLB	4	Status[EXL, ERL]	4-8α
	Status	3	LLbit	7
Index Load Tag			TagLo, TagHi, ECC	8βε
Index Store Tag	TagLo, TagHi, ECC	8ε		
CACHE Hit ops			Status[CH]	8ε
CACHE ops	cache line (see note)	ε	cache line (see note)	ε
Load/Store	EntryHi.ASID Status[KSU, EXL, ERL, RE], Config[K0, DB], TLB	4		
	Config[SB]	7		
	WatchHi, WatchLo	4-5		
Load/Store exception			EPC, Status, Cause, BadVaddr, Context, XContext	8
Instruction fetch exception			EPC, Status	8
			Cause, BadVAddr, Context, XContext	4
Instruction fetch	EntryHi[ASID], Status[KSU, EXL, ERL, RE], Config[K0, IB]	0α		
	Config.SB	3		
	TLB (mapped addresses)	2		
Coproc. usable test	Status[CU, KSU, EXL, ERL]	2		
Interrupt signals sampled	Cause[IP], Status[IM, IE, EXL, ERL]	3		

EntryHi.ASID refers to the *ASID* field of the *EntryHi* register.
Config[K0, DB] refers to the *K0* and *DB* fields of the *Config* register.

α The *EXL* and *ERL* bits in the *Status* register are permanently cleared in stage 8, if no exceptions abort the ERET. However the effect of clearing them is visible to an instruction fetch starting in stage 4, so the "returned to" instructions use the modified values in the *Status* register.

β Only one instruction is needed to separate Index Load Tag and MFC0 Tag, even though table timing indicates otherwise.

γ An MTC0 of a CPR must not be immediately followed by MFC0 of the same CPR.

δ With an MTC0 to *Status* that modifies *KSU* and sets *EXL* or *ERL*, it is possible for the five instructions following the MTC0 to be executed incorrectly in the new mode, and not correctly in the kernel mode. This can be avoided by setting *EXL* first, and only later changing the value of *KSU*.

ε There must be two non-load, non-CACHE instructions between a store and a CACHE instruction directed to the same primary cache line as the store.

Table B-2 lists some hazard conditions, and the number of instructions that must come between the writer and the user. The table shows the data item that creates the hazard, and the calculation for the required number of intervening instructions.

Table B-2 CP0 Hazards and Calculated Delay Times.

Writer	→	User	Hazard On	Instructions Between	Calculation
TLBWR/ TLBWI	→	TLBP	TLB entry	3	8-(4+1)
TLBWR/ TLBWI	→	load/store using new TLB entry	TLB entry	3	8-(4+1)
TLBWR/ TLBWI	→	I-fetch using new TLB entry	TLB entry	5	8-(2+1)
MTCO Status[CU]	→	Coprocessor instruction needs CU set	Status[CU]	4	7-(2+1)
TLBR	→	MFC0 EntryHi	EntryHi	3	8-(4+1)
MTC0 EntryLo0	→	TLBWR/TLBWI	EntryLo0	1	7-(5+1)
TLBP	→	MFC0 Index	Index	2	7-(4+1)
MTC0 EntryHi	→	TLBP	EntryHi	1	7-(5+1)
MTC0 EPC	→	ERET	EPC	2	7-(4+1)
MTC0 Status	→	ERET	Status	3	7-(3+1)
MTC0 Status[IE]	→	instruction interrupted[a]	Status[IE]	3	7-(3+1)

a. You cannot depend on a delay in effect if the instruction execution order is changed by exceptions. In this case, for example, the minimum delay for IE to be effective is the *maximum* delay before a pending, enabled interrupt can occur.

Cache Tests

C

Cache test mode allows an IC tester or external logic to read and write the internal instruction and data cache memories directly, through the V_R4300 I/O pins.

C.1 Cache Memory Description

V_R4300 cache memories are composed of four parts:

- instruction cache data
- instruction cache tag
- data cache data
- data cache tag

Instruction Cache Data

Instruction cache data is composed of four banks of static RAM, each bank of which is 256 rows deep.

Each row contains two 64-bit words, for a total of 128 columns per row. A 2:1 multiplexer is used to connect the double-bit columns to a 64-bit staging register, which in turn is multiplexed onto a 32-bit instruction bus.

Instruction Cache Tag

The instruction cache tag is composed of a 42-bit-wide by 256-bit-deep static RAM. A 2:1 multiplexer connects the double-bit columns to a 21-bit tag bus. The 21 bits include 20 bits of physical page address, plus a valid bit.

Data Cache Data

Data cache data is composed of two banks of static RAM, each bank of which is 256 rows deep.

Each row contains two 64-bit words, for a total of 128 columns per row. A 2:1 multiplexer is used to connect the double-bit columns to a 64-bit bus.

Data Cache Tag

Data cache tag is composed of a 44-bit-wide by 256-bit-deep static RAM. Each 44-bit row consists of two 22 bit tags. Each tag contains a 20-bit physical page address, a valid bit, and a dirty bit.

C.2 Test Mode Description

In order to conserve die area, existing data paths are used to move data between the caches and the I/O pins. Since the data paths are shared, access of the instruction and data cache memories can be pipelined. This appendix contains timing diagrams illustrating read and write transactions.

Since the internal datapath of the processor is 64 bits wide and the System interface bus is 32 bits wide, data is transferred on the bus in two consecutive system clock cycles, but is written to and read from the cache internally in one processor clock cycle. Since the caches are written with 64-bit data, the index must be aligned to double word addresses (that is, **SysAD(2:0)** = 000), and tag indexes must be aligned to block addresses, which is 16 bytes for data cache and 32 bytes for instruction cache (that is, **SysAD(3:0)** = 0000 for the data cache, **SysAD(4:0)** = 00000 for the instruction cache).

For both reads and writes, the addressed doubleword of data is presented on the bus in little-endian order; the least significant bits 31:0 are transferred in Data0 and the most significant bits 63:32 are transferred in Data1.

For reads, address and command are clocked in and three cycles later data is clocked out of the processor for both the data and tag portions of the instruction and data caches. For writes, DCData, DCTag, and ICData portion behave similarly: the index is followed by the write data and writes can be issued back-to-back. ICTag writes require one dead cycle, however, between successive writes.

Before testing can begin, the internal and external clock must be stable, as described in Chapter 9.

To enable the Cache Test Mode, JTAG commands are used to set the internal Cache Test Mode Sticky bit. Upon power up the following sequence of events enables Cache Test Mode:

- **DivMode(1:0)** is set to 10, divide by two mode
- **ColdReset*** is asserted (low)
- **Reset*** is asserted (low)
- **ColdReset*** is deasserted (high) after the clocks are stable
- the value of 110_2 is shifted into JTAGIR register (setting the Cache Test Mode Sticky register)

In cache test mode most internal control logic is reset and the processor pipeline is stalled. During cache test, the processor will only drive the SysAD bus in the fourth (and fifth for Data1) cycle of a read command. At all other times these inputs must be driven by external logic to a valid logic level to prevent damage to the input buffers of the processor.

C.3 Test Mode Commands

The **Int*** pins are used as the command bus for cache test mode. Here is a list of available commands:

- **Int*(2:0)** = 000 indicates a data cache data array read.
- **Int*(2:0)** = 001 indicates a data cache tag array read.
- **Int*(2:0)** = 010 indicates a data cache data array write.
- **Int*(2:0)** = 011 indicates a data cache tag array write.
- **Int*(2:0)** = 100 indicates an instruction cache data array read.
- **Int*(2:0)** = 101 indicates an instruction cache tag array read.
- **Int*(2:0)** = 110 indicates an instruction cache data array write.
- **Int*(2:0)** = 111 indicates an instruction cache tag array write.
- **Int*(4)** = 1 indicates a NOP.

Int*(4) should be used as command strobe, for instance to ensure a read or write command is only valid if this pin is low. This allows idle/NOP cycles in test mode.

At least six NOP commands must be issued to initialize the internal state of the processor once cache test mode is activated.

C.4 Cache Memory Address

In a cache memory access, the cache index is clocked in from the SysAD bus and held in a cache Index register. **Int*(3)** is used as an enable for the *Index* register; when **Int*(3)** is low a new index is read from the SysAD bus. The SysAD bus bits used are described below.

- For the instruction cache data portion, **SysAD(13:3)** are used to index the cache.
- For the instruction cache tag portion, **SysAD(13:5)** are used to index the cache.
- For the data cache data portion, **SysAD(12:3)** are used to index the cache.
- For the data cache tag portion, **SysAD(12:4)** are used to index the cache.

C.5 Cache Read

After a cache read command is clocked in, the processor reads the selected cache. External logic must stop driving the SysAD bus one cycle after it issues the read command; after four cycles the processor drives the SysAD bus for one system cycle and provides the requested data.

When the target is instruction or data cache data, a 64-bit doubleword is read to ensure all bits of the SysAD bus are driven. If the target is either instruction or data cache tag, only a subset of the SysAD bus is driven.

When reading the data cache tag, the SysAD bus contains the following values:

- **SysAD[31:12]** 20 Tag Bits
- **SysAD[11]** Valid Bit
- **SysAD[10]** Dirty Bit
- **SysAD[9:0]** Index

When reading the instruction cache tag. the SysAD bus contains the following values:

- **SysAD[31:12]** 20 Tag Bits
- **SysAD[11]** Valid Bit
- **SysAD[10:0]** Index

Figures C-1 and C-2 illustrate the cache read timing for cache data and cache tags, respectively.

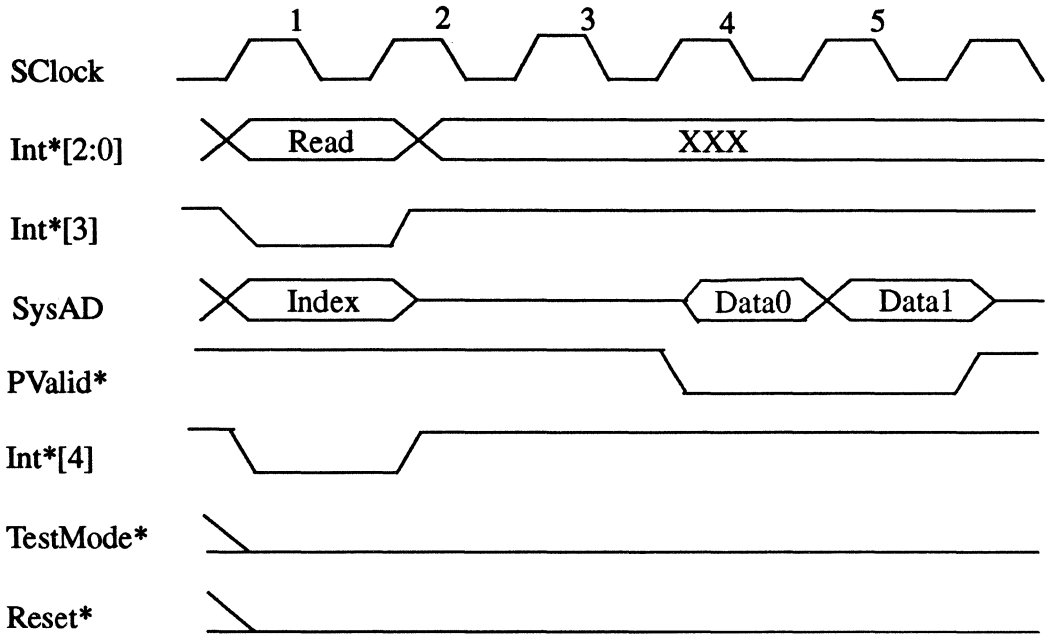


Figure C-1 Instruction and Data Cache Data Read Timing.

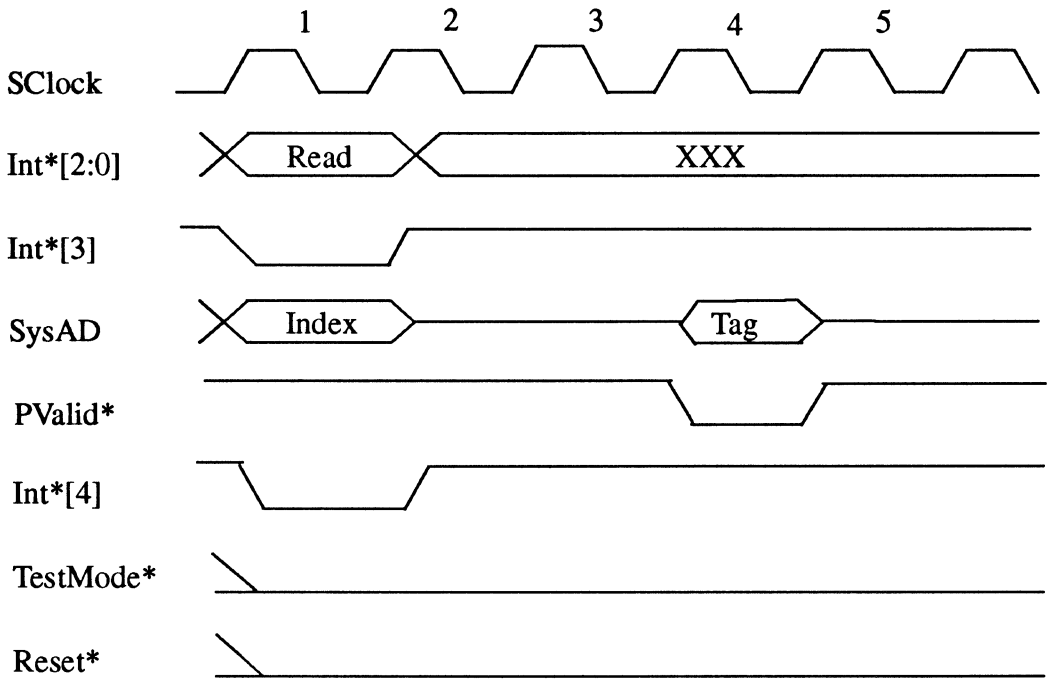


Figure C-2 Instruction and Data Cache Tag Read Timing

C.6 Cache Write

A cache write consists of a write command on the **Int*** pins, followed by a write data on the next cycle. Pipelined writes are possible by latching an index once and then writing data to different cache elements using this same address.

When writing the data cache tag portion, the SysAD bus should contain the following values:

- **SysAD[31:12]** 20 Tag Bits
- **SysAD[11]** Valid Bit
- **SysAD[10]** Dirty Bit
- **SysAD[9:0]** Unspecified

When writing instruction cache tag portion, the SysAD bus should contain the following values:

- **SysAD[31:12]** 20 Tag Bits
- **SysAD[11]** Valid Bit
- **SysAD[10:0]** Unspecified

Figures C-3, C-4, and C-5 illustrate cache write timing.

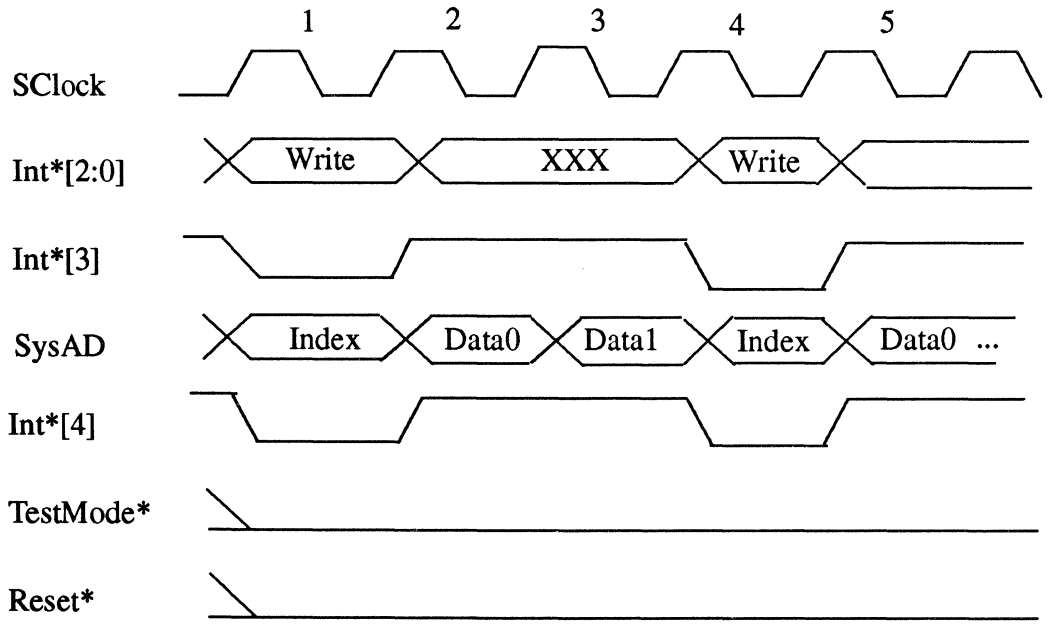


Figure C-3 Data and Instruction Cache Data Write Timing

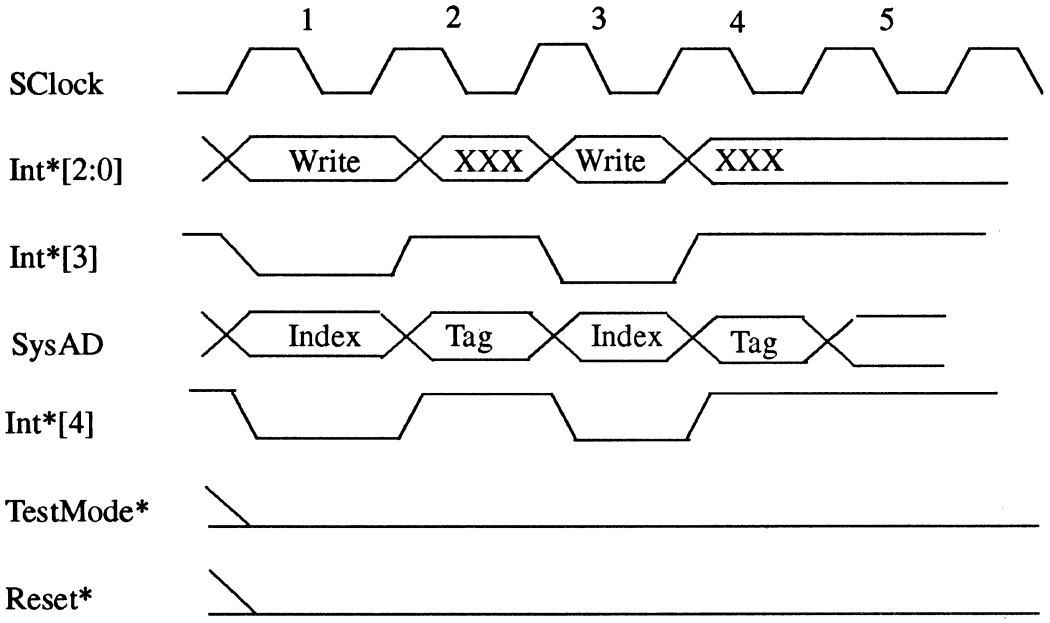


Figure C-4 Data Cache Tag Write Timing

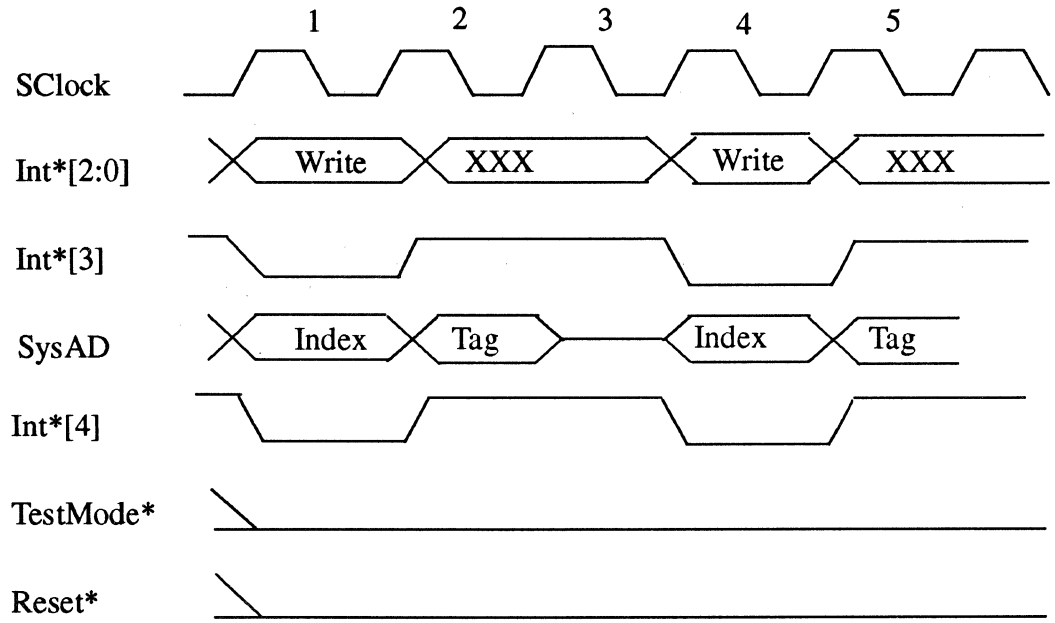


Figure C-5 Instruction Cache Tag Write Timing

Figure C-6 illustrates a back-to-back write cycle; in this case the cache address is latched and retained in the *Index* register. The first write is to a cache data bank; the second write is to the cache tag bank, with the cache address the same for both writes.

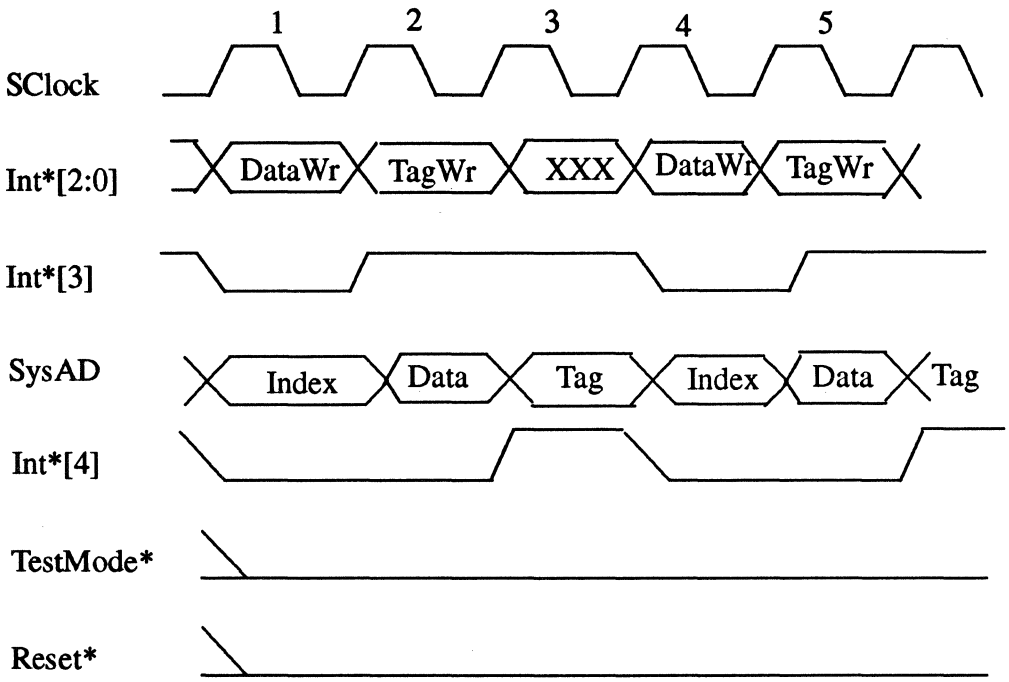


Figure C-6 Instruction and Data Cache Back-to-Back Data/Tag Write Timing

C.7 Cache Organization

Figure C-7 illustrates a topological view and the formats of the instruction and data caches. The topological view also contains address decoding scheme for each memory portion. Cache memory formats are given in Chapter 10.

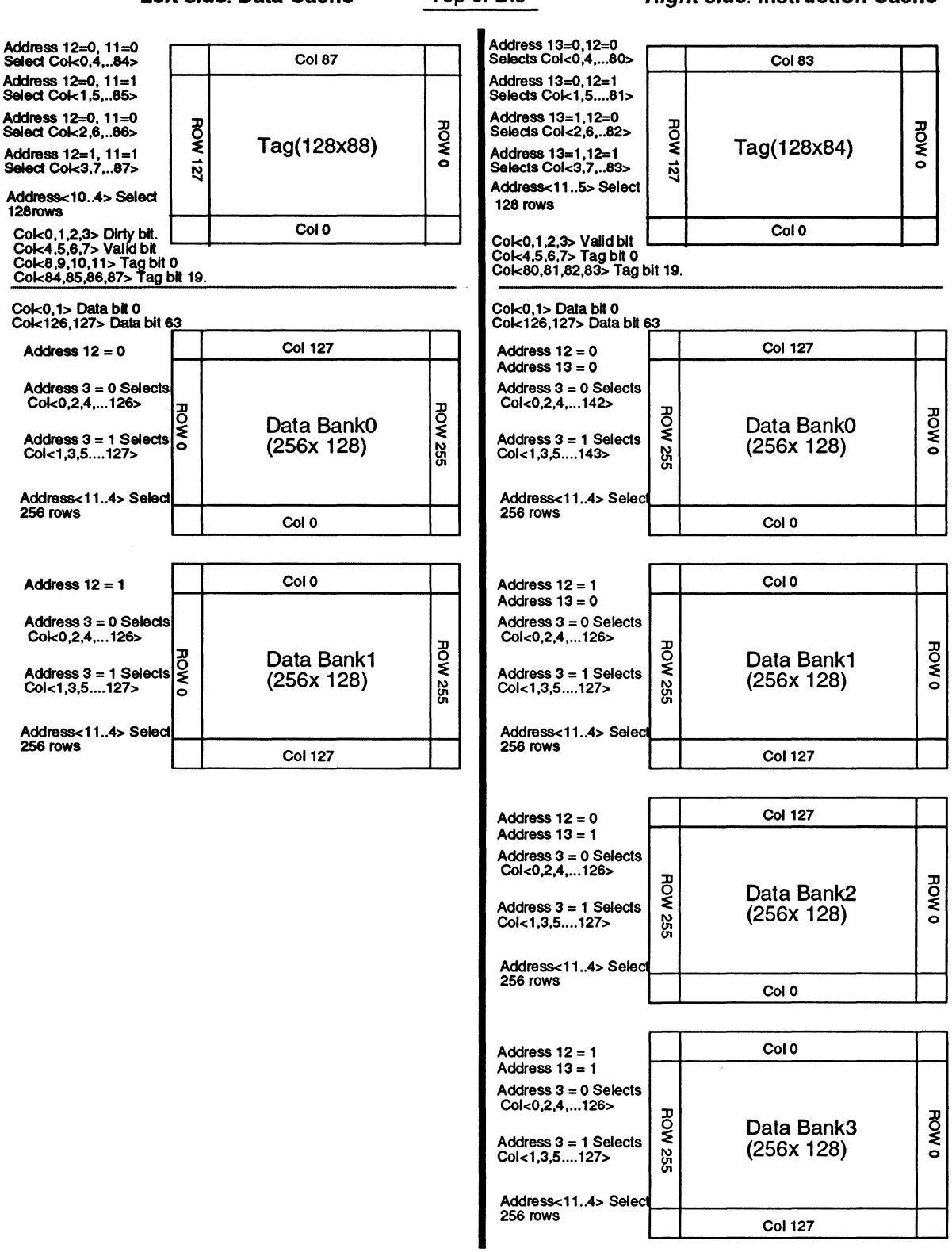


Figure C-7 Topological Cache RAM View

C.8 Testing the Dirty Bit

The memory cell for the Dirty bit is located in data cache tag portion. Since the Dirty bit is not accessible through cache test mode, executable code must be used to test this bit by executing a store instruction targeted to any location in data cache. This sets the Dirty bit, forcing a write back through Hit-Write-Back-D CACHE operation. After executing this CACHE operation, the Dirty bit is reset.

Summary of Changes

D

This chapter describes the differences between the V_R4300, V_R4200, and the V_R4000 processors, from the perspective of the V_R4300.

D.1 Differences between the V_R4300 and the V_R4000

The V_R4300 processor implements the MIPS III instruction set, as does the V_R4000. As with any two hardware implementations of the same architecture, there are some differences between V_R4300 and the V_R4000. While V_R4300 has tried to remain as close to the V_R4000 implementation as possible – both from a system design and software viewpoint – there are some visible differences. Although this section highlights the differences between V_R4300 and the V_R4000, it is not intended to be a complete list and cannot replace a thorough reading of the document.

The primary differences between V_R4300 and the V_R4000 are in the system interface bus definition and cache handling. The V_R4300 processor does not support secondary caches and must fit into a 120-pin PQFP package. Also, the V_R4300 processor does not support multiprocessing.

Differences Visible to Software

The functional differences between the V_R4000 and the V_R4300 processors that are visible to software are contained in the implementation of Coprocessor 0.

Cache Ops

Since the V_R4300 processor does not support a secondary cache, any references to a secondary cache in the V_R4000 implementation is inapplicable to the V_R4300. On the V_R4300, any SD or SI (secondary data or instruction cache) designation for a CacheOp is undefined. All write back operations send data from primary cache to main memory. CacheOp number seven (7) is undefined for V_R4300.

The data cache Dirty bit, *W* bit in V_R4000, is cleared for the CacheOp Hit Write Back Data.

The V_R4300 implements one software visible cache line state bit, whereas the V_R4000, due to multiprocessing requirements, implements two state bits. The V_R4300 processor writes **TagLo[7]** when executing the Index_Store_Tag_D instruction into the cache line state bit, while V_R4000 writes both **TagLo[7]** and **TagLo[6]**. See the section titled, Instruction and Data Caches, later on in this chapter.

Cache Parity

The V_R4300 processor does not provide parity protection for the caches.

The *CacheErr* register (CP0 register 27) is not implemented, and any accesses to this register are undefined.

The *PErr* register (CP0 register 26) is used for diagnostic purposes only.

Status Register

The V_R4300 processor *Status* register differs slightly from the V_R4000. The bits that perform identically have been left in the same location on V_R4300 as they are in the V_R4000.

The Instruction Trace Support, *ITS*, bit of the *Status* register was always zero (unused) in the V_R4000. In the V_R4300 it is now used.

The *Status* register *CH* bit, the CP0 condition bit used for the Branch on Coprocessor 0 BC0T/BC0F instructions, is writable only by software in the V_R4300. In the V_R4000, this bit could be set or cleared by the hardware, depending on secondary cache operations. The V_R4300 does not support a secondary cache.

The *Status* register's *CE* and *DE* bits, which are associated with parity handling, are unused and hardwired to 0.

Configuration Register

The V_R4300 *Config* register implements a subset of the fields on the V_R4000, since V_R4300 does not have as many options. Some of the bit definitions have been modified.

Unimplemented Operation Exception and Other Cause Bits

An Unimplemented Operation exception (*E*) in the V_R4000 FSR supersedes all other cause bits. The Unimplemented Exception handler ignores and clears all cause bits. Therefore, during an *E* exception, all other cause bits are undefined or don't cares.

The V_R4300 conforms to a stricter definition; it can set no other FP exception cause bits when there is an Unimplemented Operation exception.

Integer Divide-by-Zero

When an integer division by zero occurs, the MIPS ISA specifies the result as *undefined*. During this illegal operation, the V_R4300 implementation of the ISA returns a value in the *LO* register that is different from that of the V_R4000.

The V_R4000 returns these values in the *Hi* and *Lo* registers, for all cases of dividend:

Dividend	Lo	Hi
>= 0	0xFFFFFFFF	dividend
< 0	0x00000001	dividend

The V_R4300 returns:

Dividend	Lo	Hi
>= 0	0x7FFFFFFF	dividend
< 0	0x80000001	dividend

Cache Parity Error Exception

V_R4300 does not support parity and never takes a parity error exception.

System Design differences

This section describes the design differences between the V_R4000 and V_R4300 processors.

Processor Initialization

The V_R4000 processor has a fairly complicated scan-based processor initialization scheme. V_R4300 processor has far fewer modes: those that are not available through software are hardwired as pins on the package.

The V_R4300 requires only requires **Reset*** to be asserted for 16 cycles during the reset sequence.

System Interface

The V_R4300 uses a protocol similar but not identical to the SysAD bus protocol. The V_R4300 system interface bus is 32 bits and does not support parity. More information about the System interface is contained in Chapter 11.

The V_R4300 processor has fixed sizes for instruction cache lines (8 words) and data cache lines (4 words). 8-word instruction blocks are written to the memory system. (Writing instruction cache lines to memory can be done with the Hit Write Back CacheOp.)

The V_R4300 supports two data rates for write transactions: D and Dxx. The rate is programmed in the *EP* field of the *Config* register.

On the V_R4000, all address cycles are guaranteed to be at least three cycles apart. In fast mode on V_R4300 (when the DataRate=D) addresses for back to back writes or reads are only one cycle apart (ADAD...).

V_R4300 guarantees that it can handle data coming onto the processor as fast as it can be delivered by the SysAD bus. There will never be a stream of data presented to the V_R4300 processor exceeding eight words, and V_R4300 can handle a data pattern of DDDDDDDD.

In order to conserve power, the V_R4300 processor provides only a single TClock.

RP Bit Effect on System Interface

On the V_R4000, **SClock** and **TClock** were unaffected by the *RP* bit. On the V_R4300, setting the *RP* bit reduces the clock frequency of **SClock** and **TClock** by a factor of four. If there are any timing-dependent features in an external agent that would be affected by this (e.g., DRAM refresh counters), the external agent must provide software with a mechanism to accommodate the frequency change.

Remaining Differences

Instruction and Data Caches

The V_R4300 implements a 16 Kbyte, direct-mapped, virtually-indexed instruction cache with a cache line of eight words (thirty-two bytes). The data cache is 8 Kbytes, direct-mapped, virtually-indexed with a cache line of four words (sixteen bytes). Software routines for initializing, invalidating, or flushing the cache must take this into consideration.

The V_R4300 implements one software-visible cache line state bit. To support multiprocessing, the V_R4000 implements two cache line state bits. To maintain compatibility with the V_R4000, if an V_R4300 cache line state is checked with an Index_load_Tag_D CacheOp, the V_R4300 will repeat the single state bit and write it to *TagLo* register *PState* field. The result is that the following *PState* values are possible for V_R4300: 00, 11. In an V_R4000 processor, states of 00, 01, 10, and 11 are possible. In the V_R4000PC, which does not support multiprocessing, the state for an invalid line is 00; the state for a line filled by a cache miss is 11 (same as V_R4300).

If the line state is modified via Index_Store_Tag_D CacheOp, the only supported values for V_R4300 are 00 and 11. The V_R4300 writes **TagLo[7]** when executing Index_Store_Tag_D instruction into line state bit, while the V_R4000 writes both **TagLo[7]** and **TagLo[6]**.

TLB

The V_R4300 implements a thirty-two way, fully-associative TLB in which each entry maps two Page Frame Numbers, even and odd. While the structure of this is identical to the V_R4000, the number of entries is reduced from the forty-eight entries supported on the V_R4000.

Interactions between ITM and TLB Ops

When a software TLB instruction is accessing the JTLB during the same cycle as an Instruction TLB Miss (ITM) stall, the V_R4000 behaves in an undefined manner, and can actually generate an erroneous TLB Invalid Exception, especially when the TLB software read/write operation (TLBWI, TLBWR, TLBR) is accessing a different entry than that accessed on behalf of the Instruction TLB miss.

The V_R4300 has corrected this TLB behavior.

Floating Point Coprocessor

Pipeline Stalls

The V_R4300 integrates the operation of CP1, the floating-point coprocessor, into the main pipeline and datapath of the integer processor. This means the integer pipe stalls for any multicycle floating point instruction. While this is invisible to software in terms of functionality, code that may have been optimized for the V_R4000 will likely see no improvement on V_R4300.

Variable Latencies

Any multicycle instruction which experiences an exception with its source operand will not cause a stall. Instead, it will issue a default result on that cycle or report a Trap exception on the next cycle, depending on the state of the trap enable flags. In addition, certain trivial calculations (such as 0*0) are performed with less latency than non-trivial cases of the same operation. The V_R4000, on the other hand, always has the same latency for each particular instruction whether an exception occurs or not.

Cvt.[s,d].l instruction

To convert a 64-bit integer to a single- or double-precision floating point number, the V_R4000 initiates an FP Unimplemented Operation exception when bits 63:52 of an integer are not all ones or all zeros.

Since it has a unified 64-bit datapath for both integer and floating-point operations, V_R4300 can process more bits out of a 64-bit integer in the long integer to floating-point convert instruction, without added cost to the design. The V_R4300 raises an FP Unimplemented Operation exception only when bits 63:55 of a 64-bit integer are not all ones or all zeros.

RP Bit Effect on *PClock*

On the V_R4000, setting the *RP* bit reduces the **PClock** frequency by a factor of two through sixteen. On V_R4300, the *RP* bit reduces the **PClock** frequency by a factor of four.

Pipeline

V_R4300 uses a simpler pipeline than the V_R4000: a five-stage pipeline that strongly resembles the pipeline used on the R3000.

The V_R4300 is not superpipelined. This is not visible in any functional way, but may impact the optimization of different code sequences. This also means there will be fewer stalls due to the basic pipeline, increasing the CPI component due to cache misses. For this reason, design of the memory system is even more crucial to the performance of the V_R4300 processor.

Interrupts

The V_R4300 dedicates bit 15 of the *Cause* register, which is bit 7 of the *Interrupt Pending* field, to the timer interrupt raised when the *Compare* and *Count* registers match. The V_R4000 allowed users to select whether bit 15 was dedicated to the timer interrupt or to *Interrupt* register bit 5. *Interrupt* register bit 5 has been eliminated on V_R4300; writing to *Interrupt* register bit 5 through the system interface has no effect.

Kernel Physical Address Segment Organization

The V_R4300 implements only two cache coherency algorithms, *uncached* and *cacheable noncoherent*. This, and the fact that V_R4300 implements only 32 physical address bits as compared to the V_R4000's 36-bit physical addresses, is reflected in the organization of the Kernel Physical Address Space segment, *xkphys*, in the virtual address map. The V_R4300 breaks this segment further into two valid address spaces, one per cache coherency algorithm, while the rest of the address space is unavailable. The V_R4000 processor, in contrast, has more valid addresses in this segment. Refer to Chapter 3 for the organization of the *xkphys* virtual address segment.

JTAG

When in Shift-IR and Shift-DR modes, IEEE Std1149.1-1990 states that JTDO should be active. The V_R4300 implements the JTAG controller as indicated in IEEE Std1149.1-1990. The V_R4000 implements an older version of the standard and does not drive JTDO under these two modes.

D.2 Differences Between the V_R4300 and the V_R4200

The primary differences between the V_R4300 and the V_R4200 lay in the System interface bus architecture and in the absence of parity protection. The requirement for V_R4300 to fit into a small PQFP package made it necessary to define a new bus, similar to but not identical with the V_R4200 system interface bus.

Table D-1 Summary of Differences Between V_R4200 and V_R4300

Feature	V_R4200	V_R4300
System Bus	64 bits	32 bits
Multiplier	No	Yes
Pipeline Frequency	80 Mhz	100 Mhz
System Frequency	40 Mhz	62.5 Mhz
Flush Buffers	1 address	4 addresses
Cache Parity	Yes	No
System Bus Parity	Yes	No
Mode Pins	Yes	through *Config* register
1:1 PClock / SClock Ratio	No	Yes
1.5:1 PClock / SClock Ratio	No	Yes
2:1 PClock / SClock Ratio	Yes	Yes
3:1 PClock / SClock Ratio	No	Yes
4:1 PClock / SClock Ratio	Yes	No
PQFP Package	208 pin	120 pin
PGA Package	179 pin	179 pin (Debug Only)
Physical Address	33 bits	32 bits
RClock & MasterOut	Yes	No
Fastest DataRate	DDx	D
CacheTestMode via JTAG	No	Yes

Software Visible Differences

Cache Parity

The V_R4300 does not provide parity protection for the caches.

The *CacheErr* register (CP0 register 27) is not implemented, and any accesses to this register are undefined.

The *PErr* register (CP0 register 26) is used for diagnostic purposes only.

Status Register

The *Status* register's *CE* and *DE* bits, which are associated with parity handling, are unused and hardwired to 0 and 1 respectively.

Configuration Register

The V_R4200 *Config* register fields, *BE* and *EP*, are set by hardware to the values specified by **BigEndian** and **DataRate** pins during reset and are read only by software; the V_R4300 sets these fields to default values during ColdReset only and allows software to modify them. *Config* register bits [19:18] are changed from 00 in V_R4200 to 01 in the V_R4300.

Cache Parity Error Exception

V_R4300 does not support parity and will never take a Parity Error exception.

D.3 System Interface

The V_R4300 uses a protocol similar to the SysAD bus protocol, but not identical. The V_R4300 system interface bus is 32-bits and does not support parity. The protocol is described in Chapter 11.

In the V_R4300, 8-word instruction blocks are written to the memory system, instead of the four doublewords separated by one dead cycle used in the V_R4200.

The V_R4300 fast data rate is D instead of DDx. The data rate is software programmable through the *Config* register. It is controlled by a hardware pin on the V_R4200.

Clocks

The V_R4300 does not output **MasterOut** and **RClock.**

The clock derivation scheme is different; instead of multiplying **MasterClock** by 2 to generate **PClock**, the multiplication factor is now set by the **DivMode(1:0)** pins of the V_R4300. This factor can be 1x, 2x, 3x or 1.5x, resulting in ratios of 1:1, 2:1, 3:1 and 3:2, respectively, between **PClock** and **MasterClock.**

SClock and **TClock** are the same frequency as **MasterClock**, instead of being derived from **PClock.** As with the V_R4200, *RP* mode on the V_R4300 divides **PClock, SClock,** and **TClock** down to a fourth of their normal frequency.

Power/Gnd Pins

There are two sets of **Vcc/Vss** on the V_R4300. One set is for I/O and core, the other for the PLL.

The V_R4200 has three sets, one for I/O, one for core, and one for the PLL.

Packaging

The V_R4300 package is 120-pin PQFP, while the V_R4200 uses 208-pin PQFP.

D.4 Remaining Differences

Physical Address

The V_R4300 physical address and physical address space is 32 bits, while the V_R4200 physical address and space is 33 bits. The V_R4300 has 20 bits in the tag portion of the caches and in the *Page Frame Number* fields of TLB LO and HI entries.

Flush Buffer

The V_R4300 has a four-doubleword (64 bit)-deep flush buffer to improve the performance of back-to-back uncached write operations.

Resets

Reset* does not need to be asserted during or after assertion of **ColdReset***. **ColdReset*** does not need to be asserted/deasserted synchronously with **MasterClock**.

TLB Shutdown

When multiple entries in the TLB match during a TLB access, the TLB no longer shuts down; instead, the processor continues to operate. The *TS* bit in the *Status* register is still set, however.

V_R4300™ MIPS RISC Microprocessor

NEC
NEC Electronics Inc.
CORPORATE HEADQUARTERS
475 Ellis Street
P.O. Box 7241
Mountain View, CA 94039-7241
TEL 415-960-6000

For literature, call toll-free 7 a.m. to 6 p.m. Pacific time: **1-800-366-9782**
or FAX your request to: **1-800-729-9288**

U10391EU1V0