USER'S MANUAL

V_R4100^{TM}

64-BIT MICROPROCESSOR

(PRELIMINARY)

μPD30100

Document No. U10050EJ3V0UM00 (3rd edition)
Date Published January 1996 P
Printed in Japan

VR4000, VR4000PC, VR4100, VR4200, VR4400, and VR-Series are trademarks of NEC Corporation.
R3000, R4000, and R6000 are trademarks of MIPS Computer Systems, Inc.
MIPS is a trademark of MIPS Technologies, Inc.
MC68000 is a trademark of Motorola Inc.
IBM370 is a trademark of IBM Corp.
iAPX is a trademark of Intel Corp.
VAX is a trademark of Digital Equipment Corp.
UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd.

Exporting this product or equipment that includes this product may require a governmental license from the U.S.A. for some countries because this product utilizes technologies limited by the export control regulations of the U.S.A.

The information in this document is subject to change without notice.
No part of this document may be copied or reproduced in any form or by any means without the prior written consent of NEC Corporation. NEC Corporation assumes no responsibility for any errors which may appear in this document.
NEC Corporation does not assume any liability for infringement of patents, copyrights or other intellectual property rights of third parties by or arising from use of a device described herein or any other liability arising from use of such device. No license, either express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC Corporation or others.
While NEC Corporation has been making continuous effort to enhance the reliability of its semiconductor devices, the possibility of defects cannot be eliminated entirely. To minimize risks of damage or injury to persons or property arising from a defect in an NEC semiconductor device, customers must incorporate sufficient safety measures in its design, such as redundancy, fire-containment, and anti-failure features.
NEC devices are classified into the following three quality grades:
"Standard", "Special", and "Specific". The Specific quality grade applies only to devices developed based on a customer designated "quality assurance program" for a specific application. The recommended applications of a device depend on its quality grade, as indicated below. Customers must check the quality grade of each device before using it in a particular application.

Standard: Computers, office equipment, communications equipment, test and measurement equipment, audio and visual equipment, home electronic appliances, machine tools, personal electronic equipment and industrial robots

Special: Transportation equipment (automobiles, trains, ships, etc.), traffic control systems, anti-disaster systems, anti-crime systems, safety equipment and medical equipment (not specifically designed for life support)

Specific: Aircrafts, aerospace equipment, submersible repeaters, nuclear reactor control systems, life support systems or medical equipment for life support, etc.

The quality grade of NEC devices is "Standard" unless otherwise specified in NEC's Data Sheets or Data Books. If customers intend to use NEC devices for applications other than those specified for Standard quality grade, they should contact an NEC sales representative in advance.
Anti-radioactive design is not implemented in this product.

NOTES FOR CMOS DEVICES

① PRECAUTION AGAINST ESD FOR SEMICONDUCTORS

Note:

Strong electric field, when exposed to a MOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop generation of static electricity as much as possible, and quickly dissipate it once, when it has occurred. Environmental control must be adequate. When it is dry, humidifier should be used. It is recommended to avoid using insulators that easily build static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work bench and floor should be grounded. The operator should be grounded using wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions need to be taken for PW boards with semiconductor devices on it.

② HANDLING OF UNUSED INPUT PINS FOR CMOS

Note:

No connection for CMOS device inputs can be cause of malfunction. If no connection is provided to the input pins, it is possible that an internal input level may be generated due to noise, etc., hence causing malfunction. CMOS devices behave differently than Bipolar or NMOS devices. Input levels of CMOS devices must be fixed high or low by using a pull-up or pull-down circuitry. Each unused pin should be connected to VDD or GND with a resistor, if it is considered to have a possibility of being an output pin. All handling related to the unused pins must be judged device by device and related specifications governing the devices.

③ STATUS BEFORE INITIALIZATION OF MOS DEVICES

Note:

Power-on does not necessarily define initial status of MOS device. Production process of MOS does not define the initial operation status of the device. Immediately after the power source is turned ON, the devices with reset function have not yet been initialized. Hence, power-on does not guarantee out-pin levels, I/O settings or contents of registers. Device is not initialized until the reset signal is received. Reset operation must be executed immediately after power-on for devices having reset function.

PREFACE

Readers

This manual targets users who intends to understand the functions of the VR4100 and to design application systems using this microprocessor.

Purpose

This manual introduces the architecture and hardware functions of the VR4100 to users, following the organization described below.

Organization

This manual consists of the following contents:

- Introduction
- Pipeline operation
- Memory management system
- Exception processing
- Hardware
- Instruction set details

How to read this manual

It is assumed that the reader of this manual has general knowledge in the fields of electric engineering, logic circuits, and microcomputers.

The R4000^{TM} in this manual represents the VR4000^{TM} and the VR4400^{TM} except for **Appendix B Difference between VR4100 and Other VR-Series Processors.**

To learn about detailed function of a specific instruction,

-> Refer to **Chapter 2 CPU Instruction Set Summary** and **Chapter 14 CPU Instruction Set Details**.

To learn about the overall functions of the VR4100,

-> Read this manual in sequential order.

To learn about electrical specifications,

-> Refer to **Chapter 13 Electrical Characteristics**.

Legend

Data significance	: Higher on left and lower on right
Active low	: $\overline{\text{xxx}}$ (bar over pin and signal names)
Numeric representation	: binary ... xxxx or xxxx_2
	: decimal ... xxxx
	: hexadecimal ... 0xxxxx
Revised point	: star on the margin

[MEMO]

Introduction

1

The VR4100 microprocessor is a low-cost, low-power microprocessor that is compatible with the MIPS^{TM} I, MIPS II, and MIPS III Instruction Set Architecture (ISA), except for the Floating-point operating instructions, LL/LLD instruction and SC/SCD instruction.

The chip does not provide on-chip support for a secondary cache or multiprocessing, and Floating-Point operation.

1.1 VR4100 Processor Characteristics

The VR4100 processor has the following characteristics:

- 64-bit processing
- 33-MHz internal clock, derived from an external 8.25-MHz clock
- Optimized 5-stage pipeline, 2 Kbyte I-cache and 1 Kbyte D-cache size, and 32-double-entry TLB size
- 40-bit virtual address space, 32-bit physical address space
- Low-voltage operation (3.3 volt)
- Fully static circuit
- Write-back cache
- Fast Multiply-and-Accumulate unit
- Power management features, which include the following four operating modes
    - Fully Speed mode
    - ★ Standby mode
    - Suspend mode
    - Hibernate mode
- 100-pin thin plastic quad flat pack (TQFP)
- No Floating Point Unit
- No Secondary cache or multiprocessing

1.2 VR4100 Processor Implementation

This section describes the following:

- the 64-bit architecture of the VR4100 processor
- the CPU instruction pipeline (described in detail in Chapter 3)
- an overview of the System interface (described in detail in Chapter 11)
- an overview of the CPU registers (detailed in Chapters 4 and 5) and CPU instruction set (detailed in Chapter 2 and Chapter 14)
- data formats and byte ordering
- 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 VR4100 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 VR4100 processor.

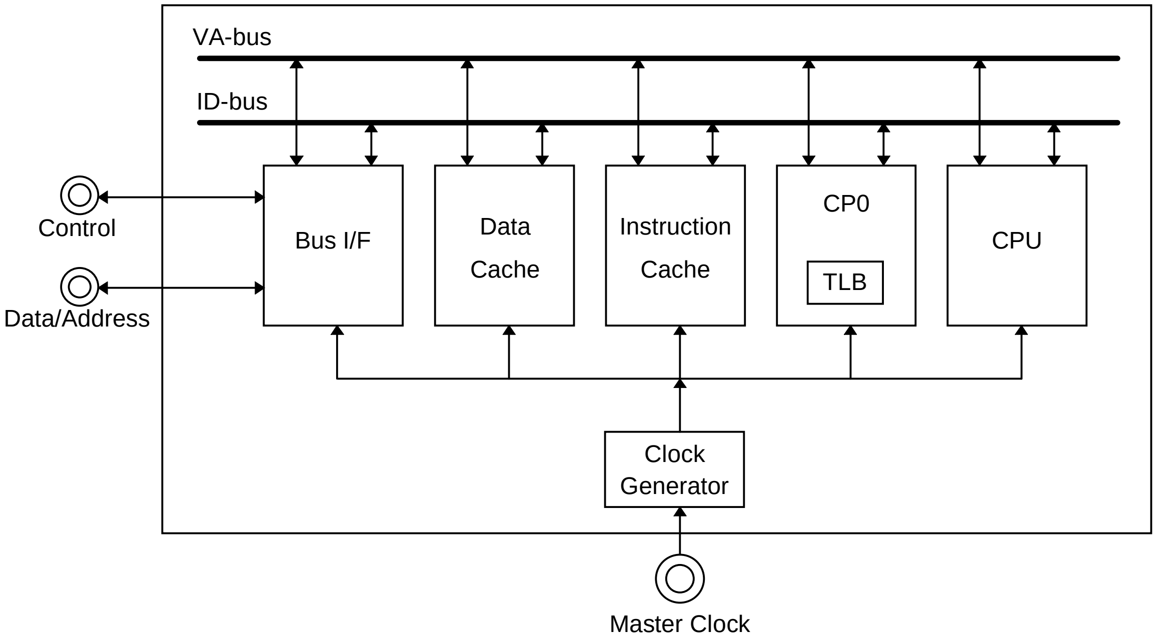


Figure 1-1 VR4100 Processor Internal Block Diagram

CPU has the hardware resources to execute integer and instructions. It has a 64-bit register file, 64-bit integer datapath, and Multiply-and-Accumulator.

Coprocessor 0 (CP0) has the memory management unit (MMU) and handles exception processing. The MMU handles address translation and checks memory accesses that occur between different memory segments (user, supervisor, or kernel). The translation lookaside buffer (TLB) is used to translate virtual to physical addresses.

Instruction Cache is direct-mapped, virtually-indexed, and physically-tagged.

Data Cache is a direct-mapped, virtually-indexed and physically-tagged write-back cache.

Bus Interface allows the processor to access external resources. It contains a 32-bit multiplexed address/data bus, with per-byte parity, clock signals, interrupts, and various control signals.

★ **Clock Generator** quadruples the input clock (labeled **MasterClock**) frequency to produce the pipeline clock. This clock is then divided to produce the System interface clock, according to $\overline{\text{Div2}}$ pin. The processor uses a phase-locked loop (PLL) to generate the pipeline clock, that operates at quadruple the frequency of the **MasterClock**.

VR4100 Instruction Pipeline

The VR4100 processor has a 5-stage instruction pipeline. Under normal circumstances, one instruction is issued each cycle.

The instruction pipeline of the VR4100 processor operates at quadruple the frequency of the **MasterClock**. 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, *GPRs*

In addition, the processor provides the following special registers:

- 64-bit Program Counter, the *PC* register
- 64-bit *HI* register, containing the integer multiply, divide and multiply-and-add upper doubleword result
- 64-bit *LO* register, containing the integer multiply, divide and multiply-and-add lower doubleword result

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 VR4100 processor mode of operation.

Figure 1-2 shows the VR4100 processor registers.

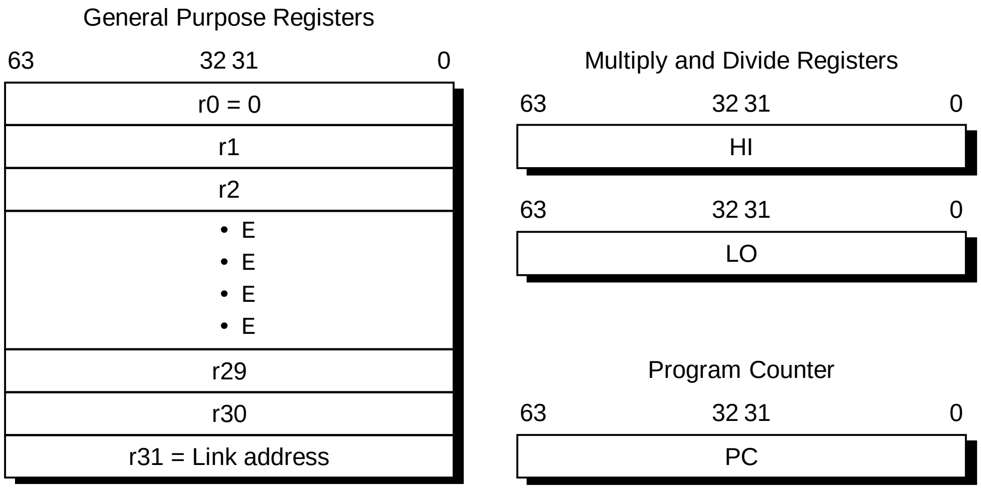


Figure 1-2   VR4100 Processor Registers

The VR4100 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). CP0 registers are described later in this chapter.

CPU Instruction Set Overview

Each CPU instruction is 32 bits long. As shown in Figure 1-3, there are three instruction formats:

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

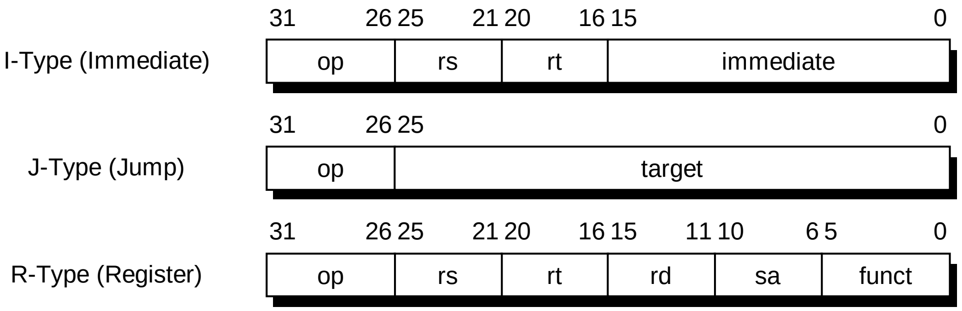


Figure 1-3 CPU Instruction Formats

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

Instruction decoding is greatly simplified by limiting the number of formats to these three. This limitation means that the more complicated (and less frequently used) operations and addressing modes can be synthesized by the compiler, using sequences of these same simple instructions.

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

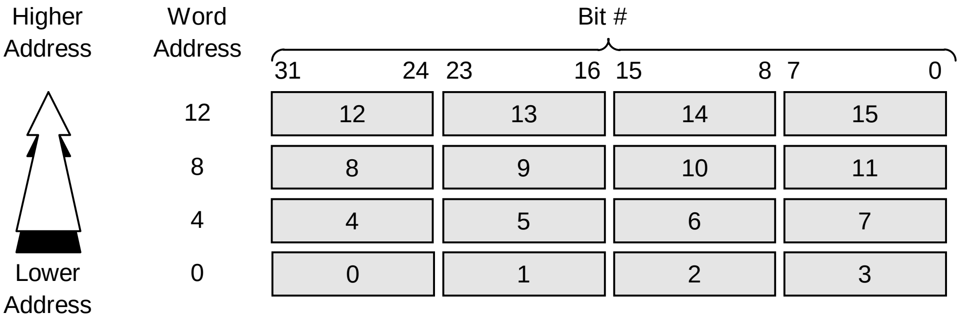
- **Load and Store** instructions move data between memory and general registers. They are all immediate (I-type) instructions, since the only addressing mode supported is base register plus 16-bit, signed immediate offset.
- **Computational** instructions perform arithmetic, logical, shift, multiply,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

Chapter 2 provides a more detailed summary and Chapter 14 gives a complete description of each instruction.

Data Formats and Addressing

The VR4100 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 a dedicated pin, **BigEndian**. Endianness refers to the location of byte 0 within the multi-byte data structure. Figures Figure 1-4 and Figure 1-5 show the ordering of bytes within words and the ordering of words within multiple-word structures for the big-endian and little-endian conventions.

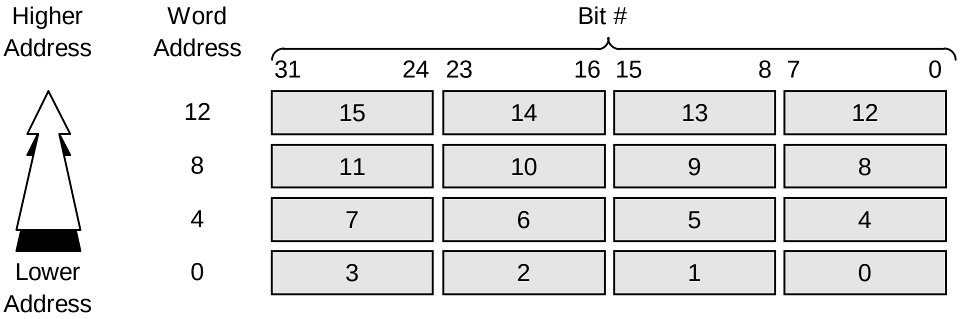
When the VR4100 processor is configured as a big-endian system, byte 0 is the most-significant (leftmost) byte, thereby providing compatibility with MC68000$^{\text{TM}}$ and IBM 370$^{\text{TM}}$ conventions. Figure 1-4 shows this configuration.



Word Address	31 24	23 16	15 8	7 0
12	12	13	14	15
8	8	9	10	11
4	4	5	6	7
0	0	1	2	3

Figure 1-4 Big-Endian Byte Ordering

When configured as a little-endian system, byte 0 is always the least-significant (rightmost) byte, which is compatible with iAPX$^{\text{TM}}$ x86 and DEC VAX$^{\text{TM}}$ conventions. Figure 1-5 shows this configuration.



Word Address	31 24	23 16	15 8	7 0
12	15	14	13	12
8	11	10	9	8
4	7	6	5	4
0	3	2	1	0

Figure 1-5 Little-Endian Byte Ordering

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

Figures Figure 1-6 and Figure 1-7 show little-endian and big-endian byte ordering in doublewords.

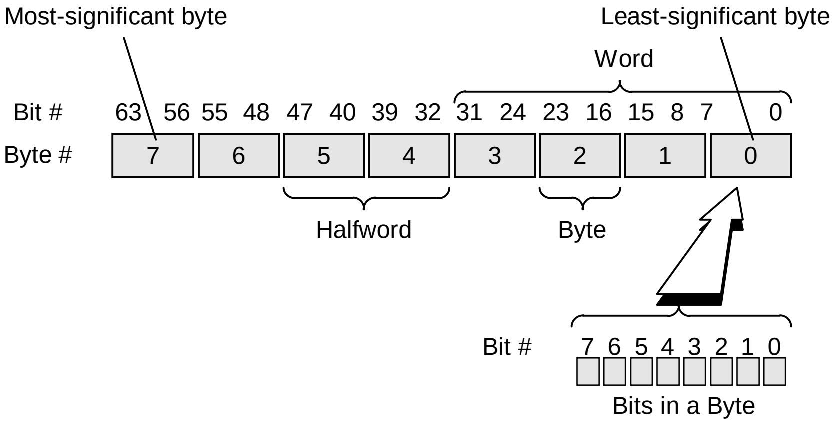


Figure 1-6 Little-Endian Data in a Doubleword

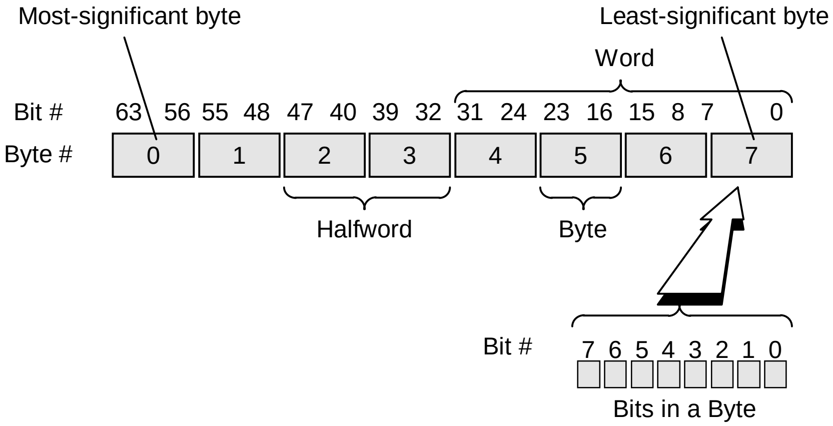


Figure 1-7 Big-Endian Data in a Doubleword

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

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

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

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

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

Figures Figure 1-8 and Figure 1-9 show the access of a misaligned word that has byte address 3

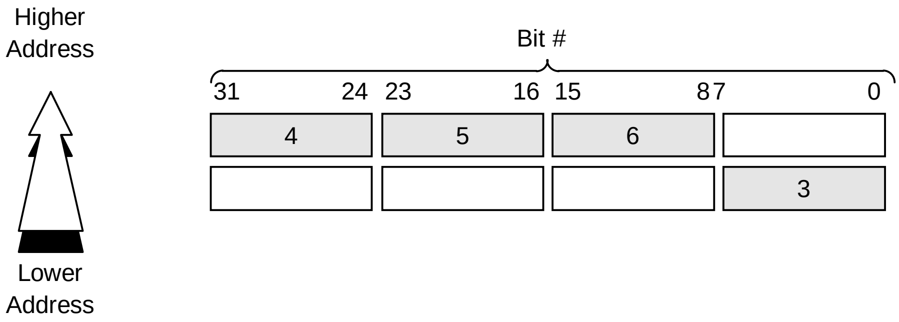


Figure 1-8 Big-Endian Misaligned Word Addressing

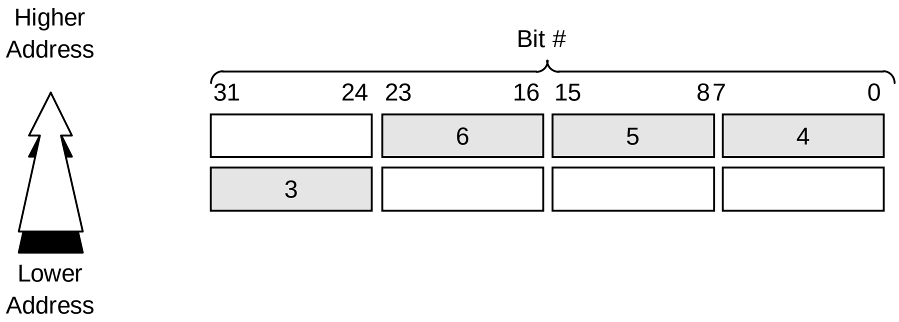


Figure 1-9 Little-Endian Misaligned Word Addressing

Coprocessors (CP0-CP3)

The MIPS ISA defines four coprocessors (designated CP0 through CP3):

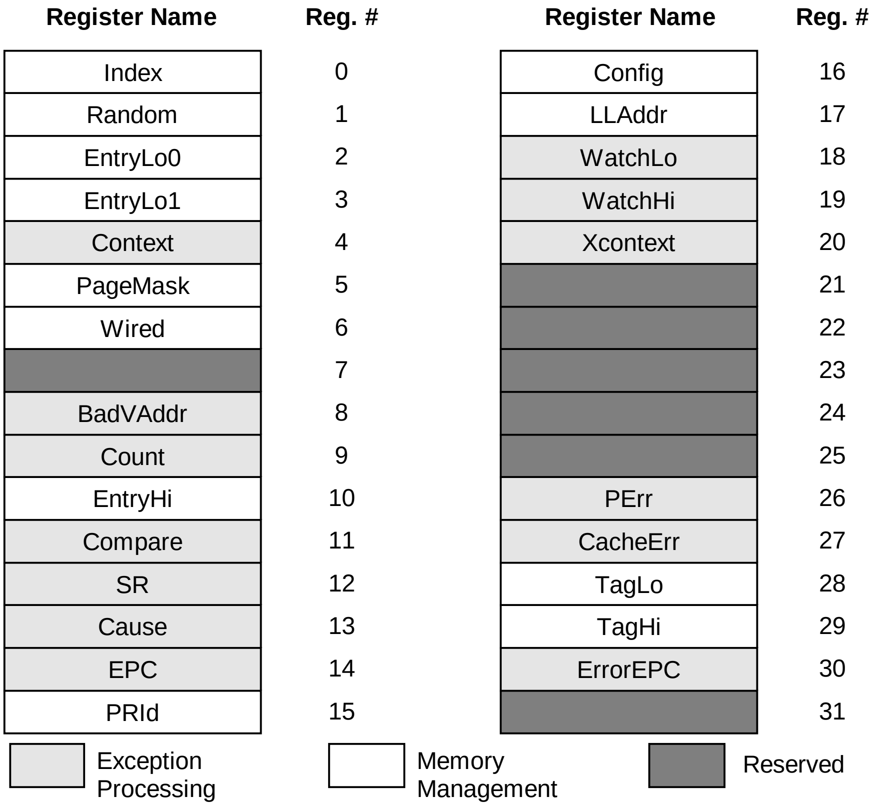
- Coprocessor 0 (**CP0**) is incorporated on the CPU chip and supports the Figure 1-9 virtual memory system and exception handling. CP0 is also referred to as the *System Control Coprocessor*.
- Coprocessor 1 (**CP1**) is reserved for floating-point coprocessor operations.
- Coprocessor 2 (**CP2**) is reserved for future definition by MIPS.
- Coprocessor 3 (**CP3**) is no longer defined. CP3 opcodes are reserved for future extensions.

CP0 is 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 shown in Figure 1-10 and described in Table 1-1 manipulate the memory-management and exception-handling capabilities of the CPU.



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 1-10 CP0 Registers

Number	Register	Description
0	Index	Programmable pointer into TLB array
1	Random	Pseudorandom pointer into TLB aray (read only)
2	EntryLo0	Load half of TLB entry for even virtual address (VPN)
3	EntryLo1	Load 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	Reserved
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	Cache parity bits
27	CacheErr	Cache Error and Status register
28	TagLo	Cache Tag register
29	TagHi	Cache Tag register
30	ErrorEPC	Error Exception Program Counter
31	-	Reserved

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

Floating-Point Operation

The VR4100 inimplementes the Floating-point Unit (FPU) instruction set. Coprocessor unusable exception will occur whenever any FPU instructions are excecuted. If needs, Floating-point instructions should be emulated by software in exception handler.

Memory Management System (MMU)

The VR4100 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 VR4100 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 4.

Joint TLB

For fast virtual-to-physical address decoding, the VR4100 uses a large, fully associative TLB which 64 Virtual pages to their corresponding physical addresses. The TLB is organized as 32 pairs of even-odd entries, and maps a virtual address and address space identifier into the 4-GB physical address space.

The page size can be configured, on a per-entry basis, to map a page size of 1KB to 256KB (in multiples of 4). A CP0 register is loaded with the page size of a mapping, and that size is entered into the TLB when a new entry is written. Thus, operating systems can provide special purpose maps; for example, a typical frame buffer can be memory mapped using only one TLB entry.

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.

Operating Modes

The VR4100 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 4.

Instruction Cache

The VR4100 incorporates a direct-mapped on-chip instruction cache. This virtually indexed, physically tagged cache is 2KB in size and is protected with word parity.

Because the cache is virtually indexed, the virtual-to-physical address translation occurs in parallel with the cache access, thus further simultaneously. The tag holds a 22-bit physical address and valid bit, and is parity protected.

The instruction cache is 64-bits wide, and can be refilled or accessed in a single processor cycle. Instruction fetches require only 32 bits per cycle, for a peak instruction bandwidth of 132MB/sec. The line size is four instructions (16 bytes).

Data Cache

For fast, single cycle data access, the VR4100 includes a 1KB on-chip data cache that is direct-mapped with a fixed 16-byte (four words) line size.

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

The write policy is writeback, which means that a store to a cache line does not immediately cause memory to be updated. This increases system performance by reducing bus traffic and eliminating the bottleneck of waiting for each store operation to finish before issuing a subsequent memory operation.

Associated with the Data Cache is the store buffer. When the VR4100 executes a Store instruction, this single-entry buffer gets written with the store data while the tag comparison is performed. If the tag matches, then the data is written into the Data Cache in the next cycle that the Data Caches not accessed (the next non-load cycle). The store buffer allows the VR4100 to execute a store ever processor cycle and to perform back-to-back stores without penalty.

Bus Interface

Bus Interface allows the processor access to external resources required to satisfy internal requirements. It contains a 32-bit wide, multiplexed address and data bus, clock signals, interrupts, and a number of control signals.

Clock Generator

The VR4100 has 5 clocks that the user must be aware of. First, there is the pipeline clock, PClock. This clock is used for the pipeline and pipeline related functions internal to the VR4100. It is four times the MasterClock frequency. The next clock is MasterOut. This clock output is aligned with MasterClock. And the next clock is the system interface clock, SClock. This is also an internal clock and is used to sample data at the system interface and to clock data into the processor system interface output registers. The SClock is a divided version of the PClock. The divisor is selected by ★ $\overline{\text{Div2}}$ pin.

There is the TClock, Transmit clock. The TClock is used to clock the output registers (signals transmitted to the VR4100) of the external agent and is at the same frequency as SClock.

[MEMO]

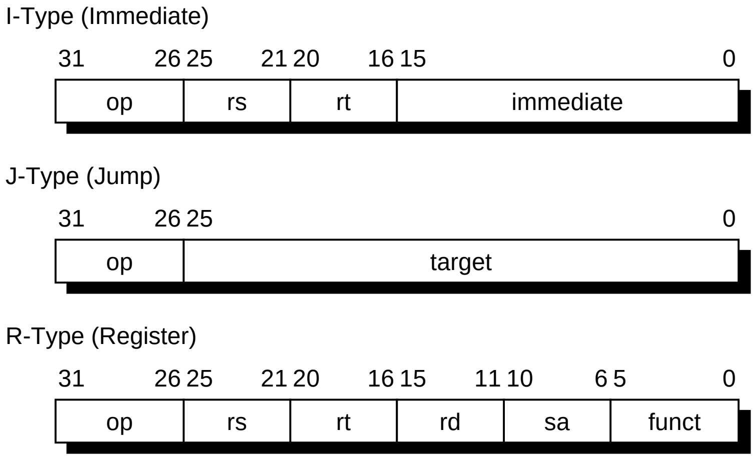
CPU Instruction Set Summary

2

This chapter is an overview of the central processing unit (CPU) instruction set; refer to Chapter 14 for detailed descriptions of individual CPU instructions.

2.1 CPU Instruction Formats

Each CPU instruction consists of a single 32-bit word, aligned on a word boundary. There are three instruction formats -- immediate (I-type), jump (J-type), and register (R-type) -- as shown in Figure 2-1. The use of a small number of instruction formats simplifies instruction decoding, allowing the compiler to synthesize more complicated (and less frequently used) operations and addressing modes from these three formats as needed.



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

Figure 2-1 CPU Instruction Formats

In the MIPS architecture, coprocessor instructions are implementation-dependent; see Chapter 14 for details of individual Coprocessor 0 instructions.

Support of the MIPS ISA

The VR4100 processor does not support a multiprocessor operating environment, and the synchronization support instructions defined in the MIPS II and MIPS III ISA -- the Load Linked and Store Conditional instructions -- cause reserved instruction exception. The load link bit (*LLbit*) is eliminated.

Note that the SYNC instruction is handled as a NOP since all load/store instructions in this processor are executed in program order.

Load and Store Instructions

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

Scheduling a Load Delay Slot

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

In the VR4100 processor, the instruction immediately following a load instruction can use the contents of the loaded register, however in such cases hardware interlocks insert additional real cycles. Consequently, scheduling load delay slots can be desirable, both for performance and VR-Series^{TM} processor compatibility. However, the scheduling of load delay slots is not absolutely required in the VR4100.

Instruction	PCycles Required
Load	1

Table 2-1 Load and Store Instruction Cycle Timing

Defining Access Types

Access type indicates the size of an VR4100 processor data item to be loaded or stored, set by the load or store instruction opcode. Access types are defined in Chapter 14.

Regardless of access type or byte ordering (endianness), the address given specifies the low-order byte in the addressed field.

- For a big-endian configuration, the low-order byte is the most-significant byte.
- For a little-endian configuration, the low-order byte is the least-significant byte.

The access type, together with the three low-order bits of the address, define the bytes accessed within the addressed doubleword (shown in Table 2-2). Only the combinations shown in Table 2-2 are permissible; other combinations cause address error exceptions.

See Chapter 14 for individual descriptions of CPU load and store instructions.

Access Type Mnemonic (Value)	Low Address Bits			Bytes Accessed															
				Big endian (63.....31.....0) Byte								Little endian (63.....31.....0) Byte							
Double word (7)	0	0	0	0	1	2	3	4	5	6	7	7	6	5	4	3	2	1	0
Septibyte (6)	0	0	0	0	1	2	3	4	5	6			6	5	4	3	2	1	0
	0	0	1		1	2	3	4	5	6	7	7	6	5	4	3	2	1	
Sextibyte (5)	0	0	0	0	1	2	3	4	5					5	4	3	2	1	0
	0	1	0			2	3	4	5	6	7	7	6	5	4	3	2		
Quintibyte (4)	0	0	0	0	1	2	3	4							4	3	2	1	0
	0	1	1				3	4	5	6	7	7	6	5	4	3			
Word (3)	0	0	0	0	1	2	3									3	2	1	0
	1	0	0					4	5	6	7	7	6	5	4				
TripleByte (2)	0	0	0	0	1	2											2	1	0
	0	0	1		1	2	3									3	2	1	
	1	0	0					4	5	6			6	5	4				
	1	0	1						5	6	7	7	6	5					
Halfword (1)	0	0	0	0	1													1	0
	0	1	0			2	3									3	2		
	1	0	0					4	5					5	4				
	1	1	0							6	7	7	6						
Byte (0)	0	0	0	0															0
	0	0	1		1													1	
	0	1	0			2											2		
	0	1	1				3									3			
	1	0	0					4							4				
	1	0	1						5					5					
	1	1	0							6			6						
	1	1	1								7	7							

Table 2-2 Byte Access within a Doubleword

Computational Instructions

Computational instructions can be either in register (R-type) format, in which both operands are registers, or in immediate (I-type) format, in which one operand is a 16-bit immediate.

Computational instructions perform the following operations on register values:

- arithmetic
- logical
- shift
- multiply
- divide

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

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

64-bit Operations

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

Cycle Timing for Multiply and Divide Instructions

MFHI and MFLO Instructions(described in Chapter 14) are interlocked so that any attempt to read them before prior instructions complete delays the execution of these instructions until the prior instructions finish.

Table 2-3 gives the number of processor cycles(PCycles) required to resolve the interlock or stall between various multiply or divide instructions and a subsequent MFHI or MFLO instruction.

Instruction	PCycles Required
MULT	1
MULTU	1
DIV	35
DIVU	35
DMULT	4
DMULTU	4
DDIV	67
DDIVU	67
MADD16	1
DMADD16	1

Table 2-3   Multiply/Divide Instruction Cycle Timing

For more information about computational instructions, refer to the individual instruction as described in Chapter 14.

Jump and Branch Instructions

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

Instruction	PCycles Required
Branch	1
Jump	1

Table 2-4   Jump and Branch Instruction Cycle Timing

Overview of Jump Instructions

Subroutine calls in high-level languages are usually implemented with Jump or Jump and Link instructions, both of which are J-type instructions. In J-type format, the 26-bit target address shifts left 2 bits and combines with the high-order 4 bits of the current program counter to form an absolute address.

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

For more information about jump instructions, refer to the individual instruction as described in Chapter 14.

[†] The target instruction of a taken Branch is fetched in the EX stage of the Branch instruction. Branch comparison and target address calculation are done in phase 2 of RF of the Branch instruction. The architecturally-defined branch delay slot of one cycle is still required. A Jump instruction also requires one delay slot. Branch likely instructions which are not taken kill the delay slot instruction.

Overview of Branch Instructions

All branch instruction target addresses are computed by adding the address of the instruction in the delay slot to the 16-bit *offset* (shifted left 2 bits and sign-extended to 64 bits). All branches occur with a delay of one instruction.

If a branch likely instruction is not taken, the instruction in its delay slot is nullified. For all other branch instructions, the instruction in its delay slot is unconditionally executed.

For more information about branch instructions, refer to the individual instruction as described in Chapter 14.

Special Instructions

Special instructions allow the software to initiate traps; they are always R-type. For more information about special instructions, refer to the individual instruction as described in Chapter 14.

Coprocessor 0 (CP0) Instructions

CP0 instructions perform operations specifically on the System Control Coprocessor registers to manipulate the memory management and exception handling facilities of the processor. Chapter 14 details CP0 instructions.

List of CPU Instructions

Tables 2-5 through 2-19 list CPU instructions common to MIPS R-Series processors, along with those instructions that are extensions to the instruction set architecture. The extensions result in code space reductions, multiprocessor support, and improved performance in operating system kernel code sequences -- for instance, in situations where run-time bounds-checking is frequently performed. Table 2-18 lists CP0 instructions.

OpCode	Description
LB	Load Byte
LBU	Load Byte Unsigned
LH	Load Halfword
LHU	Load Halfword Unsigned
LW	Load Word
LWL	Load Word Left
LWR	Load Word Right
SB	Store Byte
SH	Store Halfword
SW	Store Word
SWL	Store Word Left
SWR	Store Word Right

Table 2-5 CPU Instruction Set: Load and Store Instructions

OpCode	Description
ADDI	Add Immediate
ADDIU	Add Immediate Unsigned
SLTI	Set on Less Than Immediate
SLTIU	Set on Less Than Immediate Unsigned
ANDI	AND Immediate
ORI	OR immediate
XORI	Exclusive OR Immediate
LUI	Load Upper Immediate

Table 2-6 CPU Instruction Set: Computational (ALU Immediate) Instructions

OpCoce	Description
ADD	Add
ADDU	Add Unsigned
SUB	Subtract
SUBU	Subtract Unsigned
SLT	Set on Less Than
SLTU	Set on Less Than Unsigned
AND	AND
OR	OR
XOR	Exclusive OR
NOR	NOR

Table 2-7   CPU Instruction Set: Computational (3-Operand, R-Type) and phase and phase 1 of EX 1 of EX

OpCode	Description
MULT	Multiply
MULTU	Multiply Unsigned
DIV	Divide
DIVU	Divide Unsigned
MFHI	Move From HI
MTHI	Move To HI
MFLO	Move From LO
MTLO	Move To LO

Table 2-8   CPU Instruction Set: Computational (Multiply and Divide)

OpCode	Description
J	Jump
JAL	Jump And Link
JR	Jump Register
JALR	Jump And Link Register
BEQ	Branch on Equal
BNE	Branch on Not Equal
BLEZ	Branch on Less Than or Equal to Zero
BGTZ	Branch on Greater Than Zero
BLTZ	Branch on Less Than Zero
BGEZ	Branch on Greater Than or Equal to Zero
BLTZAL	Branch on Less Than Zero And Link
BGEZAL	Branch on Greater Than or Equal to Zero And Link
BC0T	Branch on Coprocessor 0 True
BC0F	Branch on Coprocessor 0 False

Table 2-9   CPU Instruction Set: Jump and Branch Instructions

OpCode	Description
BEQL	Branch on Equal Likely
BNEL	Branch on Not Equal Likely
BLEZL	Branch on Less Than or Equal to Zero Likely
BGTZL	Branch on Greater Than Zero Likely
BLTZL	Branch on Less Than Zero Likely
BGEZL	Branch on Greater Than or Equal to Zero Likely
BLTZALL	Branch on Less Than Zero And Link Likely
BGEZALL	Branch on Greater Than or Equal to Zero And Link Likely
BC0TL	Branch on Coprocessor 0 True Likely
BC0FL	Branch on Coprocessor 0 False Likely

Table 2-10   CPU Instruction Set: Branch Likely Instructions

OpCode	Description
SLL	Shift Left Logical
SRL	Shift Right Logical
SRA	Shift Right Arithmetic
SLLV	Shift Left Logical Variable
SRLV	Shift Right Logical Variable
SRAV	Shift Right Arithmetic Variable

Table 2-11   CPU Instruction Set: Shift Instructions Move To HI

OpCode	Description
SYNC	Synchronize memory references
SYSCALL	System Call
BREAK	Break
TGE	Trap if Greater Than or Equal
TGEU	Trap if Greater Than or Equal Unsigned
TLT	Trap if Less Than
TLTU	Trap if Less Than Unsigned
TEQ	Trap if Equal
TNE	Trap if Not Equal
TGEI	Trap if Greater Than or Equal Immediate
TGEIU	Trap if Greater Than or Equal Immediate Unsigned
TLTI	Trap if Less Than Immediate
TLTIU	Trap if Less Than Immediate Unsigned
TEI	Trap if Equal Immediate
TNEI	Trap if Not Equal Immediate

Table 2-12   CPU Instruction Set: Special Instructions

OpCode	Description
LD	Load Doubleword
LDL	Load Doubleword Left
LDR	Load Doubleword Right
LWU	Load Word Unsigned
SD	Store Doubleword
SDL	Store Doubleword Left
SDR	Store Doubleword Right

Table 2-13   MIPS III Extensions: Load and Store Instructions

OpCode	Description
DADDI	Doubleword Add Immediate
DADDIU	Doubleword Add Immediate Unsigned

Table 2-14   MIPS III Extensions: Computational (ALU Immediate)

OpCode	Description
DADD	Doubleword Add
DADDU	Doubleword Add Unsigned
DSUB	Doubleword Subtract
DSUBU	Doubleword Subtract Unsigned

Table 2-15   MIPS III Extensions: Computational (3-operand, R-type)

OpCode	Description
DMULT	Doubleword Multiply
DMULTU	Doubleword Multiply Unsigned
DDIV	Doubleword Divide
DDIVU	Doubleword Divide Unsigned

Table 2-16 MIPS III Extensions: Computational (Multiply and Divide)

OpCode	Description
DSLL	Doubleword Shift Left Logical
DSRL	Doubleword Shift Right Logical
DSRA	Doubleword Shift Right Arithmetic
DSLLV	Doubleword Shift Left Logical Variable
DSRLV	Doubleword Shift Right Logical Variable
DSRAV	Doubleword Shift Right Arithmetic Variable
DSLL32	Doubleword Shift Left Logical + 32
DSRL32	Doubleword Shift Right Logical + 32
DSRA32	Doubleword Shift Right Arithmetic + 32

Table 2-17 MIPS III Extensions: Shift Instructions

OpCode	Description
DMFC0	Doubleword Move From CP0
DMTC0	Doubleword Move To CP0
MTC0	Move to CP0
MFC0	Move from CP0
TLBR	Read Indexed TLB Entry
TLBWI	Write Indexed TLB Entry
TLBWR	Write Random TLB Entry
TLBP	Probe TLB for Matching Entry
ERET	Exception Return
CACHE	Cache Operation
HIBERNATE	Hibernate
SUSPEND	Suspend
STANDBY	Standby

Table 2-18   CP0 Instructions

OpCode	Description
MADD16	Multiply and Add 16 bits
DMADD16	Doubleword Multiply and Add 16 bits

Table 2-19   VR4100 Extension Instructions

The VR4100 Processor Pipeline

3

This chapter describes the basic operation of the VR4100 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 VR4100 implementation of a write buffer.

3.1 Pipeline Stages

The CPU has a five-stage instruction pipeline; each stage takes one PCycle (one cycle of PClock, which runs at four times the frequency of MasterClock), and each PCycle has two phases: Φ1 and Φ2, as shown in Figure 3-1. Thus, the execution of each instruction takes at least 5 PCycles. An instruction can take longer -- for example, if the required data is not in the cache, the data must be retrieved from main memory.

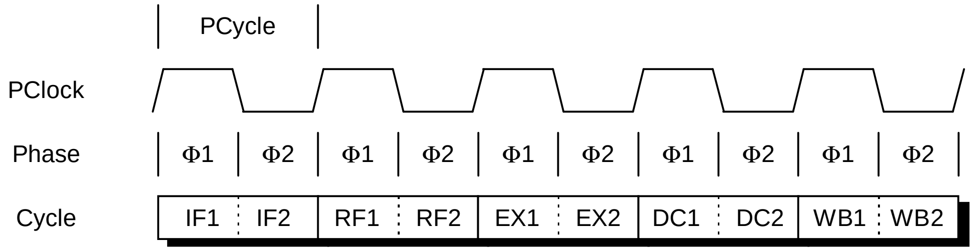


Figure 3-1 Pipeline Stages

The five pipeline stages are:

- IF - 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 3-2 shows the five stages of the instruction pipeline; the next section describes the pipeline stages.

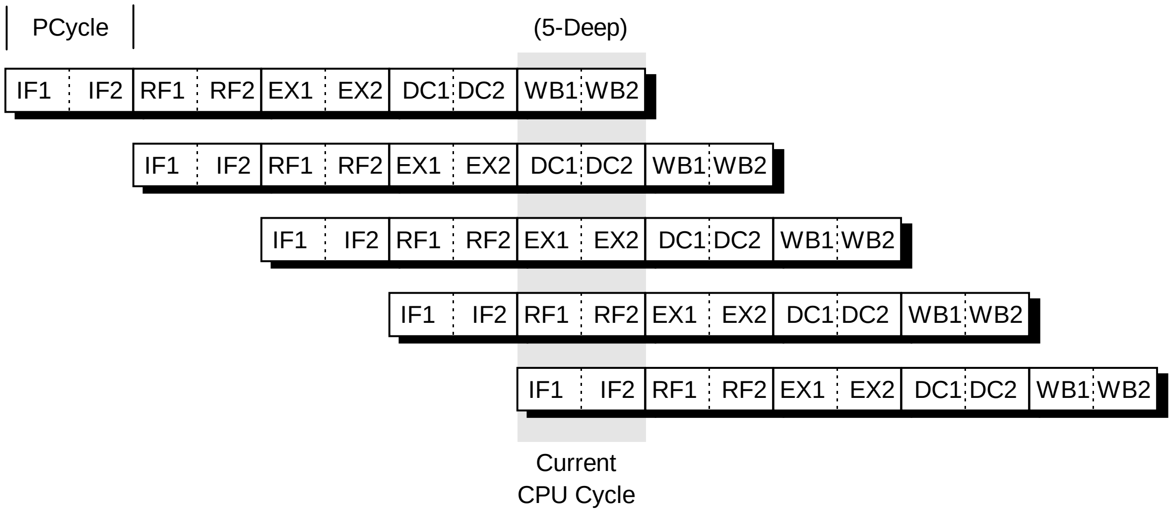


Figure 3-2 Instruction Execution in the Pipeline

Pipeline Activities

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

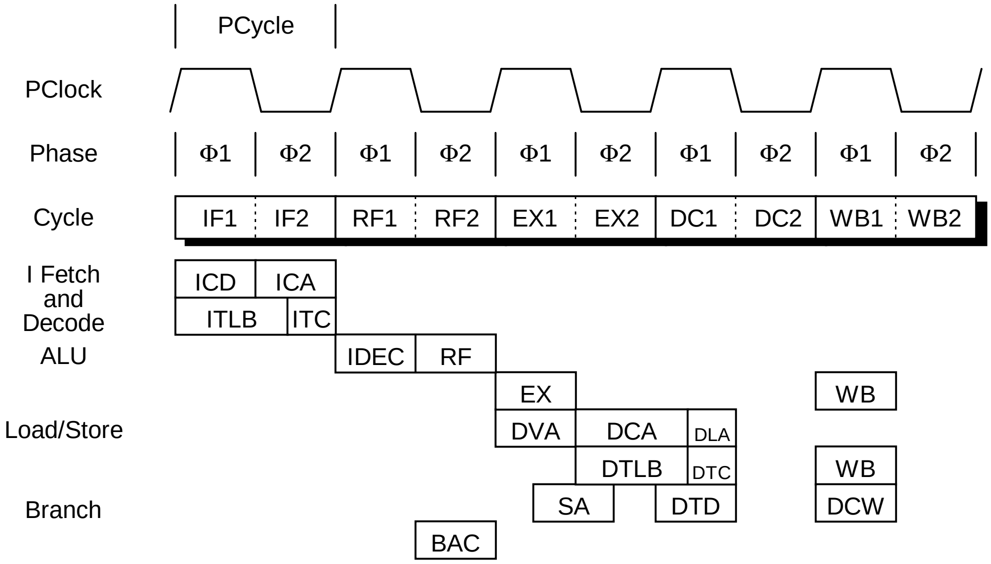


Figure 3-3 Pipeline Activities

Cycle	Begins During this Phase	Mnemonic	Description
IF	Φ1	ICD	Instruction Cache Address Decode
		ITLB	Instruction Address Translation Match/Read
	Φ2	ICA	Instruction Cache array Access
		ITC	Instruction Tag Check
RF	Φ1	IDEC	Instruction Decode
	Φ2	RF	Register operand Fetch
		BAC	Branch Address Calculation
EX	Φ1	EX	Operation Stage
		DVA	Data Virtual Address calculation
		SA	Store Align
	Φ2	DCA	Data Cache Address Decode/array access
		DTLB	Data address Translation Match/Read
DC	Φ1	DLA	Data cache Load Align
		DTC	Data Tag Check
		DTD	Data Transmit to Data cache
WB	Φ1	DCW	Data Cache Write
		WB	Write Back to register file

Table 3-1 Description of Pipeline Showing Stage in Which Activities Commence

3.2 Branch Delay

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

Figure 3-4 illustrates the branch delay and the location of the branch delay slot.

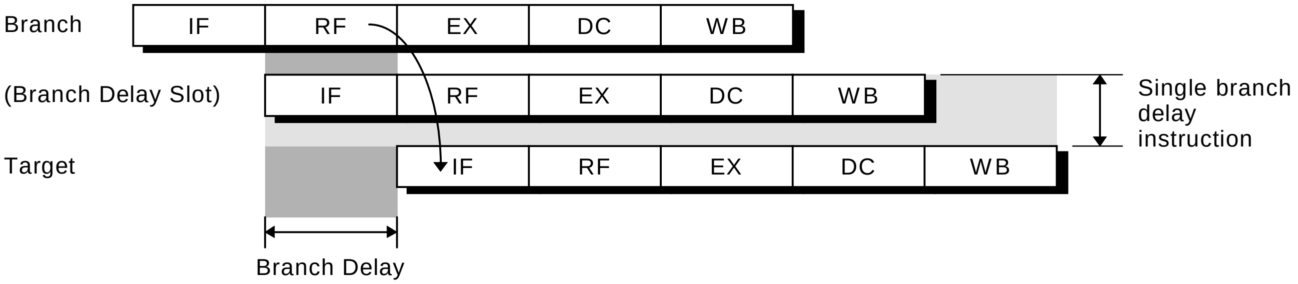


Figure 3-4 CPU Pipeline Branch Delay

3.3 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 VR4100 processor, the instruction immediately following a load instruction can use the contents of the loaded register, however in such cases hardware interlocks insert additional real cycles. Consequently, scheduling load delay slots can be desirable, both for performance and VR-Series processor compatibility. However, the scheduling of load delay slots is not absolutely required in the VR4100.

3.4 Pipeline Operation

The operation of the pipeline is illustrated by the following examples that describe how typical instructions are executed. The instructions described are: ADD, JALR, BEQ, TLT, LW, and SW. Each instruction is taken through the pipeline and the operations that occur in each relevant stage are described.

Add Instruction

ADD rd,rs,rt

IF stage In phase 1 of the IF stage, the eleven least-significant bits of the virtual address are used to address the instruction cache. In phase 2 of the IF stage, the cache index is compared with the page frame number and the cache data is read out. The virtual PC is incremented by 4 so that the next instruction can be fetched.

RF stage 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 1.

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.

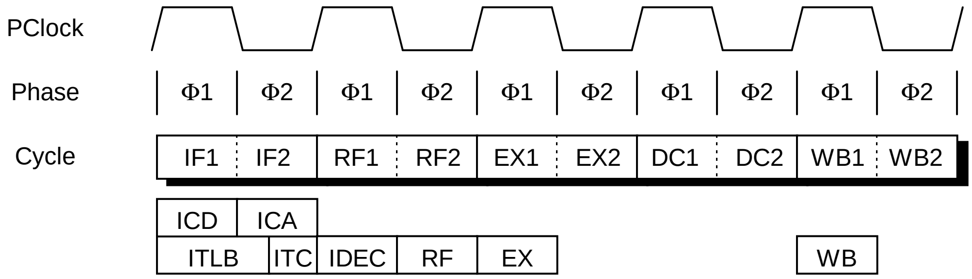


Figure 3-5 Add Instruction Pipeline Activities

Jump and Link Register

JALR rd,rs

IF stage Same as the IF 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. The value of register *rs* is clocked into the virtual PC latch. This value is used in phase 1 to fetch the next instruction.

The value of the virtual PC incremented during the IF 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.

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

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

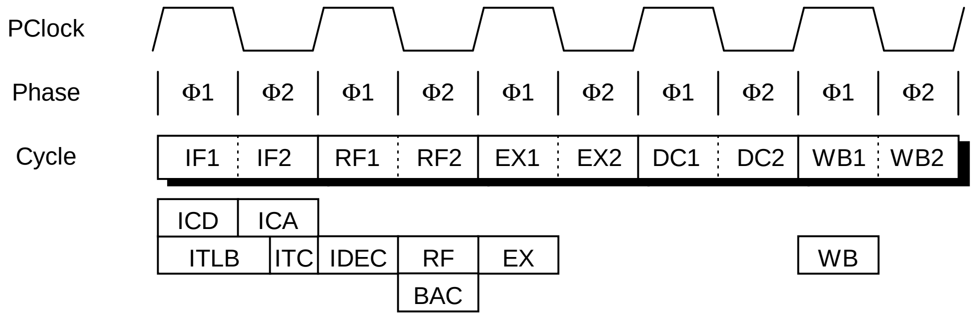


Figure 3-6 Jump and Link Register Instruction Pipeline Activities

Branch on Equal

BEQ rs,rt,offset

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

RF stage During phase 2, the register file is addressed with the *rs* and *rt* fields. During phase 2, 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*.

EX stage 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.

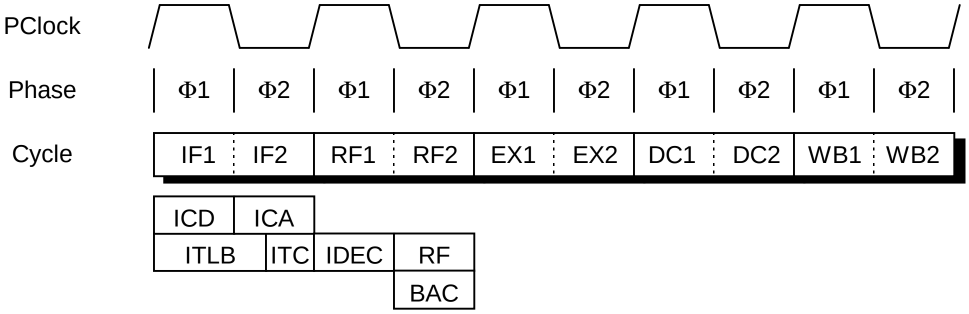


Figure 3-7 Branch on Equal Instruction Pipeline Activities

Trap if Less Than

TLT rs,rt

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

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

EX stage 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 1.

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, a Trap Exception occurs. The PC register is loaded with the value of the exception vector and instructions following in previous pipeline stages are killed.

DC stage No operation

WB stage The *EPC* register is loaded with the value of the PC if the *less than* condition was met in the EX 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.

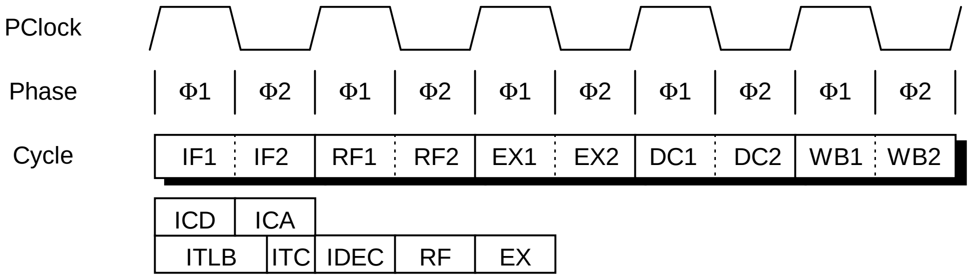


Figure 3-8 Trap if Less Than Instruction Pipeline Activities

Load Word

LW rt,offset(base)

IF stage Same as the IF 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 1 represents the effective virtual address of the operand (DVA).

DC stage 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.

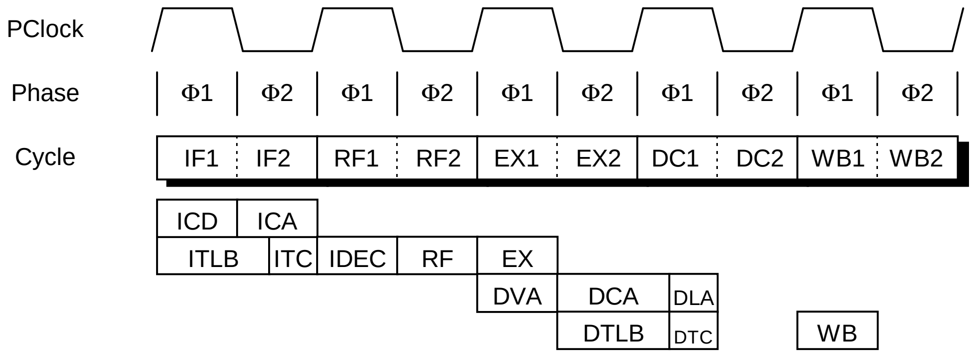


Figure 3-9 Load Word Instruction Pipeline Activities

Store Word

SW rt,offset (base)

IF stage Same as the IF 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 are latched in the output latches during phase 1. 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 1.

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.

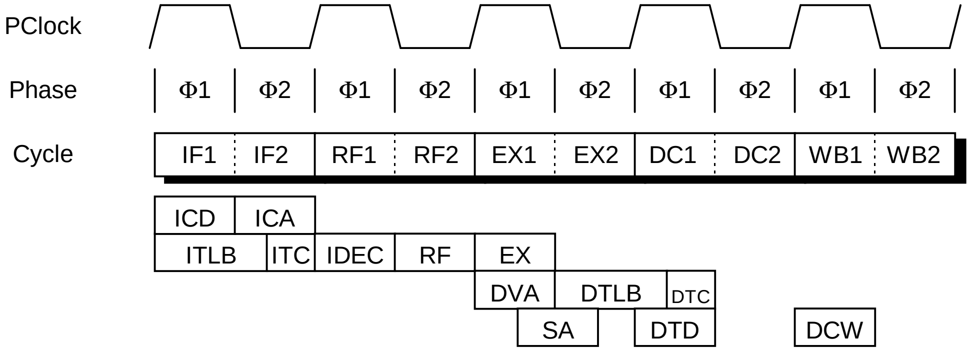


Figure 3-10 Store Word Instruction Pipeline Activities

3.5 Interlock and Exception Handling

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

As shown in Figure 3-11, all interlock and exception conditions are collectively referred to as faults.

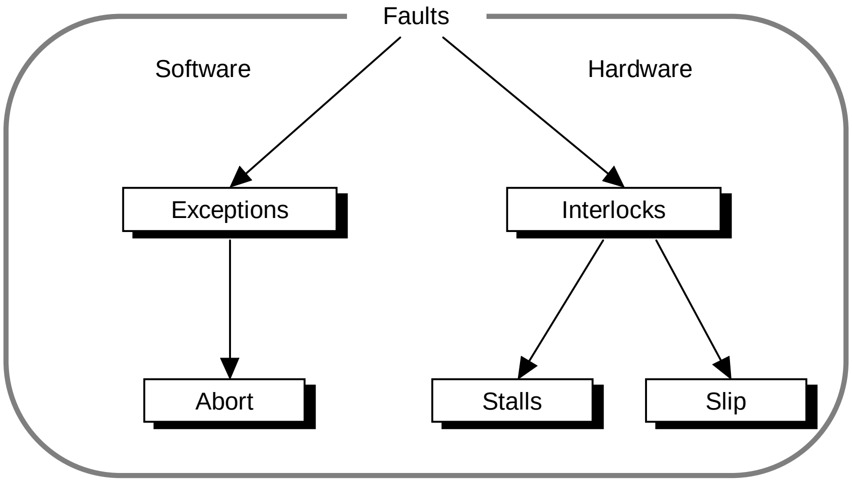
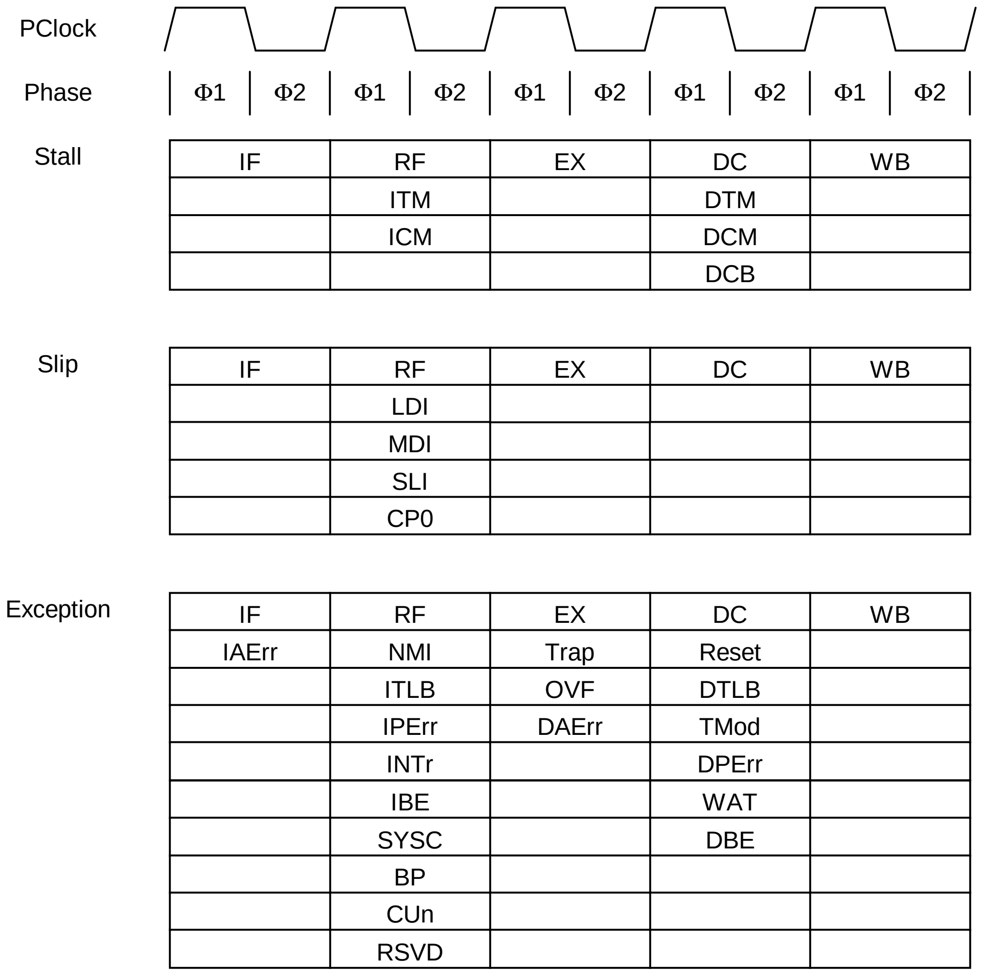


Figure 3-11 Interlocks, Exceptions, and Faults

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

Because each exception or interlock condition corresponds to a particular pipeline stage, a condition can be traced back to the particular instruction in the exception/interlock stage, as shown in Figure 3-12. For instance, an LDI Interlock is raised in the RF stage.

Tables 3-2 through 3-4 describe the pipeline interlocks and exceptions listed in Figure 3-12.



Stall

IF	RF	EX	DC	WB
	ITM		DTM	
	ICM		DCM	
			DCB	

Slip

IF	RF	EX	DC	WB
	LDI			
	MDI			
	SLI			
	CP0			

Exception

IF	RF	EX	DC	WB
IAErr	NMI	Trap	Reset	
	ITLB	OVF	DTLB	
	IPErr	DAErr	TMod	
	INTr		DPErr	
	IBE		WAT	
	SYSC		DBE	
	BP			
	CUn			
	RSVD			

Figure 3-12 Correspondence of Pipeline Stage to Interlock and Exception Condition

Stall	Description
ITM	Instruction TBL Miss
ICM	Instruction Cache Miss
DTM	Data TLB Miss
DCM	Data Cache Miss
DCB	Data Cache Busy

Table 3-2 Description of Pipeline Stall

Slip	Description
LDI	Load Data Interlock
MDI	MD busy Interlock
SLI	Store-Load Interlock
CP0	Coprocessor 0 Interlock

Table 3-3 Description of Pipeline Slip

Exception	Description
IAErr	Instruction Address Error
NMI	Non-maskable Interrupt
ITLB	Instruction Translation exception
IPErr	Instruction Parity Error
INTr	External Interrupt
IBE	Instruction Bus Error
SYSC	System Call
BP	Breakpoint
CUn	Coprocessor Unusable
RSVD	Reserved Instruction
Trap	Trap
OVF	Integer Overflow
DAErr	Data Address Error
Reset	Reset
DTLB	Data Translation exception
DTMod	Data TLB Modified
DPErr	Data Parity Error
WAT	Watch exception
DBE	Data Bus Error

Table 3-4 Description of Pipeline Exception

Exception Conditions

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

When an exceptional conditions is detected for an instruction, the VR4100 will kill it and all following instructions. When this instruction reaches the WB stage, the exception flag causes it to write various CP0 registers with the exception state, change the current PC to the appropriate exception vector address and clear the exception bits of earlier pipeline stages.

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

Figure 3-13 shows the exception detection procedure (e.g., a reserved instruction exception).

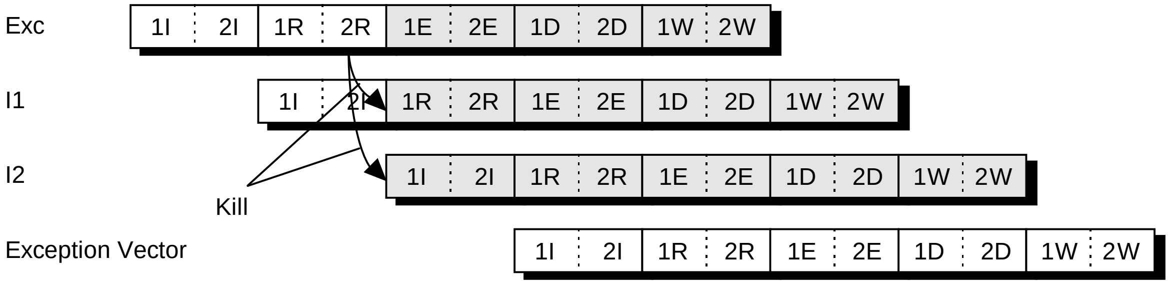
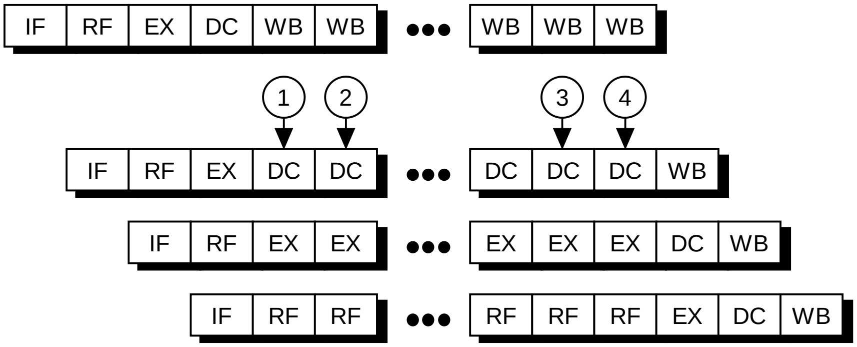

Exc
1I 2I 1R 2R 1E 2E 1D 2D 1W 2W
I1
1I 2I 1R 2R 1E 2E 1D 2D 1W 2W
I2
1I 2I 1R 2R 1E 2E 1D 2D 1W 2W
Kill
Exception Vector
1I 2I 1R 2R 1E 2E 1D 2D 1W 2W

Figure 3-13   Exception Detection

Stall Conditions

★ Stalls are used to stop the pipeline for conditions detected after the RF pipe-stage. When a stall occurs, the processor will resolve the condition and then the pipeline will continue. Figure 3-14 shows a data cache miss stall, and Figure 3-15 shows a CACHE operation stall.


IF RF EX DC WB WB ●●● WB WB WB
1
2
3
4
IF RF EX DC DC ●●● DC DC DC WB
IF RF EX EX ●●● EX EX EX DC WB
IF RF RF ●●● RF RF RF EX DC WB

1 Detect Data Cache Miss

2 Start moving dirty cache line data to write buffer

★ 3 Get first doubleword into cache

4 Load remainder of cache line into cache and restart pipeline

Figure 3-14 Data Cache Miss Stall

If the cache line to be replaced is dirty -- the W bit is set -- the data is moved to the internal write buffer in the next cycle. Then the write back data is written back to memory. The last word of data is returned to the cache in 3 and the pipeline will then restart.

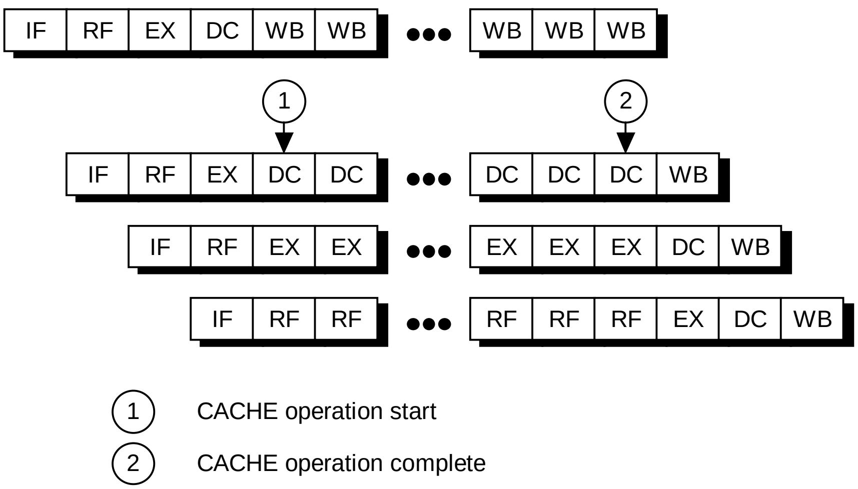

IF RF EX DC WB WB ●●● WB WB WB
IF RF EX DC DC ●●● DC DC DC WB
IF RF EX EX ●●● EX EX EX DC WB
IF RF RF ●●● RF RF RF EX DC WB
1
2
1 CACHE operation start
2 CACHE operation complete

Figure 3-15 CACHE Operation Stall

When the CACHE operation enters the DC pipe-stage, the pipeline stalls while the CACHE operation is serviced. The pipeline begins running again when the CACHE operation is complete, allowing the instruction fetch to proceed.

Slip Conditions

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

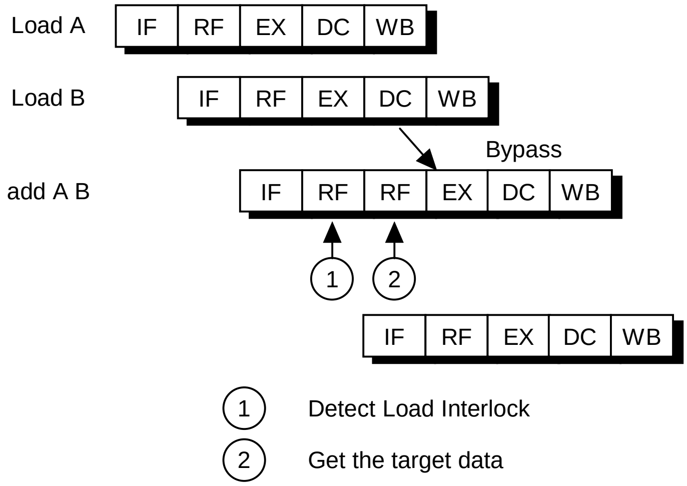


Figure 3-16 Load Data Interlock

★ Load Data Interlock is detected in RF shown in as Figure 3-16 and the pipeline slips in its RF stage. Load Data Interlock occurs when data fetched by a load instruction and data moved from HI, LO or CP0 register is required by the next immediate instruction. The pipeline begins running again when the clock after the target of the load is read from the data cache, HI, LO and CP0 register. The data returned at the end of the DC stage is input into the end of the RF stage, using the bypass multiplexers.

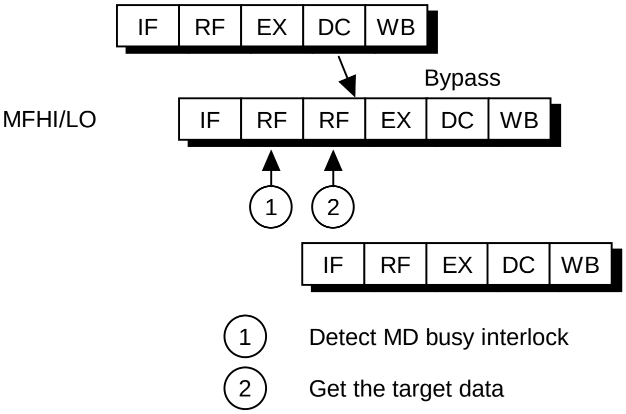


Figure 3-17 MD busy Interlock

MD Busy Interlock is detected in RF as shown in Figure 3-17 and the pipeline slips in its RF stage. MD Busy Interlock occurs when Hi/Lo register is required by MFHi/Lo operation before finishing Mult/Div execution. The pipeline begins running again the clock after finishing Mult/Div execution. The data returned from the HI/LO register at the end of the DC stage is input into the end of the RF stage, using the bypass multiplexers. ★

Store-Load Interlock is detected in EX stage and the pipeline slips in RF stage. Store-Load Interlock occurs when Store operation followed by Load operation is detected. The pipeline begins running again one clock after.

Coprocessor0 Interlock is detected in EX stage and the pipeline slips in RF stage. Coprocessor Interlock occurs when MTC0 config/status operation is detected. The pipeline begins running again one clock later.

Bypassing

In some cases, data and conditions produced in the EX and DC stages of the pipeline are made available to the RF 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 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.

3.6 Code Compatibility

The VR4100 has been designed with consideration to program compatibility with other VR-Series processors. However, its architecture differs in several points with that of other processors, so that programs that can be executed on other processors may not run on the VR4100. Similarly, programs that can be executed on the VR4100 cannot necessarily be run on other processors. Cautions that should be noted when porting programs between the VR4100 and other processors are listed below.

- The VR4100 does not have a floating-point unit (FPU) and thus does not support FPU instructions.
- Product-sum instructions (DMADD16, MADD16) have been added to the VR4100.
- Power mode instructions (HIBERNATE, STANDBY, SUSPEND) have been added to the VR4100 because it supports Power mode.
- The VR4100 does not have the LL bit used for synchronization in a multiprocessor operating environment. Therefore, the VR4100 does not support instructions which manipulate the LL bit (LL, LLD, SC, SCD).

For more information on instructions, refer to Chapter 14 and the VR4000, VR4400, and VR4200™ User's Manuals.

Memory Management

4

The VR4100 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.

4.1 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 ranges in size from 1 Kbytes to 256 Kbytes, in multiples of 4 -- that is 1K, 4K, 16K, 64K, 256K.

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 4-1).

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 and the TLB may be disabled. The TLB-Shutdown *(TS)* bit in the *Status* register is set to 1 if the TLB is disabled.

[†] 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.

4.2 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 4-1 shows the translation of a virtual address into a physical address.

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

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

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

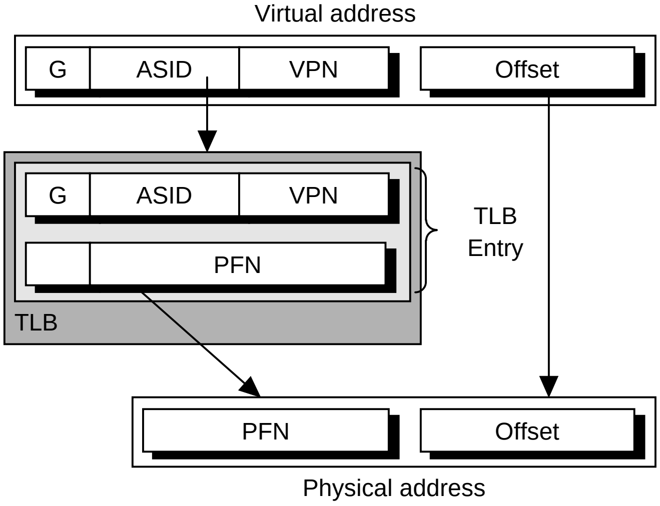


Figure 4-1 Overview of a Virtual-to-Physical Address Translation

[†] Figure 4-8 shows the 32-bit and 64-bit versions of the processor TLB entry.

As shown in Figure 4-2 and Figure 4-3, 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.

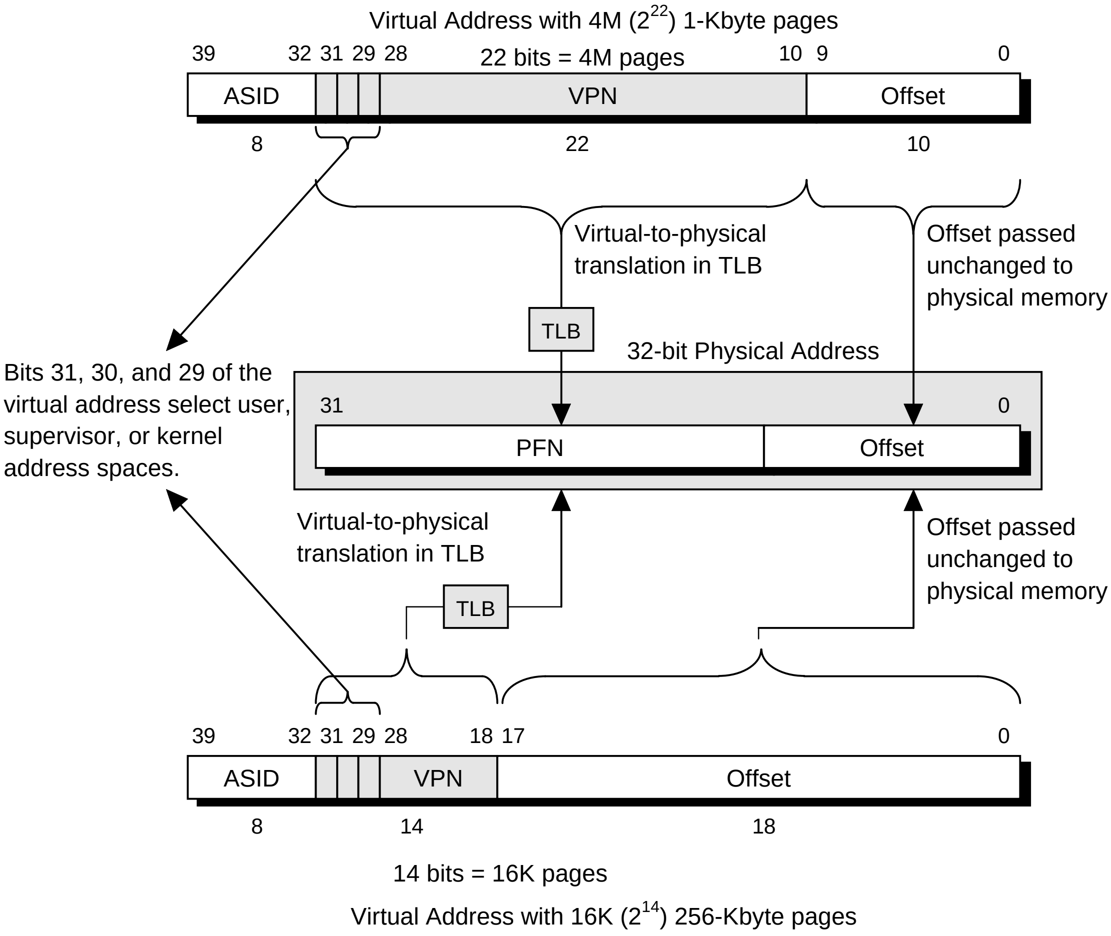
Virtual-to-physical translation is described in greater detail throughout the remainder of this chapter; Figure 4-19 is a flow diagram of the process shown at the end of this chapter.

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

32-bit Mode Address Translation

Figure 4-2 shows the virtual-to-physical-address translation of a 32-bit mode address. This figure illustrates the two possible page sizes: a 1-Kbyte page (10 bits) and a 256-Kbyte page (18 bits).

- The top portion of Figure 4-2 shows a virtual address with a 10-bit, or 1-Kbyte, page size, labelled *Offset*. The remaining 22 bits of the address represent the VPN, and index the 4M-entry page table.
- The bottom portion of Figure 4-2 shows a virtual address with a 18-bit, or 256-Kbyte, page size, labelled *Offset*. The remaining 14 bits of the address represent the VPN, and index the 16K-entry page table.



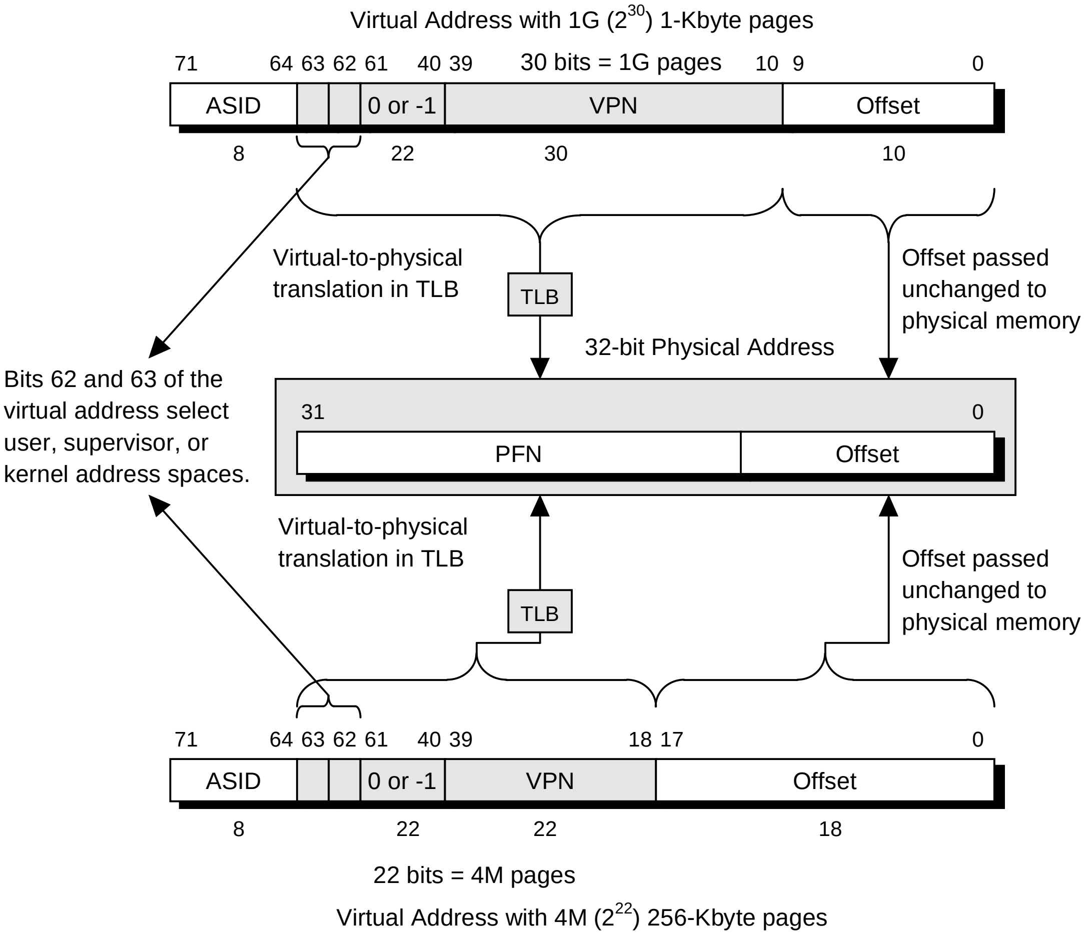
Virtual Address with 4M (2^{22}) 1-Kbyte pages
39
32
31
29
28
22 bits = 4M pages
10
9
0
ASID
VPN
Offset
8
22
10
Virtual-to-physical translation in TLB
Offset passed unchanged to physical memory
TLB
32-bit Physical Address
Bits 31, 30, and 29 of the virtual address select user, supervisor, or kernel address spaces.
31
0
PFN
Offset
Virtual-to-physical translation in TLB
Offset passed unchanged to physical memory
TLB
39
32
31
29
28
18
17
0
ASID
VPN
Offset
8
14
18
14 bits = 16K pages
Virtual Address with 16K (2^{14}) 256-Kbyte pages

Figure 4-2   32-bit Mode Virtual Address Translation

64-bit Mode Address Translation

Figure 4-3 shows the virtual-to-physical-address translation of a 64-bit mode address. This figure illustrates the two possible page sizes: a 1-Kbyte page (10 bits) and a 256-Kbyte page (18 bits).

- The top portion of Figure 4-3 shows a virtual address with a 10-bit, or 1-Kbyte, page size, labelled *Offset*. The remaining 30 bits of the address represent the VPN, and index the 1G-entry page table.
- The bottom portion of Figure 4-3 shows a virtual address with a 18-bit, or 256-Kbyte, page size, labelled *Offset*. The remaining 22 bits of the address represent the VPN, and index the 4M-entry page table.



Virtual Address with 1G (2^{30}) 1-Kbyte pages
71
64 63 62 61
40 39
30 bits = 1G pages
10 9
0
ASID
0 or -1
VPN
Offset
8
22
30
10
Virtual-to-physical translation in TLB
TLB
Offset passed unchanged to physical memory
32-bit Physical Address
Bits 62 and 63 of the virtual address select user, supervisor, or kernel address spaces.
31
0
PFN
Offset
Virtual-to-physical translation in TLB
TLB
Offset passed unchanged to physical memory
71
64 63 62 61
40 39
18 17
0
ASID
0 or -1
VPN
Offset
8
22
22
18
22 bits = 4M pages
Virtual Address with 4M (2^{22}) 256-Kbyte pages

Figure 4-3   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 4-4 shows User mode virtual address space.

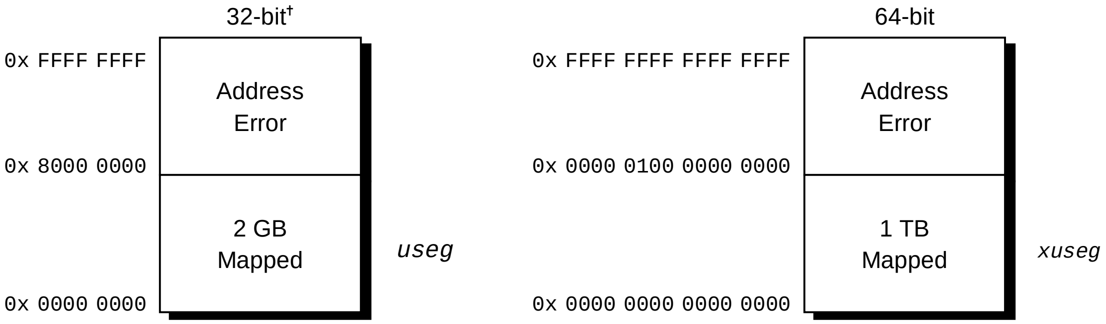


Figure 4-4 User Mode Virtual Address Space[†]

[†] The VR4100 uses 64-bit addresses internally. In Kernel mode, the processor saves, resets and initializes each register before switching context. In 32-bit mode, addresses consist of 32 bits with bit 31 used as sign extension for bits 32 to 63. Normally, 32-bit mode programs cannot generate invalid addresses. However, when context switching occurs and Kernel mode is entered, values other than previously sign-extended 32-bit addresses may be saved to 64-bit registers. In such a case, the User mode program may generate invalid addresses.

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

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

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

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

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

Table 4-1 lists the characteristics of the two user mode segments, *useg* and *xuseg*.

Address Bit Values	Status Register Vit 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)

Table 4-1 32-bit and 64-bit User Mode Segments

[†] The cached *(C)* field in a TLB entry determines whether the reference is cached; see Figure 4-8.

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 4-4, 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.

In 32-bit User mode addressing, the TLB refill exception vector is used for TLB misses.

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 4-4. 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.

The extended addressing TLB refill exception vector is used for TLB misses.

Supervisor Mode Operations

Supervisor mode is designed for layered operating systems in which a true kernel runs in VR4100 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
- when *SX* = 1, 64-bit supervisor space is selected

Figure 4-5 shows Supervisor mode address mapping. Table 4-2 lists the characteristics of the supervisor mode segments; descriptions of the address spaces follow.

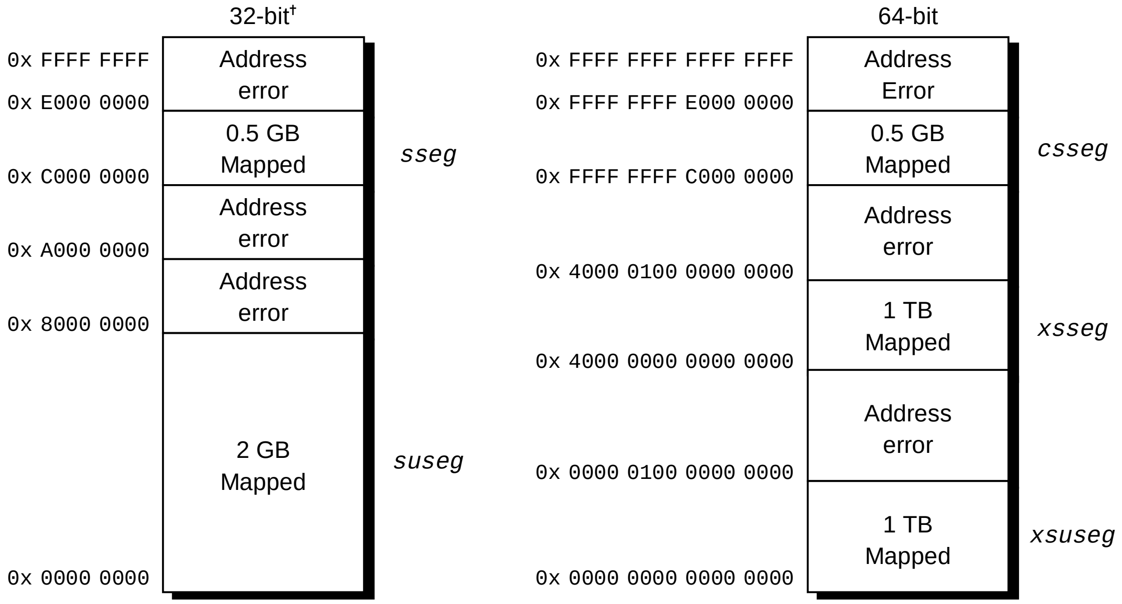


Figure 4-5   Supervisor Mode Address Space[†]

[†] The VR4100 uses 64-bit addresses internally. In 32-bit mode, addresses consist of 32 bits with bit 31 used as sign extension for bits 32 to 63.
Normally, 32-bit mode programs cannot generate invalid addresses. However, when calculating addresses, the base register + offset operation may generate a 2's complement overflow. At such time, the generated address is invalid and the result becomes undefined. Overflow may be caused in either of the following 2 cases.

- Offset bit 15 = 0, base register bit 31 = 0, (base register + offset) bit 31 = 1
- Offset bit 15 = 1, base register bit 31 = 1, (base register + offset) bit 31 = 0

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 Gbyte (2^{31} bytes)
32-bit A (31:29) = 110_2	01_2	0	0	0	sseg	0xC000 0000 through 0xDFFF FFFF	512 Mbyte (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 Mbyte (2^{29} bytes)

Table 4-2 32-bit and 64-bit Supervisor Mode Segments

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

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

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

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

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

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

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

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

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

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

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

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

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

In Supervisor mode, when *SX* = 1 in the *Status* register and bits 63:62 of the virtual address are set to 11_2, the *csseg* separate supervisor virtual address space is selected*.* 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
- when *KX* = 1, 64-bit kernel space is selected

The processor enters Kernel mode whenever an exception is detected and it remains in Kernel mode until an Exception Return (ERET) instruction is executed 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 4-6. Table 4-3 lists the characteristics of the 32-bit kernel mode segments, and Table 4-4 lists the characteristics of the 64-bit kernel mode segments.

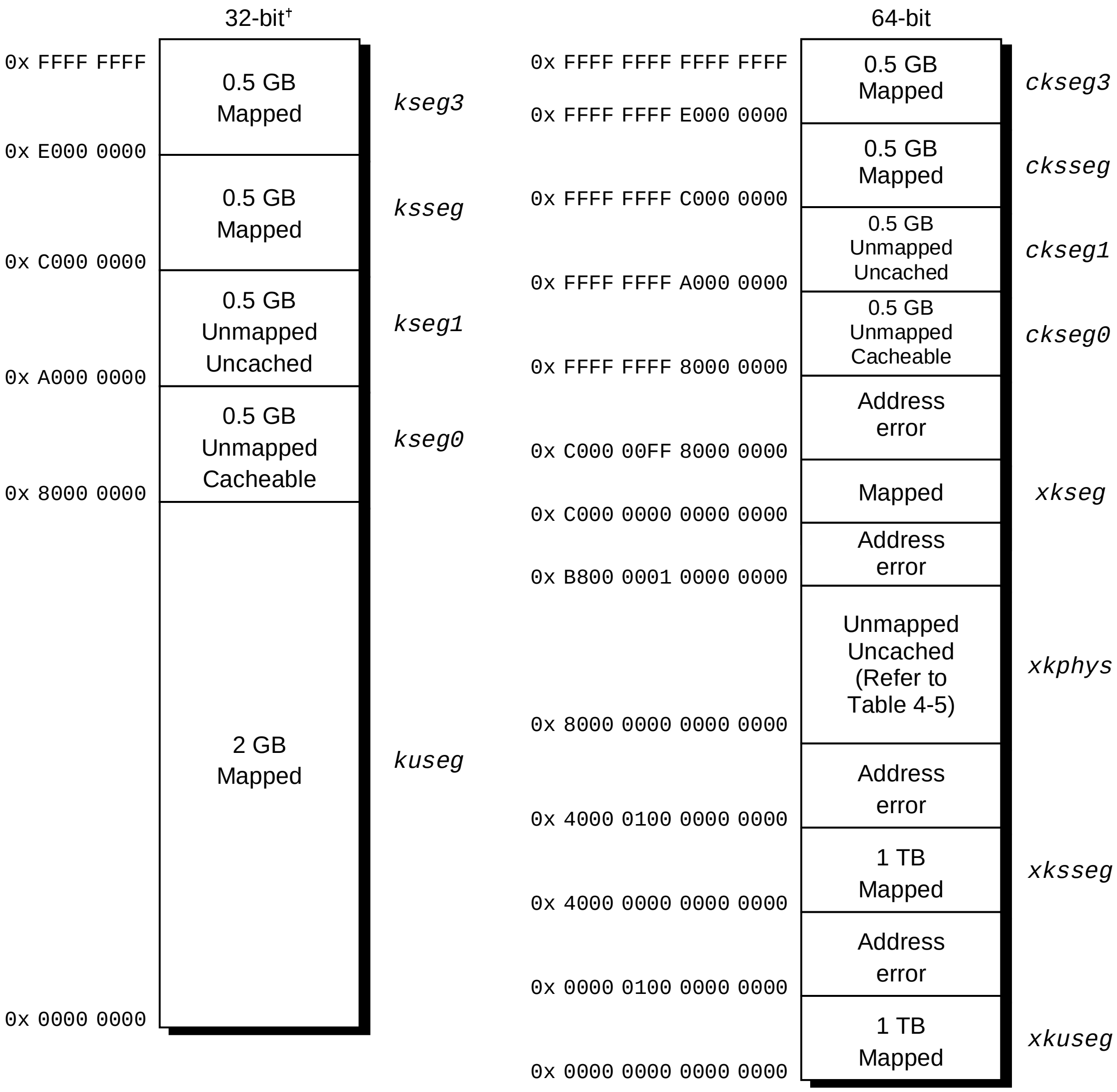


Figure 4-6 Kernel Mode Address Space[*]

[†] The VR4100 uses 64-bit addresses internally. In 32-bit mode, addresses consist of 32 bits with bit 31 used as sign extension for bits 32 to 63.
Normally, 32-bit mode programs cannot generate invalid addresses. However, when calculating addresses, the base register + offset operation may generate a 2's complement overflow. At such time, the generated address is invalid and the result becomes undefined. Overflow may be caused in either of the following 2 cases.

- Offset bit 15 = 0, base register bit 31 = 0, (base register + offset) bit 31 = 1
- Offset bit 15 = 1, base register bit 31 = 1, (base register + offset) bit 31 = 0

Address Bit Values	Status Register Bit 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 Gbyte (2^{31} bytes)
A (31:29) = 100_2				0	kseg0	0x8000 0000 through 0x9FFF FFFF	512 Mbyte (2^{29} bytes)
A (31:29) = 101_2				0	kseg1	0xA000 0000 through 0xBFFF FFFF	512 Mbyte (2^{29} bytes)
A (31:29) = 110_2				0	ksseg	0xC000 0000 through 0xDFFF FFFF	512 Mbyte (2^{29} bytes)
A (31:29) = 111_2				0	kseg3	0xE000 0000 through 0xFFFF FFFF	512 Mbyte (2^{29} bytes)

Table 4-3 32-bit Kernel Mode Segments

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

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

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

In Kernel mode, when *KX* = 0 in the *Status* register and the most-significant three bits of the virtual address are 100_2, 32-bit *kseg0* virtual address space is selected; it is the current 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 current 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.

Address Bit Values	Status Register Bit 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	xkuseg	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	$8*2^{32}$-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 Mbyte (2^{29} bytes)
A (63:62) = 11_2 A (61:31) = -1				1	ckseg1	0xFFFF FFFF A000 0000 through 0xFFFF FFFF BFFF FFFF	512 Mbyte (2^{29} bytes)
A (63:62) = 11_2 A (61:31) = -1				1	cksseg	0xFFFF FFFF C000 0000 through 0xFFFF FFFF DFFF FFFF	512 Mbyte (2^{29} bytes)
A (63:62) = 11_2 A (61:31) = -1				1	ckseg3	0xFFFF FFFF E000 0000 through 0xFFFF FFFF FFFF FFFF	512 Mbyte (2^{29} bytes)

Table 4-4 64-bit Kernel Mode Segments

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

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

As a special feature for the Cache Error handler, if the ERL bit of the Status register is set, the user address region becomes a 2^{31}-byte unmapped, uncached space. This allows the Cache Error exception code to operate uncached using *r0* as a base register.

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.

★

Value (61:59)	Cacheability and Coherency Attributes	Starting Address
0	Cached	0x8000 0000 0000 0000 through 0x8000 0000 FFFF FFFF
1	Cached	0x8800 0000 0000 0000 through 0x8800 0000 FFFF FFFF
2	Uncached	0x9000 0000 0000 0000 through 0x9000 0000 FFFF FFFF
3	Cached	0x9800 0000 0000 0000 through 0x9800 0000 FFFF FFFF
4	Cached	0xA000 0000 0000 0000 through 0xA000 0000 FFFF FFFF
5	Cached	0xA800 0000 0000 0000 through 0xA800 0000 FFFF FFFF
6	Cached	0xB000 0000 0000 0000 through 0xB000 0000 FFFF FFFF
7	Cached	0xB800 0000 0000 0000 through 0xB800 0000 FFFF FFFF

Table 4-5 Cacheability and Coherency Attributes

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

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

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

64-bit Kernel Mode, Compatibility Spaces (*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 bits 30:29 of address, as shown in Figure 4-6, 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 *ksse*g.
- *ckseg3*. This 64-bit virtual address space is kernel virtual space, compatible with the 32-bit address model *kseg3*.

4.3 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 4-7 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.

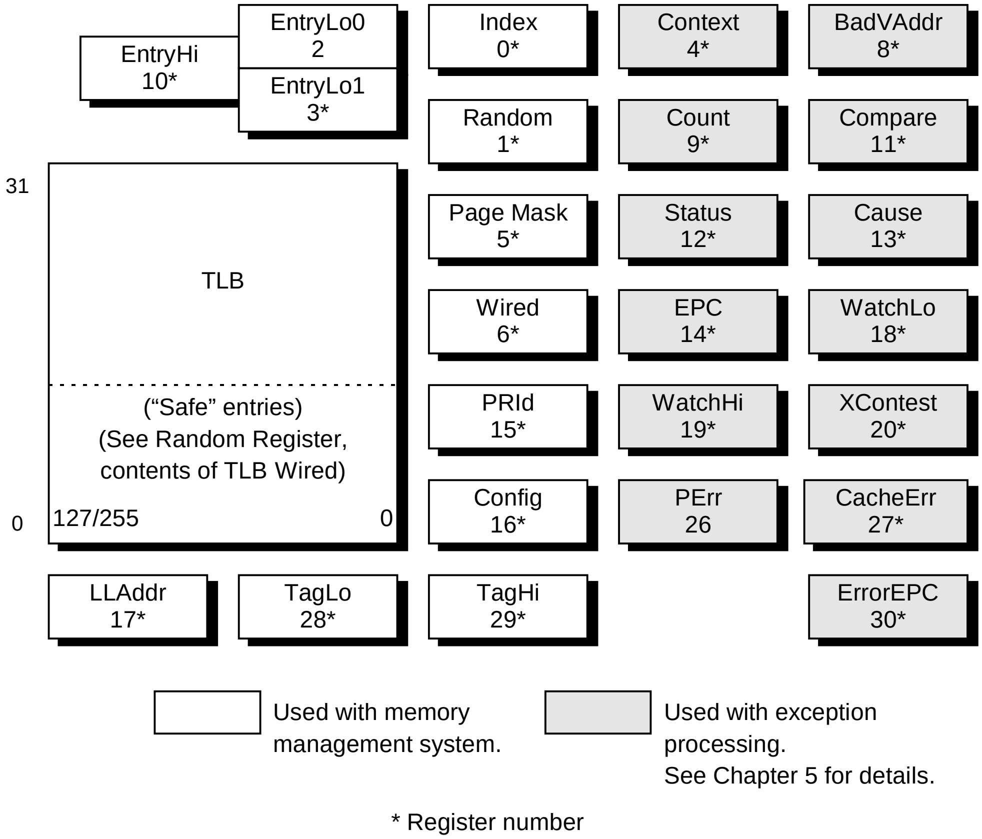


Figure 4-7 CP0 Registers and the TLB

Format of a TLB Entry

Figure 4-8 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 Figure 4-9 and Figure 4-10; for example the *Mask* field of the TLB entry is also held in the *PageMask* register.

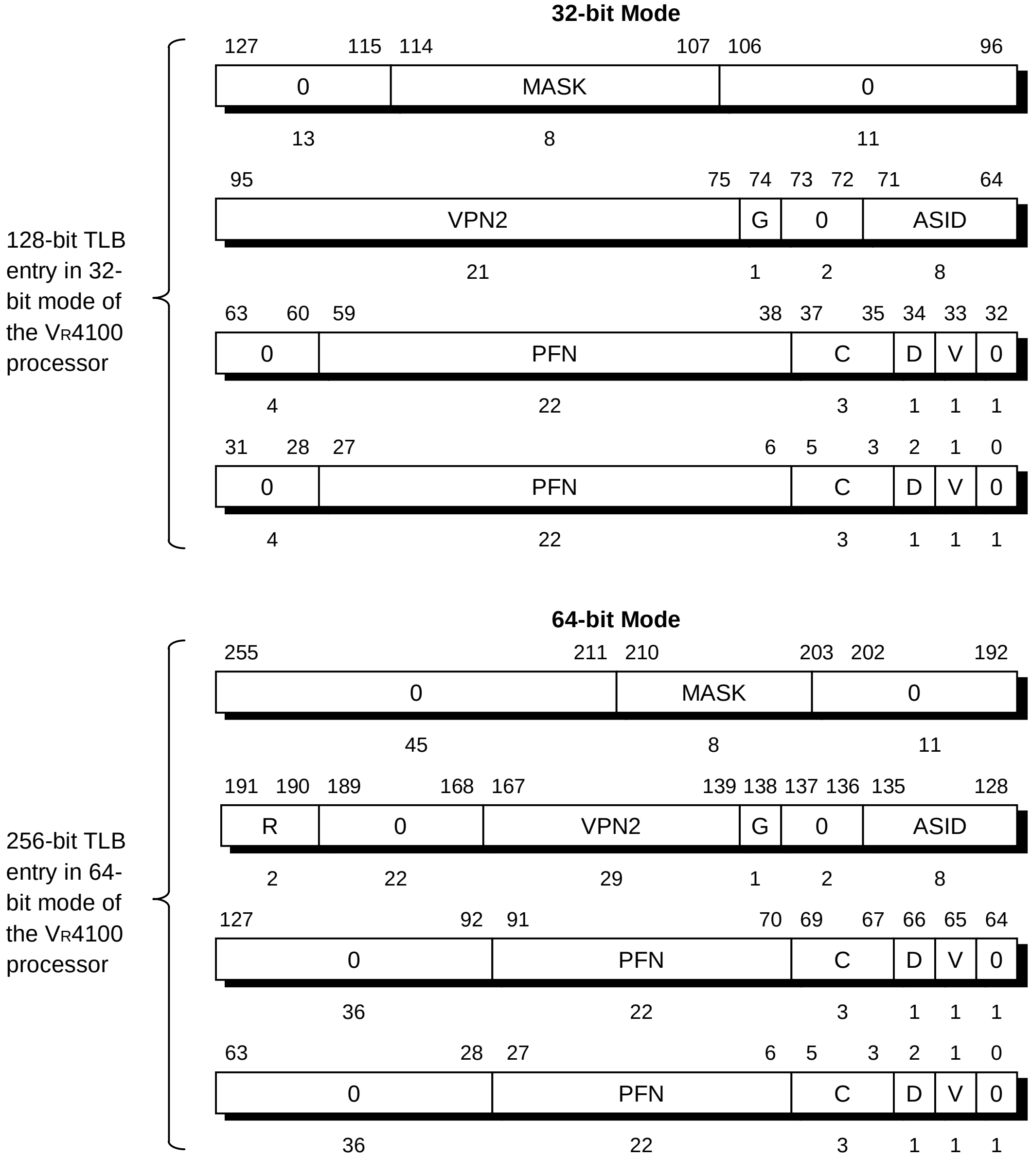
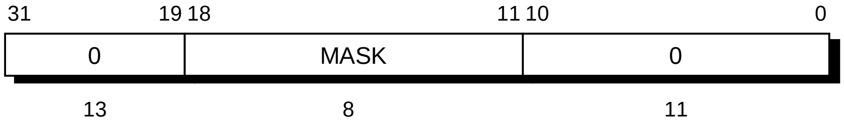


Figure 4-8 Format of a TLB Entry

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 4-9 and 4-10 describe the TLB entry fields shown in Figure 4-8.

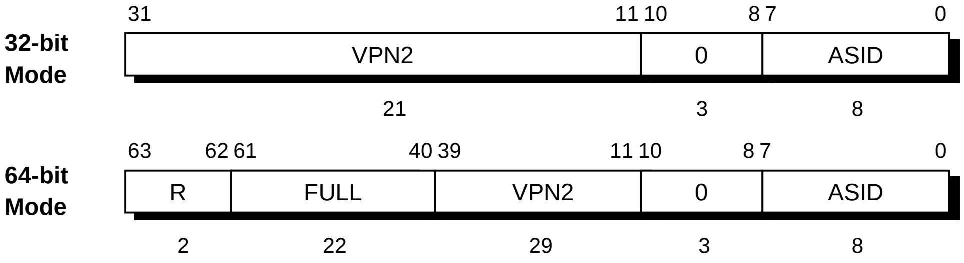
PageMask Register



Mask ..... Page comparison mask; see Table 4-9.

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

EntryHi Register



VPN2 ..... Virtual page number divided by two (maps to two pages).

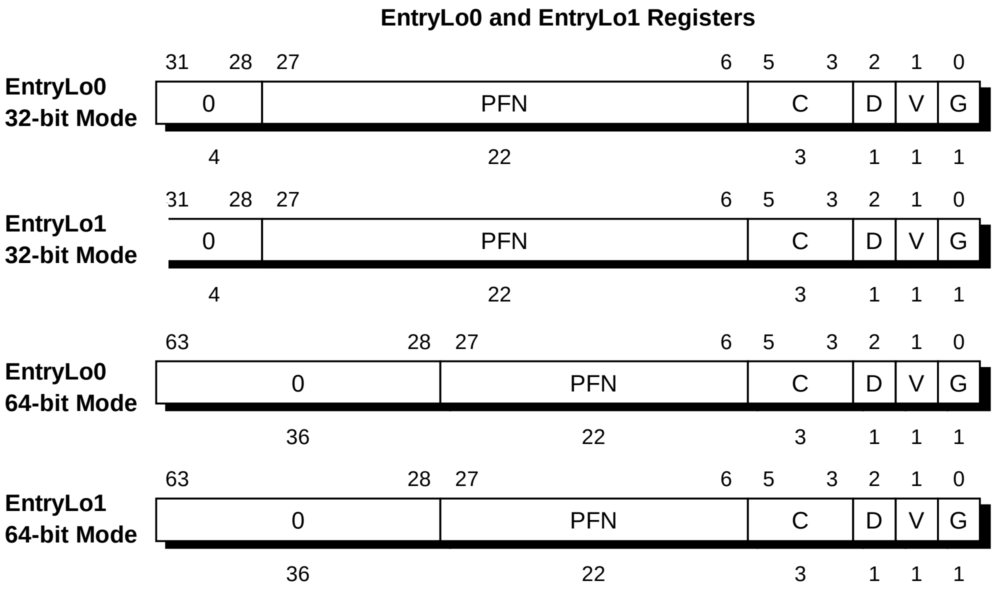
ASID ...... Address space ID field. An 8-bit field that lets multiple processes share the TLB; each process has a distinct mapping of otherwise identical virtual page numbers.

R .... Region. (00 -> user, 01 -> supervisor, 11 -> kernel) used to match $vAddr_{63...62}$

Fill .... Reserved; 0 on read; Ignored on write.

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

Figure 4-9 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 4-6.
D .... Dirty. If this bit is set, the page is marked as dirty and, therefore, writable. This bit is actually a write-protect bit that software can use to prevent alteration of data.
V .... Valid. If this bit is set, it indicates that the TLB entry is valid; otherwise, a TLBL or TLBS miss occurs.
G .... Global. If this bit is set in both Lo0 and Lo1, then the processor ignores the ASID during TLB lookup.
0 .... Must be written as zeroes, and returns zeroes when read.

Figure 4-10 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 4-6 shows the page attributes selected by the *C* bits.

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

Table 4-6 TLB Page (C) Bit Values

CP0 Registers

The following sections describe the CP0 registers, shown in Figure 4-7, 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.

Figure 4-11 shows the format of the *Index* register; Table 4-7 describes the *Index* register fields.

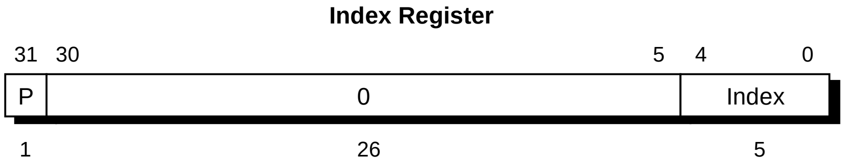


Figure 4-11   Index Register

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

Table 4-7   Index Register Field Descriptions

Random Register (1)

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 4-12 shows the format of the *Random* register; Table 4-8 describes the *Random* register fields.

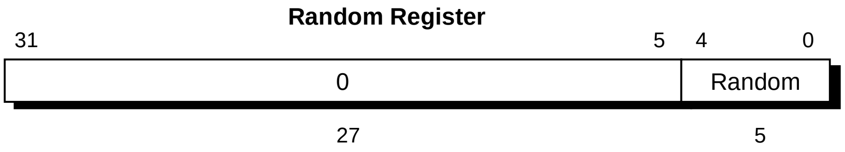


Figure 4-12   Random Register

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

Table 4-8   Random Register Field Descriptions

EntryLo0 (2), and EntryLo1 (3) Registers

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

- *EntryLo0* is used for even virtual pages.
- *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 4-10 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 4-9. Page sizes must be from 1 Kbytes to 256 Kbytes. The format of the *PageMask* register is shown in Figure 4-9.

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 18:11 are used in the comparison. When the *Mask* field is not one of the values shown in Table 4-9, the operation of the TLB is undefined.

Page Size	Bit							
	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1
1 Kbytes	0	0	0	0	0	0	0	0
4 Kbytes	0	0	0	0	0	0	1	1
16 Kbytes	0	0	0	0	1	1	1	1
64 Kbytes	0	0	1	1	1	1	1	1
256 Kbytes	1	1	1	1	1	1	1	1

Table 4-9   Mask Field Values for Page Sizes

Wired Register (6)

The *Wired* register is a read/write register that specifies the boundary between the *wired* and *random* entries of the TLB as shown in Figure 4-13. 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.

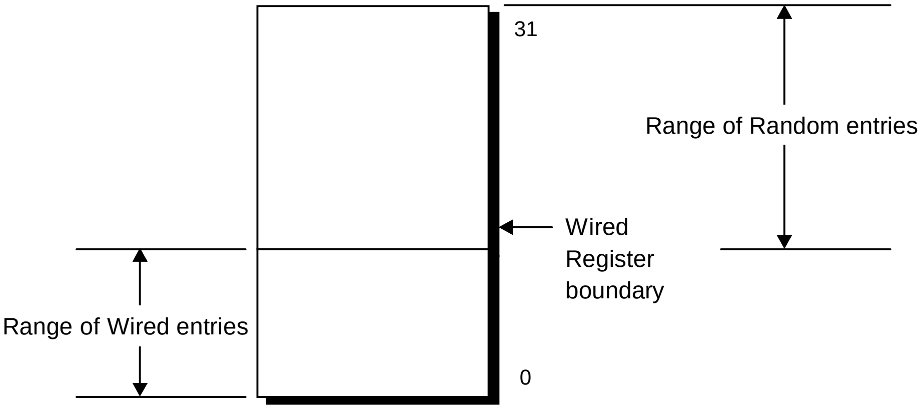


Figure 4-13   Wired Register Boundary

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 4-14 shows the format of the *Wired* register; Table 4-10 describes the register fields.

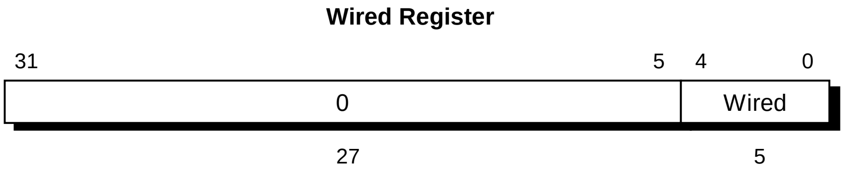


Figure 4-14   Wired Register

Field	Description
Wired	TLB Wired boundary (the number of wired TLB entries)
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 4-10   Wired Register Field Descriptions

EntryHi Register (CP0 Register 10)

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

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

Figure 4-9 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 4-15 shows the format of the *PRId* register; Table 4-11 describes the *PRId* register fields.

PRId Register

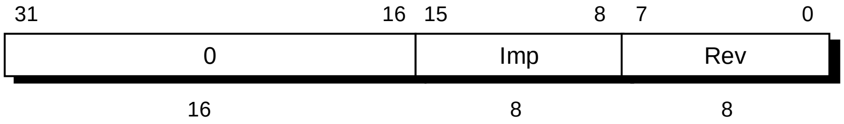


Figure 4-15 Processor Revision Identifier Register Format

Field	Description
Imp	Implementation number (0x0C for the VR4100)
Rev	Revision number
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 4-11 PRId Register Fields

The low-order byte (bits 7:0) of the *PRId* register is interpreted as a revision number, and the high-order byte (bits 15:8) is interpreted as an implementation number. The 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 selected on VR4100 processors; Table 4-12 lists these options.

Some configuration options, as defined by the *EC* and *BE* fields, are set by the hardware during Cold Reset (as derived from the state of dedicated pins on the VR4100) and are included in the *Config* register as read-only status bits for the software to access. Other configuration options are read/write (*AD, EP and K0* fields) and controlled by software; on Cold Reset these fields are undefined. Since only a subset of the R4000 options are available in the VR4100 processor, some bits are set to constants (e.g., bits 14:13) that were variable in the R4000. The *Config* register should be initialized by software before caches are used.

Figure 4-16 shows the format of the *Config* register; Table 4-12 describes the *Config* register fields.

Config Register

31	30 28	27 24	23	22 18	17	16	15	14	13 3	2 0
0	EC	EP	AD	0	1	0	BE	1	0	K0
1	3	4	1	5	1	1	1	1	11	3

Figure 4-16   Config Register Format

When the AD bit is clear to zero, processor read/write address cycle issue rate is max 4 TClock cycles (ADxxADxx or ADDxADDx etc.).

When the AD bit is set to one, that is max 2 TClock cycles (ADAD or ADDADD etc.).

Field	Description
EC	System clock ratio: 0 -> processor clock frequency divided by 2 1-6 -> Reserved 7 -> processor clock frequency divided by 1
EP	Writeback data rate: [Note 2] 0 -> DDDD word every cycle 1 -> Reserved 2 -> Reserved 3 -> DxDxDxDx 2 words every 4 cycles 4 -> Reserved 5 -> Reserved 6 -> DxxDxxDxxDxx 2 words every 6 cycles 7 -> Reserved 8 -> DxxxDxxxDxxxDxxx 2 words every 8 cycles 9-15 Reserved
BE	BigEndianMem 0 -> Little endian 1 -> Big endian
AD	Accelerate Data ratio 0 -> R4x00 compatible mode 1 -> Accelerate mode
IC	0 on Read [Note 1]
DC	0 on Read [Note 1]
IB	Primary I-cache line size 0 -> 16 bytes
DB	Primary D-cache line size 0 -> 16 bytes
K0	Kseg0 coherency algorithm (see EntryLo0 and EntryLo1 registers)
Others	Reserved. Returns indicated values when read.

Notes **1.** In the VR4100, each primary I-cache/D-cache is less than $2^{12 + IC}$ bytes ($2^{12 + IC}$ bytes is defined on MIPS archtechture in IC or DC field).

Therefore, these field have no meaning.

2. The VR4100 may used immediately after last D cycle.

Table 4-12 Config Register Field

Load Linked Address (LLAddr) Register (17)

The read/write *Load Linked Address* (*LLAddr*) register is not used with the VR4100 processor except for diagnostic purpose, and serves no function during normal operation.

LLAddr register is implemented just for a compatibility between the VR4100 to VR4000/4400.

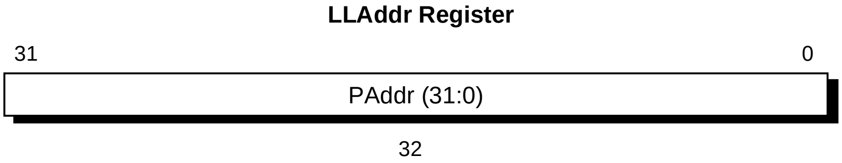


Figure 4-17   LLAddr register

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

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

When the CE bit of the Status register is cleared to zero, the *P* fields of these registers are ignored on Index Store Tag operations. Parity is computed by the store operation.

Figure 4-18 shows the format of these registers for primary cache operations. Table 4-13 lists the field definitions of the *TagLo* and *TagHi* registers.

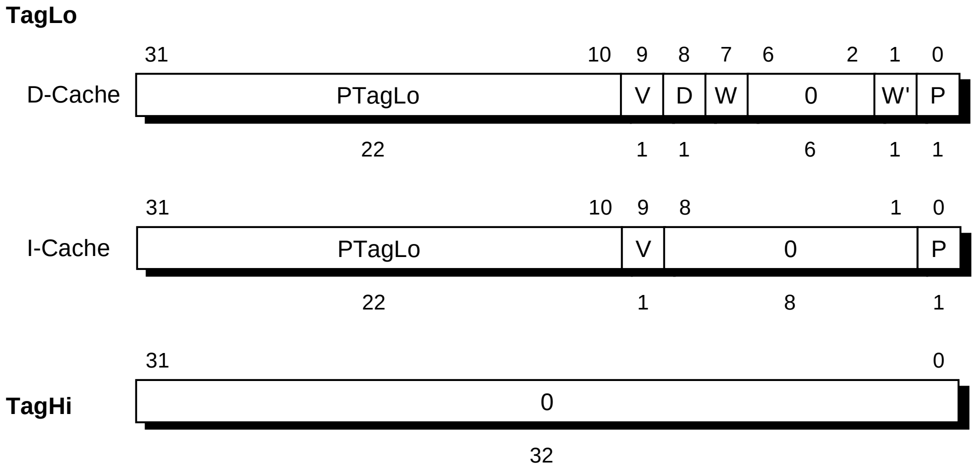


Figure 4-18 TagLo and TagHi Register Formats

Field	Description
PTagLo	Specifies the physical address bits 31:10.
W•	Even Parity for the write-back bit
D	Dirty bit
W	Write-back bit (set if cache line has been written)
V	Valid bit
P	Specifies the primary tag even parity bit.
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 4-13 Cache Tag Register Fields

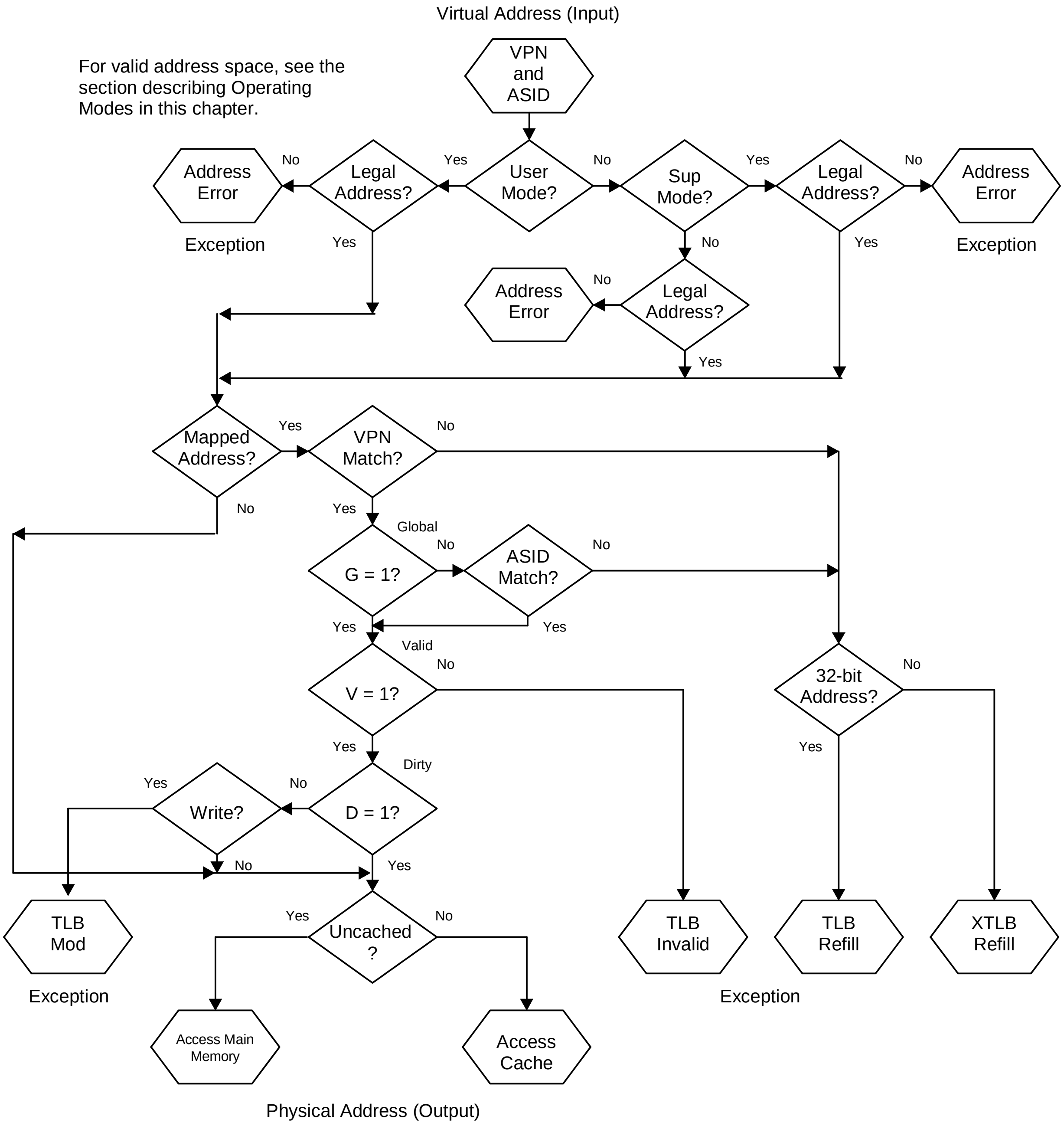
Virtual-to-Physical Address Translation Process

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

- In 32-bit mode, the highest 13 to 21 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 24 to 29 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 4-19 illustrates the TLB address translation process.



For valid address space, see the section describing Operating Modes in this chapter.

Figure 4-19   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 4-14 lists the instructions that the CPU provides for working with the TLB. See Chapter 14 for a detailed description of these instructions.

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

Table 4-14   TLB Instructions

[†] TLB miss exceptions are described in Chapter 5.

[MEMO]

CPU Exception Processing

5

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.

5.1 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 VR4100 processor supports a supervisor mode and fast TLB refill for all address spaces. The VR4100 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 (see the description of the ERET instruction in Chapter 14).

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.

5.2 Precision of Exceptions

VR4100 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.

There is a special case in which the VR4100 processor may not be able to restart easily after servicing an exception. When a Cache Data Parity Error (DPErr) exception occurs on a load with a cache hit, the VR4100 processor does not prevent the cache data (with erroneous parity) from being written back into the register file during the WB stage. The exception is still precise, since both the *EPC* and *CacheErr* registers are updated with the correct virtual address pointing to the offending load instruction, and the exception handler can still determine the cause of exception and its origin. The program can be restarted by rewriting the destination register -- not automatically, however, as in the case of all the other precise exceptions where no state change occurs.

5.3 Exception Processing Registers

This section describes the CP0 registers that are used in exception processing. Table 5-1 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 5-1 are used in exception processing, and are described in the sections that follow.

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

Table 5-1 CP0 Exception Processing Registers

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 5-1 shows the format of the *Context* register; Table 5-2 describes the *Context* register fields.

Context Register

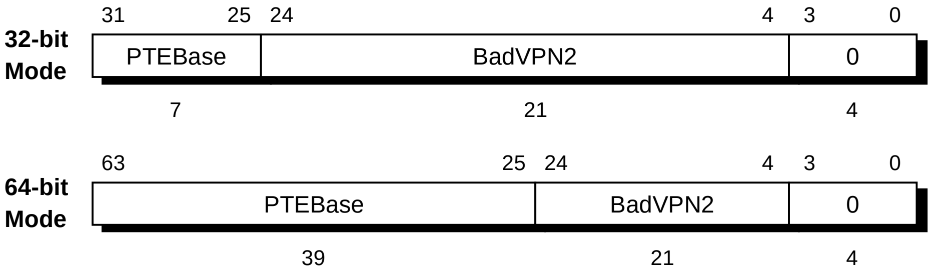


Figure 5-1   Context Register Format

Field	Description
BadVPN2	This field is written by hardware on a miss. It contains the virtual page number (VPN) of the most recent virtual address that did not have a valid translation.
PTEBase	This field is a read/write field for use by the operating system. It is normally written with a value that allows the operating system to use the Context register as a pointer into the current PTE array in memory.
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 5-2   Context Register Fields

The 21-bit *BadVPN2* field contains bits 31:11 of the virtual address that caused the TLB miss; bit 10 is excluded because a single TLB entry maps to an even-odd page pair. For a 1-Kbyte page size, this format can directly address the pair-table of 8-byte PTEs. For other page sizes, 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 5-2 shows the format of the *BadVAddr* register.

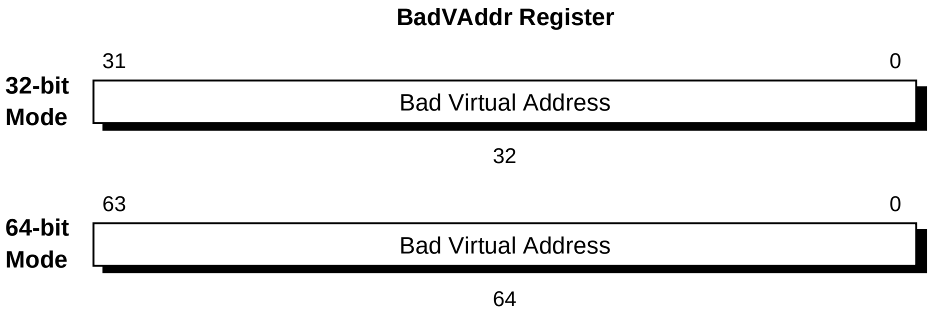


Figure 5-2   BadVAddr Register Format

Note: The *BadVAddr* register does not save any information for bus errors, since bus errors are not addressing errors.

Count Register (9)

The read/write *Count* register acts as a timer, incrementing at a constant rate -- same the MasterOut 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 5-3 shows the format of the *Count* register.

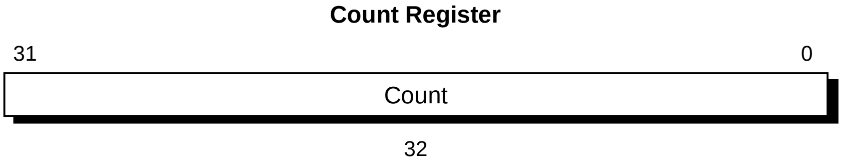


Figure 5-3 Count Register Format

Compare Register (11)

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

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

Writing a value to the *Compare* register, as a side effect, clears the timer interrupt.

For diagnostic purposes, the *Compare* register is a read/write register. Figure 5-4 shows the format of the *Compare* register.

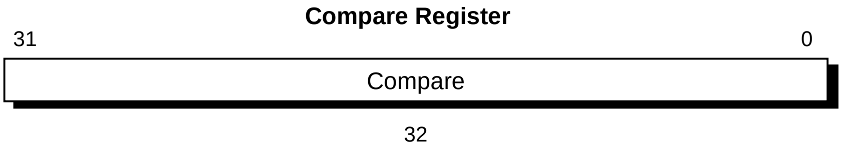


Figure 5-4 Compare Register Format

Status Register (12)

The *Status* register (SR) is a read/write register that contains the operating mode, interrupt enabling, and the diagnostic states of the processor. The following list describes the more important *Status* register fields; Figure 5-5 and Figure 5-6 show the format of the entire register, including descriptions of the fields. Some of the important fields include:

- The 8-bit *Interrupt Mask* (*IM*) field controls the enabling of 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.
- Regardless of the *CU0* bit setting, CP0 is always usable in Kernel mode.
- The 9-bit *Diagnostic Status* (*DS*) field is used for self-testing, and checks the cache and virtual memory system.
- The *Reverse-Endian (RE)* bit, bit 25, reverses the endianness of the machine for the user task. The system can be configured as either little-endian or big-endian at system reset by the BigEndian pin, as follows:
  - when RE = 1, kernel endianness is specified by the BigEndian pin and user endianness is the opposite of the kernel
  - when RE = 0, both kernel and user endianness are specified by the BigEndian pin.

Status Register Format

Figure 5-5 shows the format of the *Status* register. Table 5-3 describes the *Status* register fields. Figure 5-6 and Table 5-4 provide additional information on the *Diagnostic Status* (*DS*) field. All bits in the *DS* field except *TS* are readable and writable.

Status Register

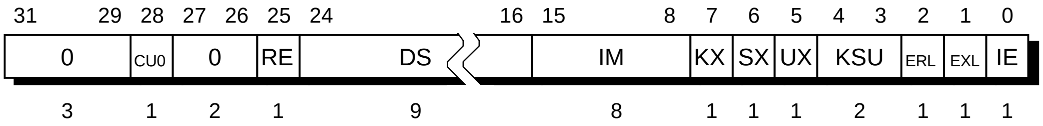


Figure 5-5 Status Register

Field	Description
CU0	Controls the usability of the coprocessor unit. CP0 is always usable when in Kernel mode, regardless of the setting of the CU0 bit. 1 -> usable 0 -> unusable
0	Reserved. Set to 0.
RE	Reserve-Endian bit, valid in User mode.
DS	Diagnostic Status field (See Figure 5-6 and Table 5-4).
IM	Interrupt Mask: controls the enabling of each of the external, internal, and software interrupts. An interrupt is taken if interrupts are enabled, and the corresponding bits are set in both the Interrupt Mask field of the Status register and the Interrupt Pending field of the Cause register. IM[7] correspond to Timer Interrupt, IM[6:2] to interrupts Int[4:0] and IM[1:0] to the software interrupts. 0 -> disabled 1 -> enabled
KX	KX controls whether the TLB Refill Vector or the XTLB Refill Vector address is used for TLB misses on kernel addresses. 0 -> TLB Refill Vector 1 -> XTLB Refill Vector
SX	Enables 64-bit virtual 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 5-3 Status Register Fields

Field	Description
UX	Enables 64-bit virtual 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 0 -> normal 1 -> error
EXL	Exception Level 0 -> normal 1 -> exception
IE	Interrupt Enable 0 -> disables interrupts 1 -> enables interrupts

Table 5-3 (cont.)   Status Register Fields

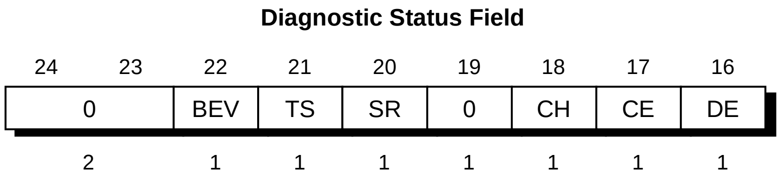


Figure 5-6 Status Register DS Field

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); used to avoid damage to the TLB if more than one TLB entry matches a single virtual address. After TLB shutdown, the processor must be reset to restore the TLB. 0 -> TLB shutdown has not occurred 1 -> TLB shutdown has occurred
SR	1 -> Indicates a soft reset or NMI has occurred.
CH	CP0 condition bit. Read/write access by software only; not accessible to hardware. 0 -> False 1 -> True
CE	Contents of the PErr register set or modify the check bits of the caches when CE = 1; see description of the PErr register.
DE	Specifies that cache parity errors cannot cause exceptions. 0 -> parity remains enabled 1 -> disables parity
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 5-4 Status Register Diagnostic Status Bits

Status Register Reset

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

- *TS* = 0, SR = 0
- *ERL* and *BEV* = 1

The *SR* bit distinguishes between a Cold Reset and Soft Reset.

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 4 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.

Cause Register (13)

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

Figure 5-7 shows the fields of this register; Table 5-5 describes the *Cause* register fields. A 5-bit exception code (*ExcCode*) indicates one of the causes, as listed in Table 5-6.

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

Field	Description
BD	Indicates whether the last exception taken occurred in a branch delay slot. 1 -> delay slot 0 -> normal
CE	Coprocessor unit number referenced when a Coprocessor Unusable exception is taken.
IP	Indicates an interrupt is pending 1 -> interrupt pending 0 -> no interrupt
ExcCode	Exception code field (see Table 5-6)
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 5-5 Cause Register Fields

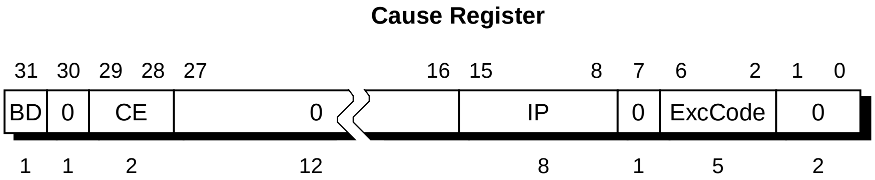


Figure 5-7 Cause Register Fearmat

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

Table 5-6 Cause Register ExcCode Field

Exception Program Counter (EPC) Register (14)

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

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 5-8 shows the format of the *EPC* register.

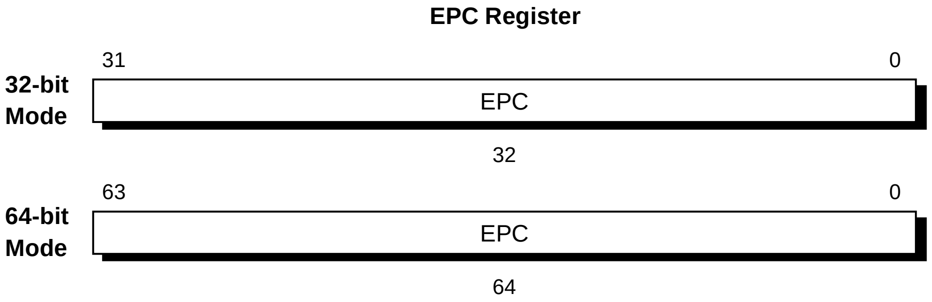


Figure 5-8 EPC Register Format

WatchLo (18) and WatchHi (19) Registers

The VR4100 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 5-9 shows the format of the *WatchLo* and *WatchHi* registers; Table 5-7 describes the *WatchLo* and *WatchHi* register fields.

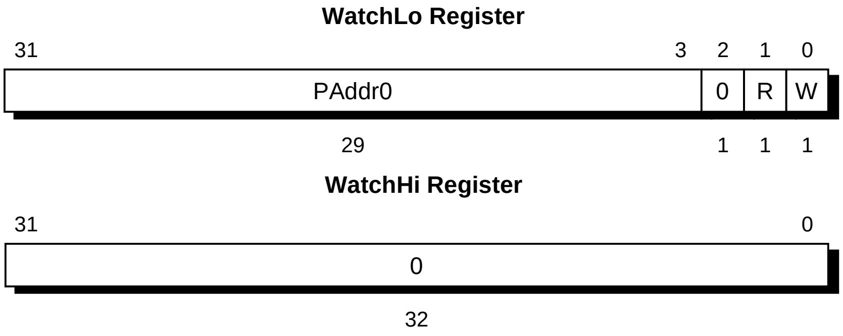


Figure 5-9   WatchLo and WatchHi Register Formats

Field	Description
PAddr0	Bit 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.

Table 5-7   WatchHi and WatchLo Register Fields

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 5-10 shows the format of the *XContext* register; Table 5-8 describes the *XContext* register fields.

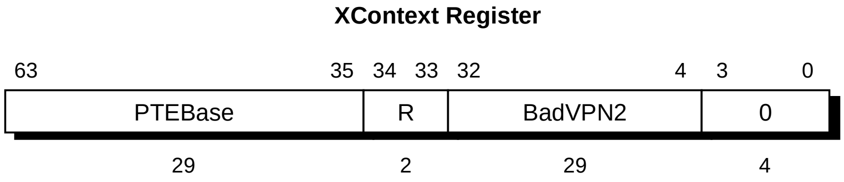


Figure 5-10   XContext Register Format

The 29-bit *BadVPN2* field has bits 39:11 of the virtual address that caused the TLB miss; bit 10 is excluded because a single TLB entry maps to an even-odd page pair. For a 1-Kbyte page size, this format may be used directly to address the pair-table of 8-byte PTEs. For more than 4K byte page and PTE sizes, shifting and masking this value produces the appropriate address.

Field	Description
BadVPN2	The Bad Virtual Page Number/2 field contains the VPN of the most recent invalidly translated virtual address, divided by 2.
R	The Region field contains bits 63:62 of the virtual address. 00_2 = user 01_2 = supervisor 11_2 = kernel
PTEBase	The Page Table Entry Base read/write field indicates the base address of the PTE in the current user address space.

Table 5-8   XContext Register Fields

Parity Error (PErr) Register (26)

The read/write *PErr* register contains the cache data parity bits for cache initialization, cache diagnostics, or cache error processing.

The *PErr* register is loaded by the Index Load Tag CACHE operation. All bit of the parity field are valid on the data cache operation. But a LSB of the parity field is valid on the instruction cache operation. The contents of the *PErr* register are:

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

Figure 5-11 shows the format of the *PErr* register; Table 5-9 describes the register fields.

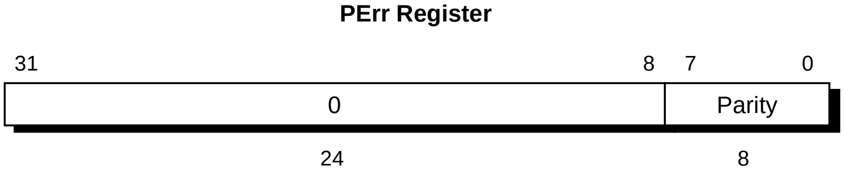


Figure 5-11   PErr Register Format

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

Table 5-9   PErr Register Fields

Cache Error (CacheErr) Register (27)

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

The *CacheErr* register holds cache index and status bits that indicate the source and nature of the error; it is loaded when a Cache Error exception is asserted.

Figure 5-12 shows the format of the *CacheErr* register and Table 5-10 describes the *CacheErr* register fields.

CacheErr Register

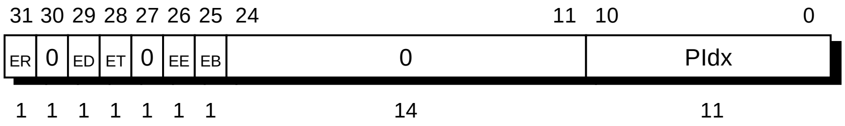


Figure 5-12 CacheErr Register Format

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

Table 5-10 CacheErr Register Fields

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

★ The *ErrorEPC* register is similar to the *EPC* register, except that *ErrorEPC* is used 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.

Figure 5-13 shows the format of the *ErrorEPC* register.

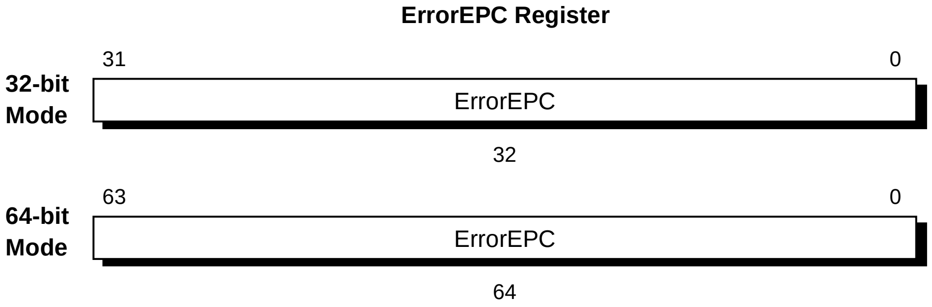


Figure 5-13   ErrorEPC Register Format

5.4 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)
- cache error
- remaining processor exceptions

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

When the processor takes an exception, the *EXL* bit is set to 1, meaning the system is in Kernel mode. After saving the appropriate state, the exception handler typically resets the *EXL* bit back to 0. While restoring the state before restarting, the handler sets the *EXL* bit back to 1.

Returning from an exception also resets the *EXL* bit to 0 (see the ERET instruction in Chapter 14).

Exception Vector Locations

The Cold Reset, Soft Reset, and NMI exceptions are always vectored to:

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

Addresses for the remaining exceptions are a combination of a *vector offset* and a *base address*.

E.g. TLB Refill vector: When *BEV* = 0, the vector base for the TLB Refill exception is in *kseg0* (cached, unmapped space) (0x8000 0000 in 32-bit mode, 0xFFFF FFFF 8000 0000 in 64-bit mode).

When *BEV* = 1, the vector base for the TLB Refill exception is in *kseg1* (uncached, unmapped space) 0xBFC0 0200 in 32-bit mode and 0xFFFF FFFF BFC0 0200 in 64-bit mode. This is an uncached and unmapped space, allowing the exception to bypass the cache and TLB.

E.g. Cache Error vector: When *BEV* = 0, the vector base for the Cache Error exception is in *kseg1* (uncached , unmapped space) (0xA000 0000 in 32-bit mode, 0xFFFF FFFF A000 0000 in 64-bit mode).

When *BEV* = 1, the vector base for the Cache Error exception is in kseg1 (uncached, unmapped space) 0xBFC0 0200 in 32-bit mode and 0xFFFF FFFF BFC0 0200 in 64-bit mode.

Unlike the other exception vectors, the Cache Error exception vector must always lie in uncached space.

64-bit mode exception vectors and their offsets are shown in Table 5-11.

Exception	Vector Base	Vector Offset
Cold Reset, Soft Reset, and NMI	0xFFFF FFFF BFC0 0000	0x0000
Cache Error	0xFFFF FFFF A000 0000 (BEV = 0) 0xFFFF FFFF BFC0 0200 (BEV = 1)	0x0100
TLB Refill, EXL = 0	0xFFFF FFFF 8000 0000 (BEV = 0) 0xFFFF FFFF BFC0 0200 (BEV = 1)	0x0000
XTLB Refill, EXL = 0	0xFFFF FFFF 8000 0000 (BEV = 0) 0xFFFF FFFF BFC0 0200 (BEV = 1)	0x0080
Other	0xFFFF FFFF 8000 0000 (BEV = 0) 0xFFFF FFFF BFC0 0200 (BEV = 1)	0x0180

Table 5-11 64-Bit Mode Exception Vector Base Addresses

Priority of Exceptions

The remainder of this chapter describes exceptions in the order of their priority shown in Table 5-12 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.

Cold Reset (highest priority)
Soft Reset
Nonmaskable Interrupt (NMI)
Address error-- Instruction fetch
TLB/XTLB refill -- Instruction fetch
TLB invalid -- Instruction fetch
Cache error -- Instruction fetch
Bus error -- Instruction fetch
System Call
Breakpoint
Coprocessor Unusable
Reserved Instruction
Trap
Integer overflow
Address error -- Data access
TLB/XTLB refill -- Data access
TLB invalid -- Data access
TLB modified -- Data write
Cache error -- Data access
Watch
Bus error -- Data access
Interrupt (lowest priority)

Table 5-12   Exception Priority Order

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 $\overline{\textbf{ColdReset}}$[†] signal is asserted and then deasserted ($\overline{\textbf{Reset}}$ must be asserted along with $\overline{\textbf{ColdReset}}$). 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*, and *TS* of the *Status* register are cleared to 0
- *ERL* and *BEV* of the *Status* register are set to 1.
- The *Random* register is initialized to the value of its upper bound.
- The *Wired* register is initialized to 0.
- ★ *EC* field and bit 22:3 of the *Config* register are set.

All other bits are undefined.

Servicing

The Cold Reset exception is serviced by:

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

[†] In the following sections -- indeed, throughout this book -- a signal with a bar, such as $\overline{\textbf{ColdReset}}$, is low active.

Soft Reset Exception

Cause

A Soft Reset (sometimes called Warm Reset) occurs when the $\overline{\textbf{ColdReset}}$ signal remains deasserted while the $\overline{\textbf{Reset}}$ pin goes from assertion to deassertion. For a Soft Reset to be valid there must be at least one cycle during which neither $\overline{\textbf{Reset}}$ nor $\overline{\textbf{ColdReset}}$ were asserted.

A Soft Reset immediately resets all state machines, and sets the *SR* bit of the *Status* Register.

Execution begins at the reset vector when the reset is deasserted. This exception is not maskable.

Processing

The CPU provides a special interrupt vector for this exception (same location as Cold Reset):

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

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

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

- *ErrorEPC* register, which contains the restart PC
- *TS* of the *Status* register are cleared to 0
- *ERL*, *SR*, and *BEV* of the *Status* register are set to 1.

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

Servicing

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

Nonmaskable Interrupt (NMI) Exception

Cause

The Nonmaskable Interrupt (NMI) exception occurs in response to the falling edge of the $\overline{\textbf{NMI}}$ pin. An NMI can also be set by an external write through the **SysAD** bus.

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

Processing

The CPU provides a special interrupt vector for this exception (same location as Cold Reset):

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

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

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

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

- *ErrorEPC* register, which contains the restart PC
- *TS* of the *Status* register are cleared to 0
- *ERL*, *SR*, and *BEV* of the *Status* register are set to 1.

Servicing

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

Address Error Exception

Cause

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

- load or store a doubleword that is not aligned on a doubleword boundary
- load 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
- branch to an address that is not aligned on a word boundary . ★

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 (*AdE*S) 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 was referenced in 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 E*ntryLo0/EntryLo1* register; the *EntryHi* and *EntryLo* registers are written into the TLB.

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

TLB Invalid Exception

Cause

The TLB Invalid exception occurs when a virtual address reference matches a TLB entry that is marked invalid (TLB valid bit is 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

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.

Cache Error Exception

Cause

The Cache Error exception occurs when either a primary cache parity error or a System bus parity error is detected. This exception is not maskable, but error detection may be disabled by the *DE* bit of the *Status* register.

If a parity error is detected when the *DE* bit of *Status* register is not set, a cache error exception is taken during one of the following operations:

- an instruction fetch from instruction cache
- a load from the data cache
- tag parity check on a store
- main memory read by the processor
- most of the CACHE ops (no exception is taken for the CACHE ops Index Load Tag or Index Store Tag)

In case of a parity or bus error on the second double word returned from memory for data cache, the cache error exception is not taken and the cache line is marked invalid. This keeps the exceptions precise, since the processor pipeline may have advanced after receipt of the critical word.

Processing

The processor sets the *ERL* bit in the *Status* register, saves the exception restart address in *ErrorEPC* register, and then transfers to a special vector in uncached space.

If the BEV bit = 0, the vector is one of the following: 0xA000 0100 in 32-bit mode, or 0xFFFF FFFF A000 0100 in 64-bit mode

If the BEV bit = 1, the vector is one of the following: 0xBFC0 0300 in 32-bit mode, or 0xFFFF FFFF BFC0 0300 in 64-bit mode

No other registers are changed.

Servicing

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

Bus Error Exception

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 Cause 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 Cause 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 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.

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 load or store 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 instruction which causes 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

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.

5.5 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
- cache error exception and its 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).

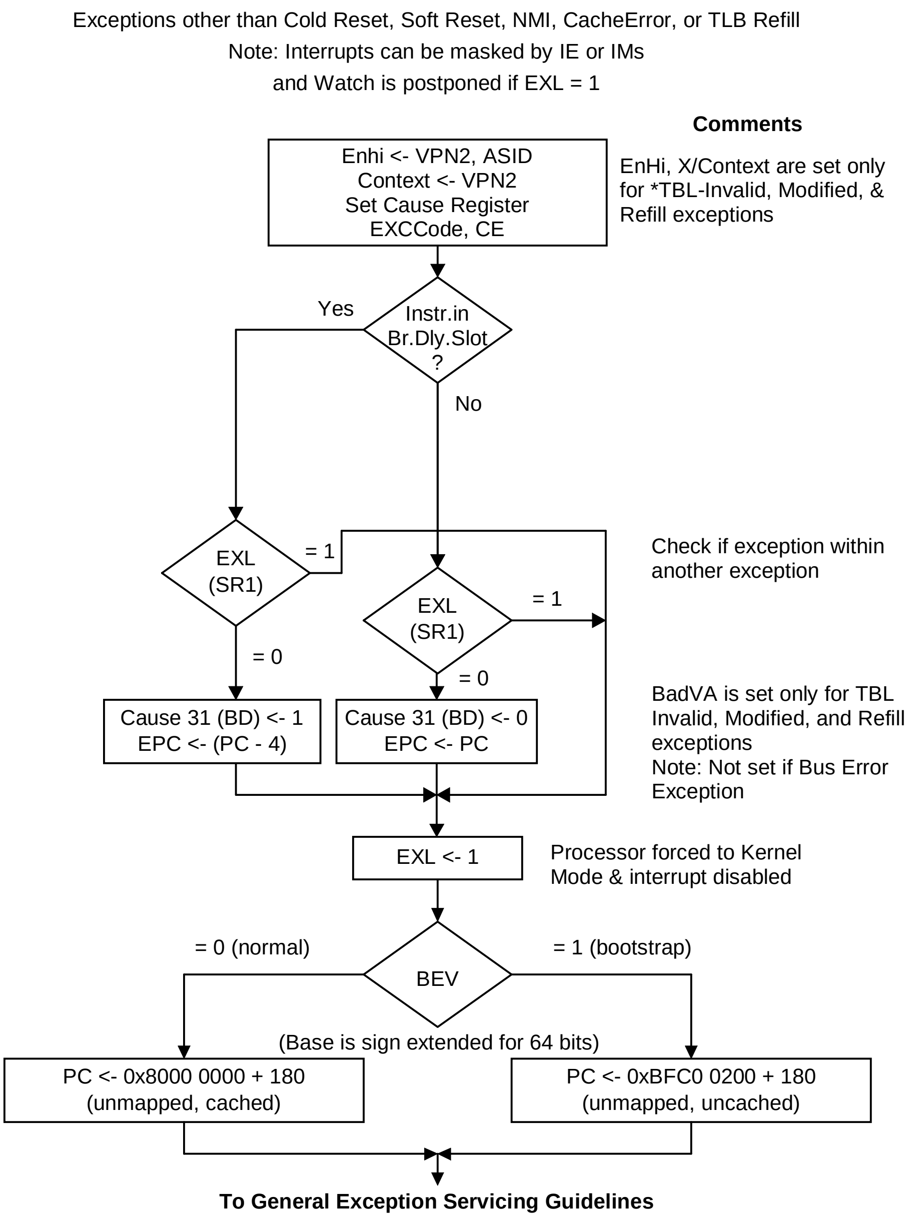

Exceptions other than Cold Reset, Soft Reset, NMI, CacheError, or TLB Refill
Note: Interrupts can be masked by IE or IMs
and Watch is postponed if EXL = 1
Comments
Enhi <- VPN2, ASID
Context <- VPN2
Set Cause Register
EXCCode, CE
EnHi, X/Context are set only for *TBL-Invalid, Modified, & Refill exceptions
Yes
Instr.in Br.Dly.Slot ?
No
EXL (SR1)
= 1
Check if exception within another exception
EXL (SR1)
= 1
= 0
= 0
Cause 31 (BD) <- 1
EPC <- (PC - 4)
Cause 31 (BD) <- 0
EPC <- PC
BadVA is set only for TBL Invalid, Modified, and Refill exceptions
Note: Not set if Bus Error Exception
EXL <- 1
Processor forced to Kernel Mode & interrupt disabled
= 0 (normal)
BEV
= 1 (bootstrap)
(Base is sign extended for 64 bits)
PC <- 0x8000 0000 + 180
(unmapped, cached)
PC <- 0xBFC0 0200 + 180
(unmapped, uncached)
To General Exception Servicing Guidelines

Figure 5-14 General Exception Handler (HW)

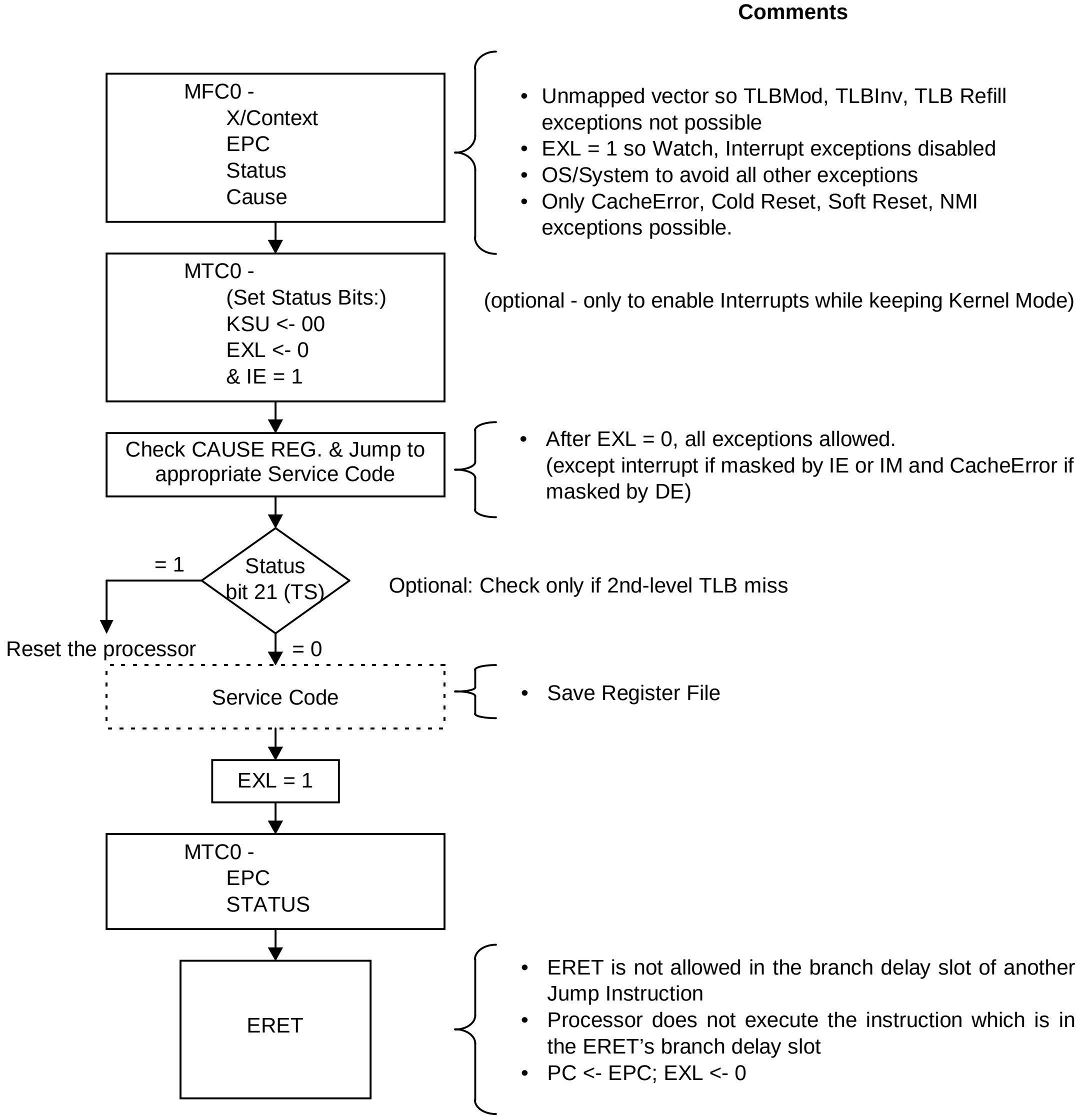

Comments
MFC0 -
X/Context
EPC
Status
Cause
• Unmapped vector so TLBMod, TLBInv, TLB Refill exceptions not possible
• EXL = 1 so Watch, Interrupt exceptions disabled
• OS/System to avoid all other exceptions
• Only CacheError, Cold Reset, Soft Reset, NMI exceptions possible.
MTC0 -
(Set Status Bits:)
KSU <- 00
EXL <- 0
& IE = 1
(optional - only to enable Interrupts while keeping Kernel Mode)
Check CAUSE REG. & Jump to appropriate Service Code
• After EXL = 0, all exceptions allowed. (except interrupt if masked by IE or IM and CacheError if masked by DE)
= 1
Status bit 21 (TS)
Optional: Check only if 2nd-level TLB miss
Reset the processor
= 0
Service Code
• Save Register File
EXL = 1
MTC0 -
EPC
STATUS
ERET
• ERET is not allowed in the branch delay slot of another Jump Instruction
• Processor does not execute the instruction which is in the ERET's branch delay slot
• PC <- EPC; EXL <- 0

Figure 5-15 General Exception Servicing Guidelines (SW)

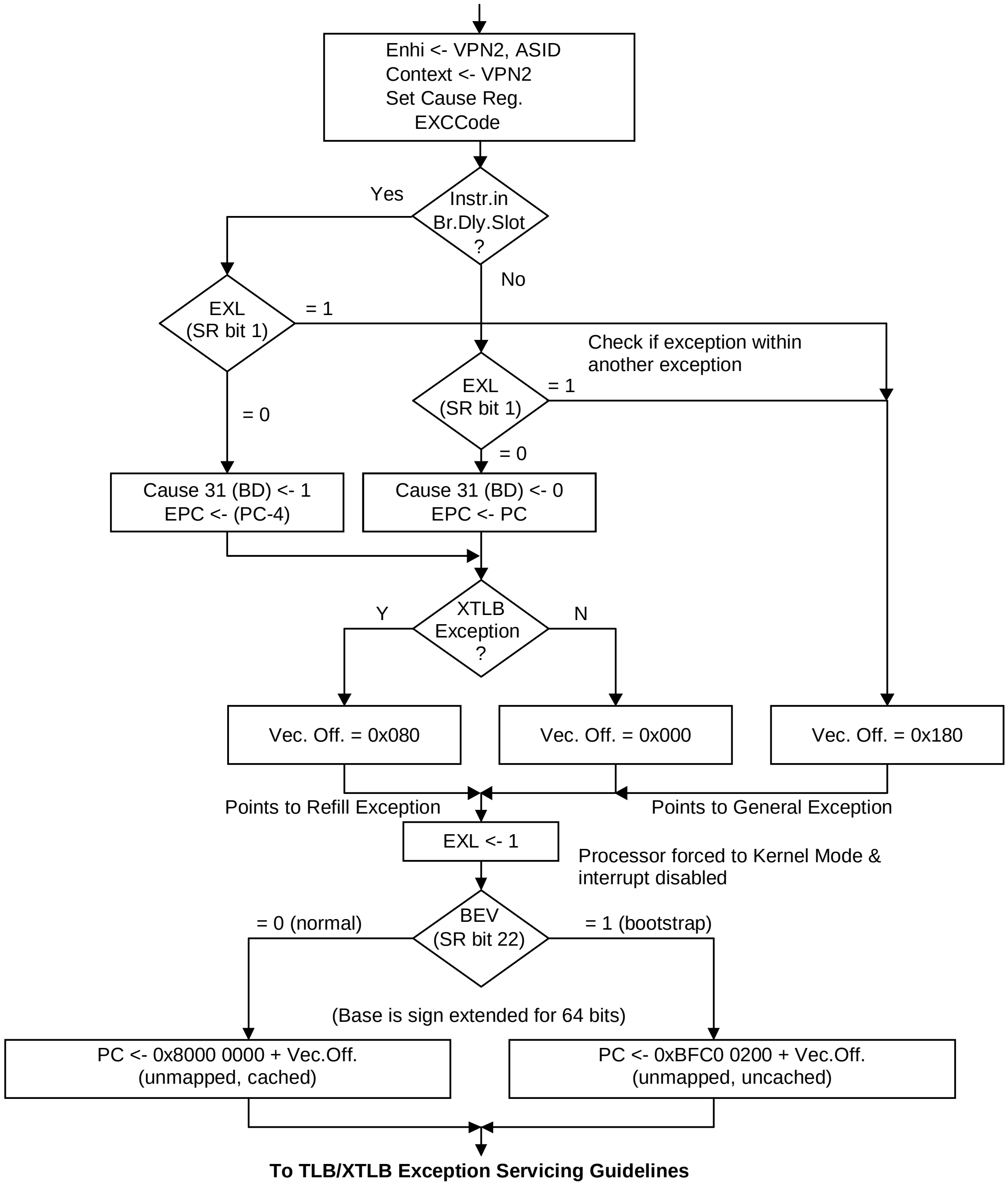
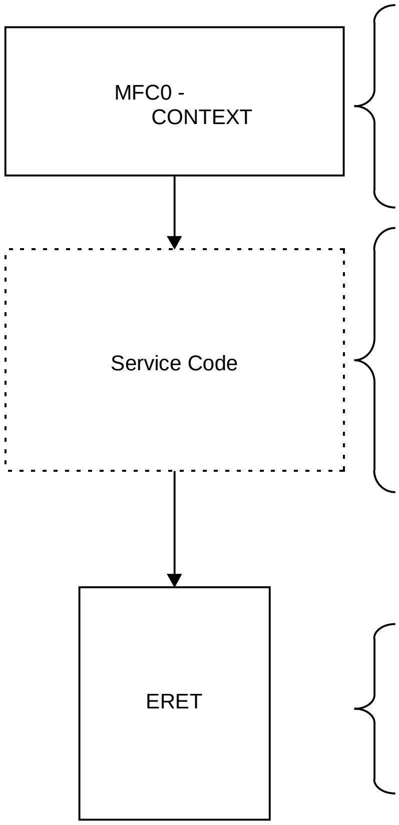


Figure 5-16 TLB/XTLB Miss Exception Handler (HW)



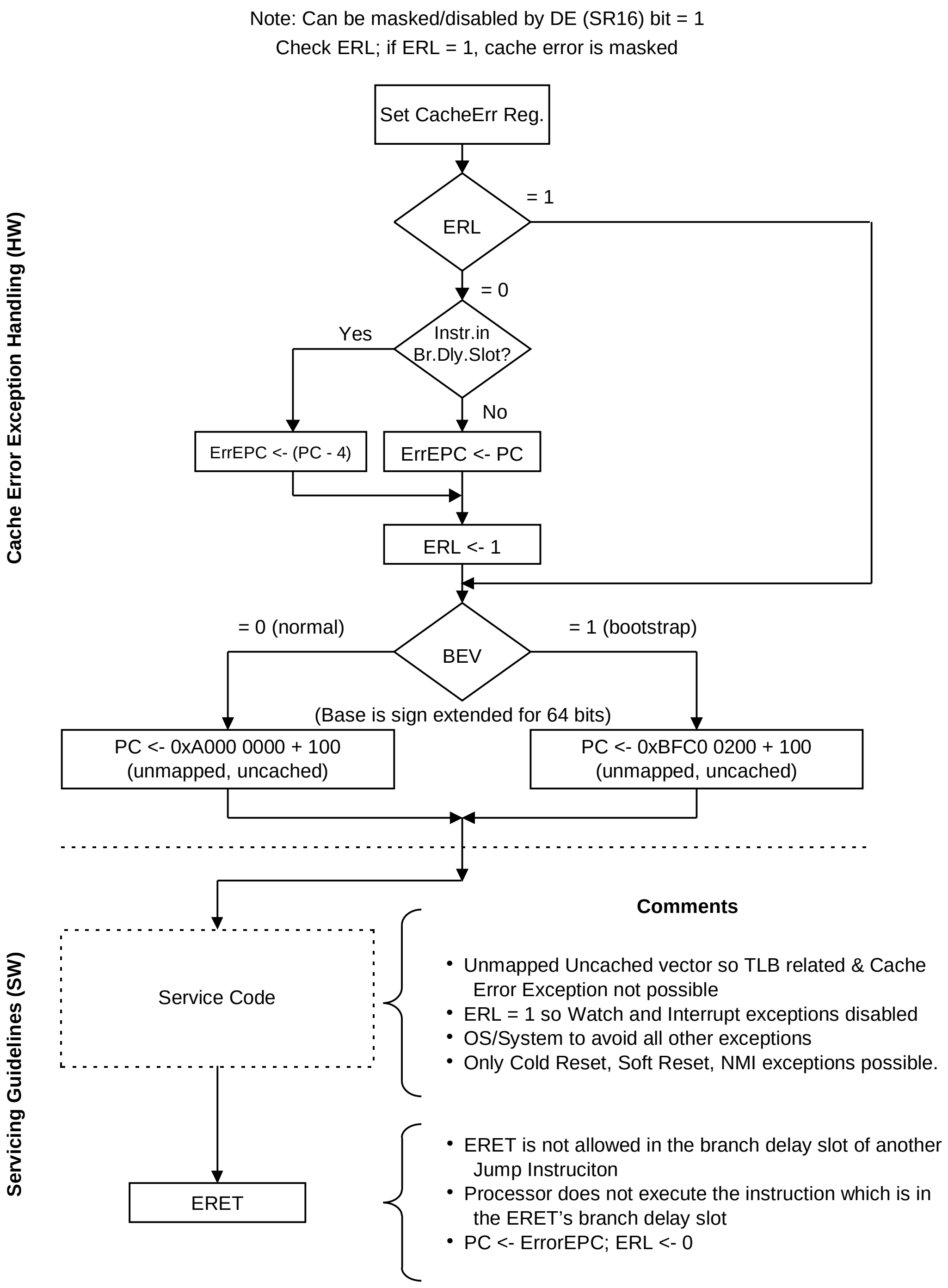
Comments

- Unmapped vector so TLBMod, TLBInv, TLB Refill exceptions not possible
- EXL = 1 so Watch, Interrupt exceptions disabled
- OS/System to avoid all other exceptions
- Only CacheError, Cold Reset, Soft Reset, NMI exceptions possible.

- Load the mapping of the virtual address in X/Context Reg. Move it to ENLO and Write into the TLB
- There could be a TLB miss again during the mapping of the data or instruction address. The processor will jump to the general exception vector since the EXL is 1. (Option to complete the first level refill in the general exception handler or ERET to the original instruction and take the exception again)

- ERET is not allowed in the branch delay slot of another Jump Instruction
- Processor does not execute the instruction which is in the ERET’s branch delay slot
- PC <- EPC; EXL <- 0

Figure 5-17   TLB/XTLB Exception Servicing Guidelines (SW)


Note: Can be masked/disabled by DE (SR16) bit = 1
Check ERL; if ERL = 1, cache error is masked

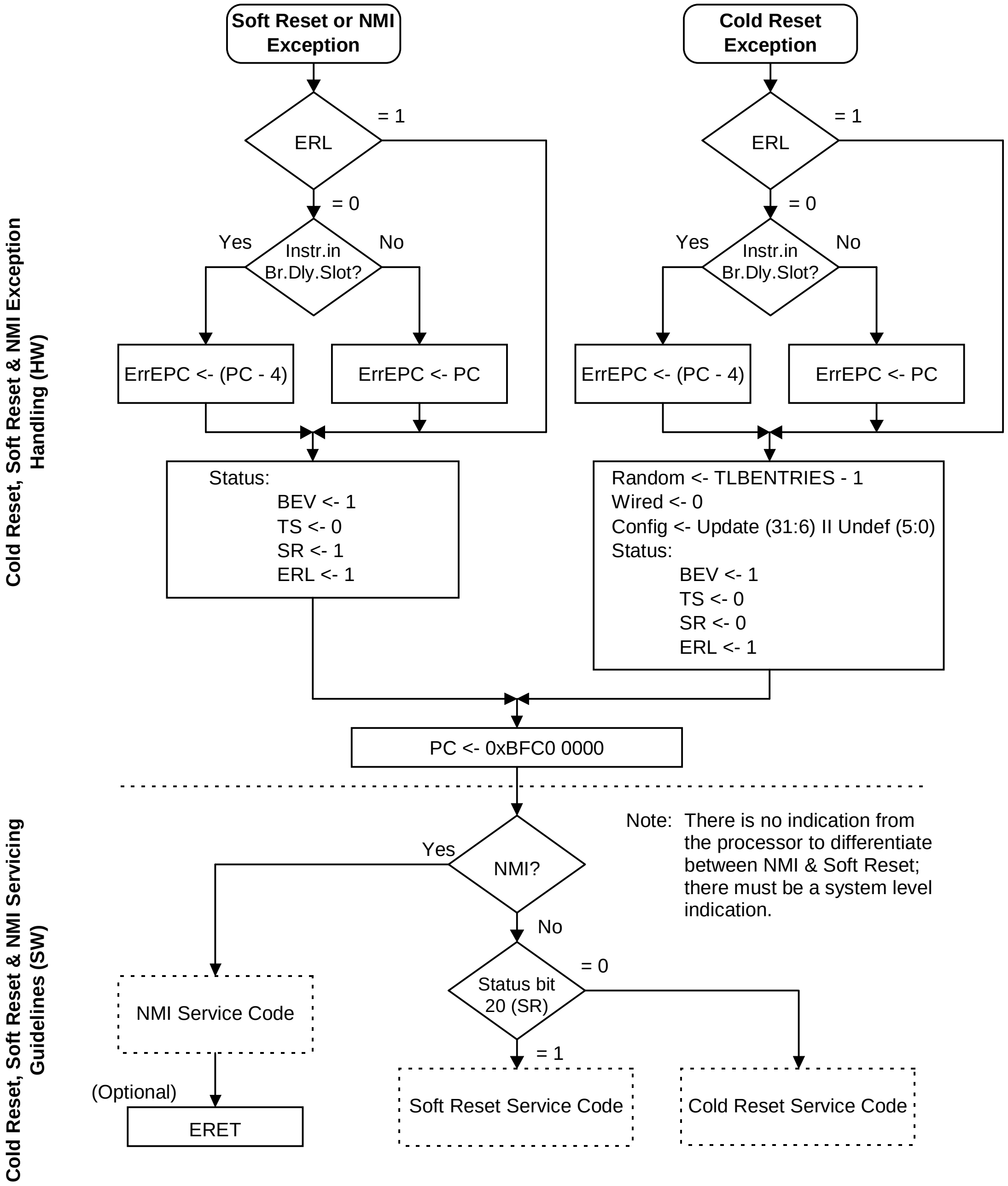
Cache Error Exception Handling (HW)
Set CacheErr Reg.
ERL
= 1
= 0
Instr.in Br.Dly.Slot?
Yes
No
ErrEPC <- (PC - 4)
ErrEPC <- PC
ERL <- 1
BEV
= 0 (normal)
= 1 (bootstrap)
(Base is sign extended for 64 bits)
PC <- 0xA000 0000 + 100 (unmapped, uncached)
PC <- 0xBFC0 0200 + 100 (unmapped, uncached)
Servicing Guidelines (SW)
Service Code
ERET

Comments

- Unmapped Uncached vector so TLB related & Cache Error Exception not possible
- ERL = 1 so Watch and Interrupt exceptions disabled
- OS/System to avoid all other exceptions
- Only Cold Reset, Soft Reset, NMI exceptions possible.

- ERET is not allowed in the branch delay slot of another Jump Instruciton
- Processor does not execute the instruction which is in the ERET’s branch delay slot
- PC <- ErrorEPC; ERL <- 0

Figure 5-18 Cache Error Exception Handling (HW) and Servicing Guidelines (SW)



Note: There is no indication from the processor to differentiate between NMI & Soft Reset; there must be a system level indication.

Figure 5-19 Cold Reset, Soft Reset & NMI Exception Handling (HW) and Servicing Guidelines (SW)

VR4100 Processor Signal Descriptions

6

This chapter describes the signals used by and in conjunction with the VR4100 processor. The signals include the System interface, the Clock/Control interface, the Interrupt interface, and the Initialization interface.

Signals are listed in bold, and low active signals have a trailing asterisk -- for instance, the low-active External Request signal is $\overline{\mathbf{EReq}}$. The signal description also tells if the signal is an input (the processor receives it) or output (the processor sends it out).

Figure 6-1 illustrates the functional groupings of the processor signals.

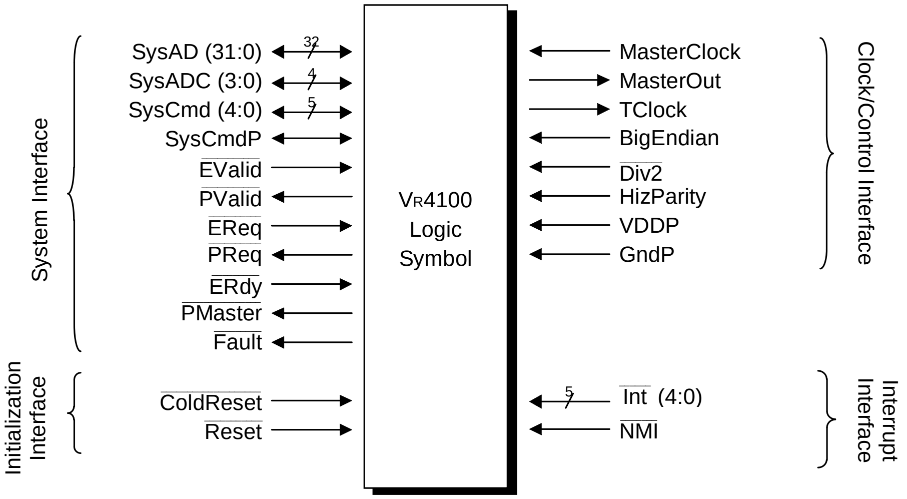


Figure 6-1   VR4100 Processor Signals

6.1 VR4100 Signals

System Interface Signals

System interface signals provide the connection between the VR4100 processor and the other components in the system.

Table 6-1 lists the system interface signals.

Name	Definition	Direction	Description
SysAD (31:0)	System address/data bus	In/Out	A 32-bit address and data bus for communication between the processor and an external agent
SysADC (3:0)	System address/data check bus	In/Out	A 4-bit bus containing check bits for the SysAD bus (one even-parity bit per byte)
SysCmd (4:0)	System command/data identifier	In/Out	A 5-bit bus for command and data identifier transmission between the processor and an external agent
SysCmdP	System command/data identifier bus parity	In/Out	A single, even-parity bit for the SysCmd bus
$\overline{\text{EValid}}$	External agent Valid	In	Signal that the 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 during this cycle
$\overline{\text{PValid}}$	Processor Valid	Out	Signal that 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 during this cycle
$\overline{\text{EReq}}$	External Request	In	Signal that the external agent requests system interface bus ownership
$\overline{\text{PReq}}$	Processor Request	Out	Signal that the processor requests system interface bus ownership

Table 6-1 System Interface Signals

Name	Definition	Direction	Description
$\overline{\text{PMaster}}$	Processor Master	Out	Signal that the processor is the master of the system interface bus
$\overline{\text{ERdy}}$	External Ready	In	Signal that an external agent is capable of accepting a processor request
$\overline{\text{Fault}}$	Fault	Out	Signal that the SysCmd bus parity error occurred

Table 6-1 (cont.) System Interface Signals

Clock/Control Interface Signals

The Clock/Control interface signals make up the interface for clocking and maintenance. Table 6-2 lists the Clock/Control interface signals.

Name	Definition	Direction	Description
TClock	Transmit clock	Out	Transmit clock at the operational frequency of the System interface
MasterClock	Master clock	In	Master clock input that establishes the processor operating frequency
MasterOut	MasterClock Out	Out	Master clock output aligned with MasterClock
BigEndian	BigEndian byte order	In	Indicates the System interface byte ordering of data
$\overline{\text{Div2}}$	Divide frequency	In	Assertion of $\overline{\text{Div2}}$ causes the System interface to run at one-half of the PClock. De-assertion of $\overline{\text{Div2}}$ causes the System interface to run at the same frequency of the PClock. ★
HizParity	Hi-Z parity mode	In	When asserted, this signal disconnects parity terminals electrically and disenabled to detect bus parity error. VR4100 generates parity bits for the cache data.
VDDP	Quiet VDD for PLL	In	Quiet V_{DD} for the internal phaselocked loop.
GndP	Quiet Gnd for PLL	In	Quiet Gnd for the internal phaselocked loop.

Table 6-2 Clock/Control Interface Signals

Interrupt Interface Signals

The Interrupt interface signals make up the interface used by external agents to interrupt the VR4100. Table 6-3 lists the Interrupt interface signals.

Name	Definition	Direction	Description
$\overline{\text{Int}}$ [4:0]	Interrupt	In	General processor interrupt
$\overline{\text{NMI}}$	Nonmaskable interrupt	In	Nonmaskable interrupt signal

Table 6-3 Interrupt Interface Signals

Initialization Interface Signals

The Initialization interface signals make up the interface by which an external agent initializes the processor operating parameters. Table 6-4 lists the Initialization interface signals.

Name	Definition	Direction	Description
$\overline{\text{ColdReset}}$	Cold Reset	In	This signal must be asserted for a Cold Reset. The clocks SClock and TClock begin to cycle and are synchronized with the de-asserted edge of ColdReset. $\overline{\text{ColdReset}}$ must be de-asserted synchronously with MasterOut.
$\overline{\text{Reset}}$	Soft Reset	In	This signal must be asserted for any reset sequence. It can be asserted synchronously or asynchronously for a Cold Reset, or synchronously to initiate a Soft Reset. $\overline{\text{Reset}}$ must be de-asserted synchronously with MasterOut.

Table 6-4 Initialization Interface Signals

6.2 Signal Summary

This section shows the pinouts and layouts for the VR4100.

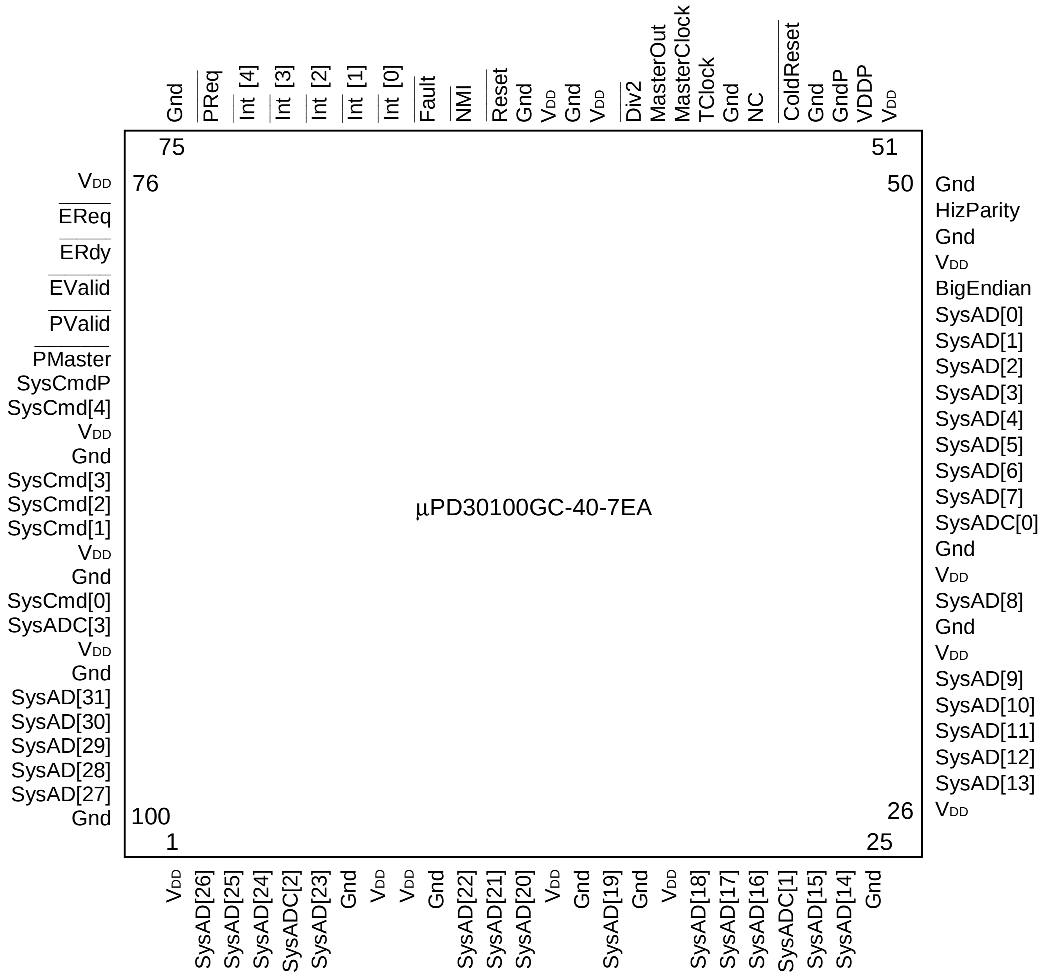


Figure 6-2 VR4100 100-pin TQFP Pinout

[MEMO]

Initialization Interface

7

This chapter describes the VR4100 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 VR4100 processor modes.

Signal names are listed in bold letters -- for instance the signal **MasterClock** indicates the processor clock. Low-active signals are indicated by bar over the signal name, such as $\overline{\textbf{ColdReset}}$ the power-on/Cold Reset signal.

7.1 Functional Overview

The VR4100 processor has the following two types of resets; they use the $\overline{\textbf{ColdReset}}$ and $\overline{\textbf{Reset}}$ 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.
- ★ **Soft Reset** restarts processor, but does not affect clocks. A Soft Reset preserves the processor internal state. The VR4100 processor differs from the R4000 in that only sixteen cycles of $\overline{\textbf{Reset}}$ assertion are required.

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

Care must be taken to coordinate system reset with other system elements. In general, bus or parity 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 VR4100 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 Figure 7-1 and Figure 7-2 later in this chapter for timing diagrams of the Cold and Soft Resets.

7.2 Reset Signal Description

This section describes the two reset signals, $\overline{\textbf{ColdReset}}$ and $\overline{\textbf{Reset}}$. ★

$\overline{\textbf{ColdReset}}$: the $\overline{\textbf{ColdReset}}$ signal must be asserted[†] (low) to reset the processor. The clocks **SClock**, and **TClock** begin to cycle and are synchronized with the deasserted edge (high) of $\overline{\textbf{ColdReset}}$.

$\overline{\textbf{Reset}}$**:** the $\overline{\textbf{Reset}}$ signal is asserted synchronously to initiate a Soft Reset. The $\overline{\textbf{Reset}}$ signal must be deasserted synchronously with **MasterOut**.

Cold Reset

A Cold Reset is used to completely reset the processor, including processor clocks. During a Cold Reset, there is no guarantee of any chip state, except for the following register bits :

- Status register: TS, SR, which are set to zero, and ERL and BEV, which are set to one.
- Random register: initialized with 31.
- Wired register: initialized with 0.
- Config register: bit 31-28, 22-3 are initialized. ★

Once power to the processor is established the $\overline{\textbf{ColdReset}}$ signal is asserted for 64,000 **MasterClock** cycles to ensure time for the processor clocks to lock to the input **MasterClock**.

$\overline{\textbf{Reset}}$ must be asserted whenever $\overline{\textbf{ColdReset}}$ is asserted; $\overline{\textbf{Reset}}$ must remain asserted for 16 cycles after the deassertion of $\overline{\textbf{ColdReset}}$.

BigEndian must be fixed 100 **MasterClock** cycles before the deassertion of $\overline{\textbf{ColdReset}}$ and never changed after the deassertion of $\overline{\textbf{ColdReset}}$

$\overline{\textbf{Div2}}$ and **HizParity** must be fixed before the power on and never changed after the power on.

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

[†] *Asserted* means the signal is true, or in its valid state. For example, the low-active $\overline{\textbf{Reset}}$ 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.

Soft Reset

A Soft Reset is used to reset the processor without affecting the clocks; in other words, a Soft Reset is a logic reset. In a Soft Reset, the processor retains as much state information as possible; all state information except for the following is retained:

- ★ the *Count* register is initialized with 0.
- the *Status* register *BEV* and *SR* bits are set (to 1)
- the *Status* register *TS* bit is cleared (to a 0)
- the *Cause* register *TimerInterrupt* bit is cleared
- any SysAD-generated interrupts are cleared
- NMI is cleared
- *Config* register is initialized

Assertion of the $\overline{\textbf{Reset}}$ signal resets the processor without disrupting the clocks, and allows the processor to retain as much of its state as possible. Since a Soft Reset takes effect immediately upon
★ assertion of the $\overline{\textbf{Reset}}$ signal, multicycle operations such as a cache miss may be aborted with the result of some loss of data.

★ A Soft Reset is started by assertion of the $\overline{\textbf{Reset}}$ pin. $\overline{\textbf{Reset}}$ must be asserted for a minimum of 16 cycles, and must be deasserted synchronously with **MasterOut**. In general, data in the processor is preserved for debugging purposes.

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

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

Figure 7-1 and Figure 7-2 show the timing diagrams for the Cold and Soft Resets.

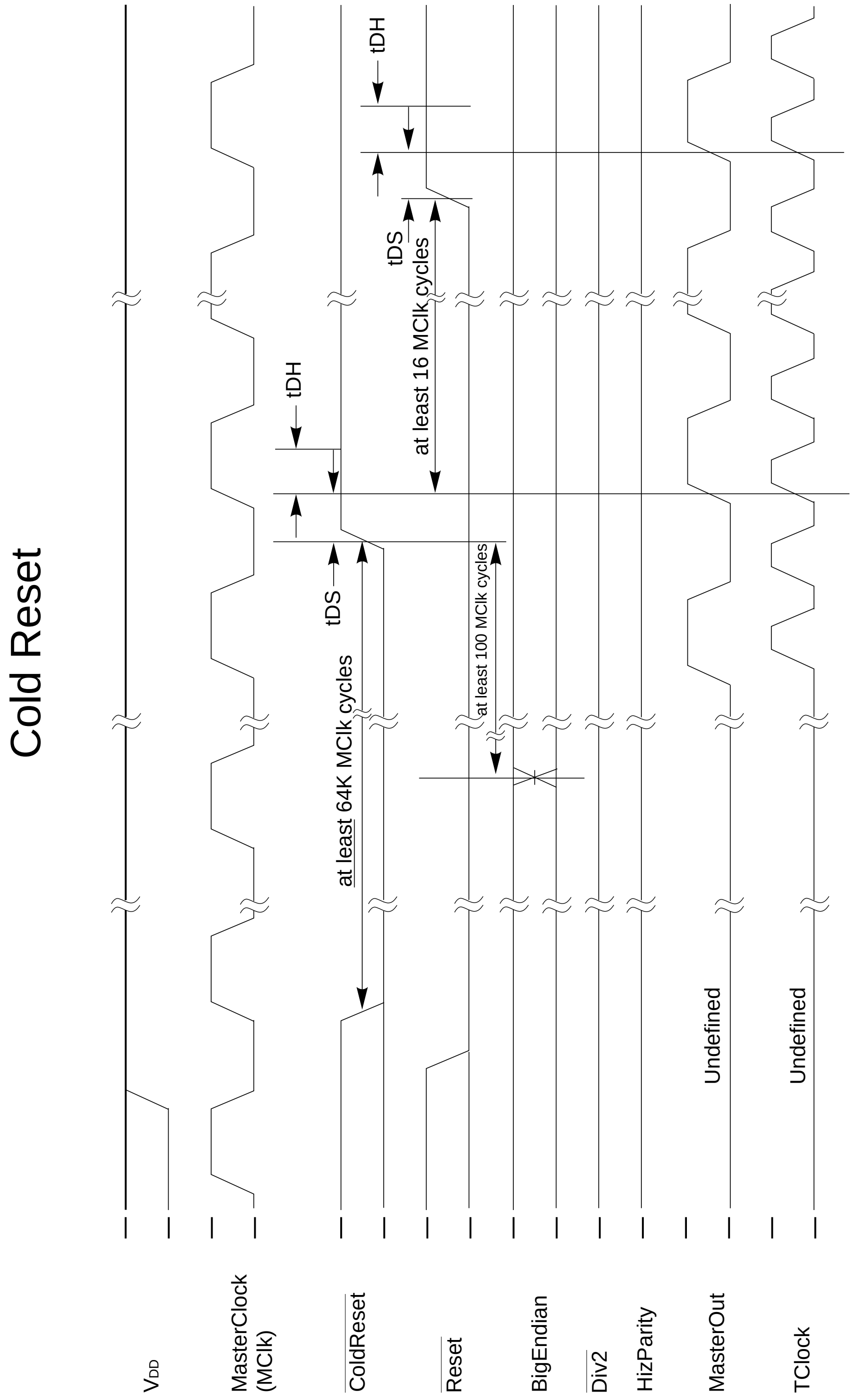


Figure 7-1   Cold Reset

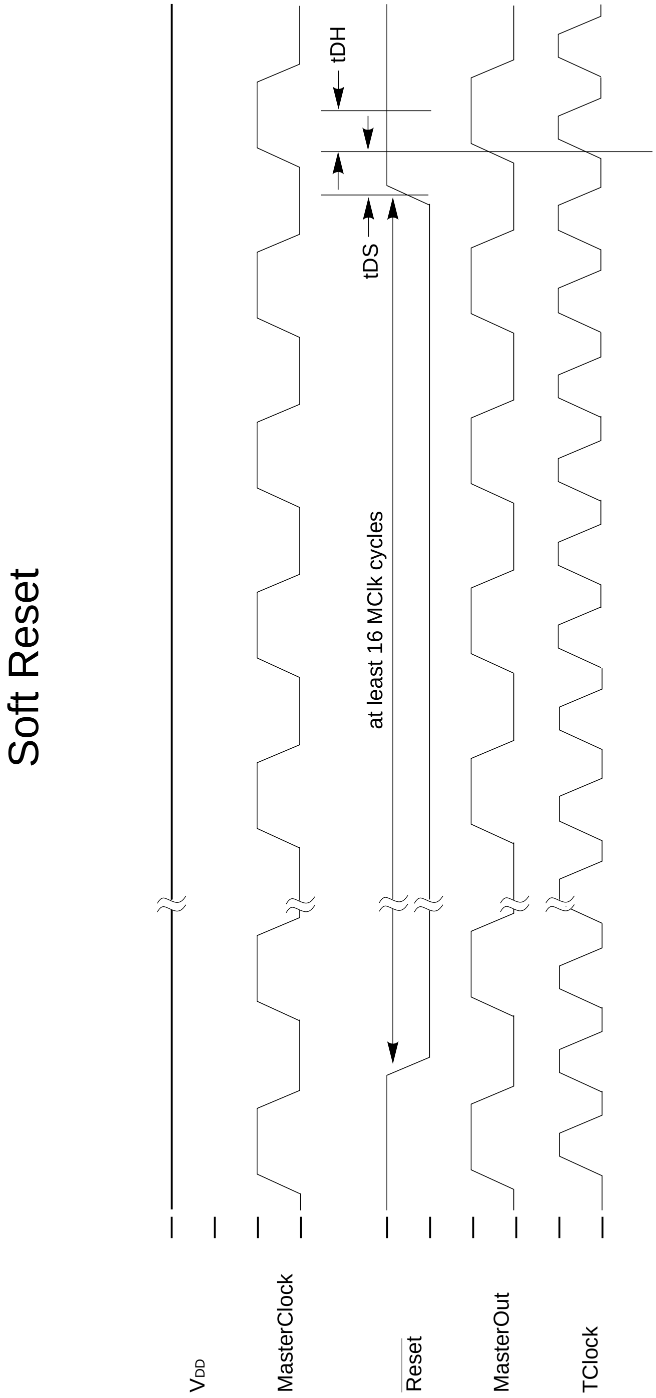


Figure 7-2   Soft Reset

7.3 VR4100 Processor Modes

The VR4100 processor supports several user-selectable modes. All modes except DivMode, and BypassPLL are set/reset by writing to the *Status* register and *Config* register.

Power Modes

The VR4100 supports four power modes: Full Speed, Standby, Suspend and Hibernate mode. This section describes these four modes.

Full Speed Mode

Normally the processor clock (**PClock**) operates at quadruple the **MasterClock** speed, and the System interface clock (**SClock**) operates at half of the **PClock** speed, making it the twice frequency as **MasterClock**.

Default state is normal clocking, and the chip returns to default state after any reset.

Suspend Mode

The users may set the processor to Suspend mode with SUSPEND Instruction. In the Suspend mode, the processor stalls the pipeline, and quits supplying clocks to all of the units except PLL and Interrupt unit. At the time, the contents of the registers and caches are kept, and **TClock** output is stopped. The processor will stay in Suspend mode until an interrupt, NMI, Soft Reset, or Cold Reset is received. On receipt of the interrupt, NMI, Soft Reset, or Cold Reset the processor will enter Full Speed mode.

Hibernate Mode

The users may set the processor to Hibernate mode with HIBERNATE instruction. In the Hibernate mode, the processor quits supplying clocks to all of the units. At the time, the contents of the registers and caches are kept, and **TClock** and **MasterOut** output is stopped. The processor will stay in Hibernate mode until a Cold Reset is received. On receipt of the Cold Reset, the processor will enter Full Speed mode. In this mode, power consumption is 0 W.

Standby Mode

In Standby mode, all internal clocks, except Timer/Interrupt unit, are frozen at hi level.

To enter Standby mode from FullSpeed mode, first execute the STANDBY instruction. When the STANDBY instruction finishes the WB stage, the VR4100 wait by the SysAD bus is idle state, after then the internal clocks will shut down, thus freezing the pipeline. The PLL, Timer/Interrupt clocks and the system interface clocks, **TClock** and **MasterOut**, will continue to run.

Once the VR4100 is in Standby mode, any interrupt, including the internally generated timer interrupt, NMI, Soft Reset, or Cold Reset will cause the VR4100 to exit Standby mode and to enter FullSpeed mode.

Privilege Modes

The VR4100 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.

Reverse Endianess

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

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).

Cache Error Check

The *CE* bit in the *Status* register substitutes the 8-bit field in the *PErr* register for computed parity. On stores, this enables the user to directly write the parity bits in the data cache, rather than having computed parity bits used. The parity bits in the instruction cache can be directly written from the *PErr* register using the Fill CACHE op and setting the *CE* bit. If *CE* is set, the Tag parity bit is written from the TagLo register rather than computed from the tag.

Disable Errors

When the *DE* bit in the *Status* register is set, it specifies that the processor does not take an exception on a cache parity error.

Interrupt Enable

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

[MEMO]

Clock Interface

8

This chapter describes the clock signals (“clocks”) used in the VR4100 processor. The subject matter includes basic system clocks, system timing parameters, connecting clocks to a phase-locked system, and connecting clocks to a system without phase locking.

8.1 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*).

Figure 8-1 and Figure 8-2 illustrate these terms.

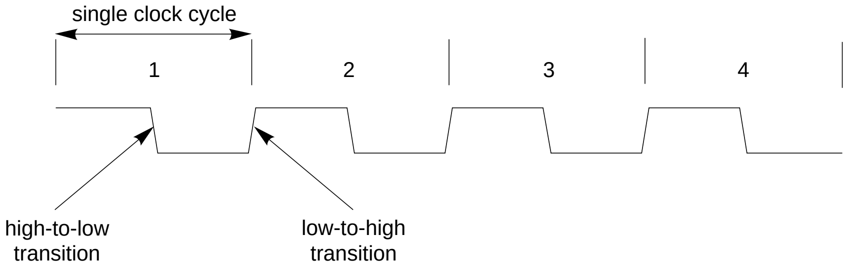


Figure 8-1 Signal Transitions

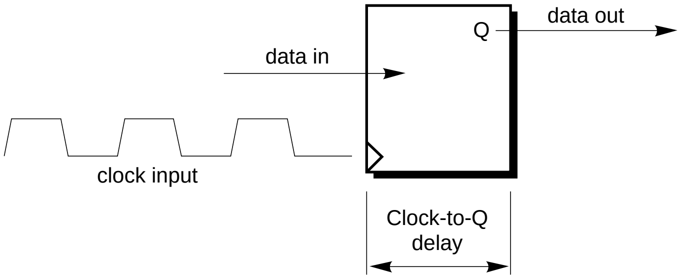


Figure 8-2 Clock-to-Q Delay

8.2 Basic System Clocks

The various clock signals used in the VR4100 processor are described below, starting with **MasterClock**, upon which the processor bases all internal and external clocking.

MasterClock

The processor bases all internal and external clocking on the single **MasterClock** input signal. The processor generates the clock output signal, **MasterOut**, at the same frequency as **MasterClock**.

MasterOut

The processor generates the clock output signal, **MasterOut**, at the same frequency as **MasterClock**. **MasterOut** clocks external logic, such as the reset and interrupt logic. The edges of **MasterOut** align with those of **MasterClock** immediately after $\overline{\textbf{ColdReset}}$ is deasserted.

PClock

The processor generates an internal clock, **PClock**, it is normally four times the **MasterClock** frequency and precisely aligns every other rising edge of **PClock** with the rising edge of **MasterOut**.
All internal registers and latches use **PClock**, which is the pipeline clock rate.

SClock

The VR4100 processor divides **PClock** by 2 programmed by $\overline{\textbf{DIV2}}$ pin to generate the internal clock signal, **SClock**. The processor uses **SClock** to sample data at the system interface and to clock data into the processor system interface output registers.

The first rising edge of **SClock**, after $\overline{\textbf{ColdReset}}$ is deasserted, is aligned with the first rising edge of **MasterOut**.

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.

[†] *External agent* is defined in Chapter 11.

TClock is identical to **SClock**. The edges of **TClock** align precisely with the edges of **SClock** .

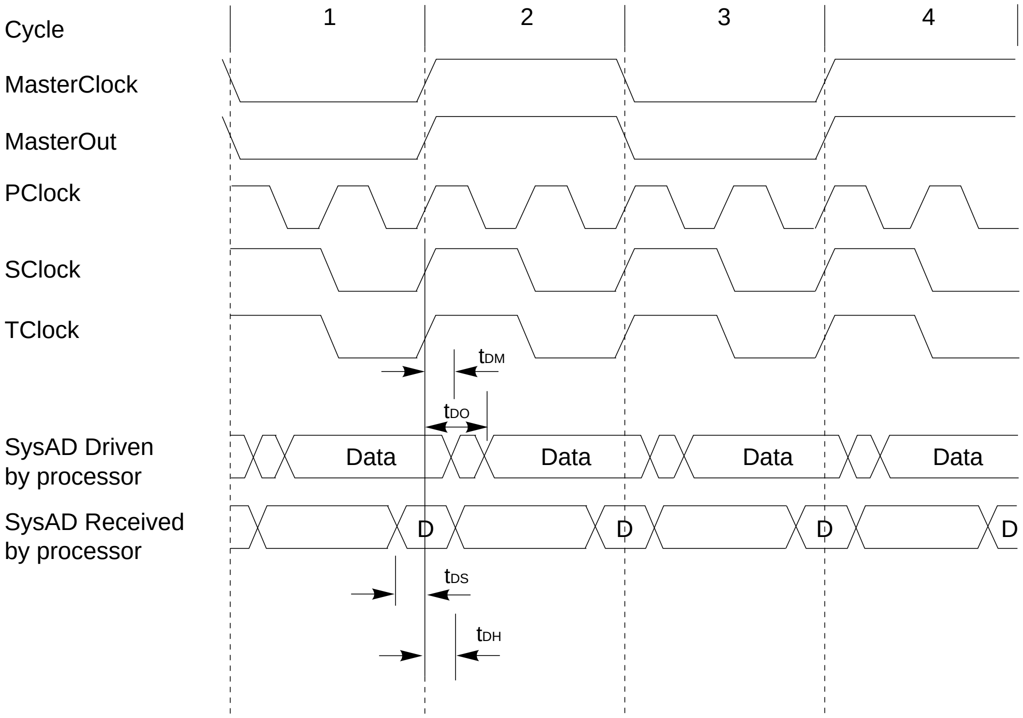


Figure 8-3 Processor Clocks, ***PClock-to-SClock Division by 2***

8.3 System Timing Parameters

As shown in Figure 8-3 data provided to the processor must be stable a minimum of t_{DS} nanoseconds (ns) before the rising edge of **TClock** and be held valid for a minimum of t_{DH} ns after the rising edge of **TClock**.

Alignment to SClock

Processor data becomes stable a minimum of t_{DM} ns and a maximum of t_{DO} ns after the rising edge of **SClock**. This drive-time is the sum of the maximum delay through the processor output drivers together with the maximum clock-to-Q delay of the processor output registers.

Alignment to MasterOut

Certain processor inputs (specifically $\overline{\textbf{ColdReset}}$, $\overline{\textbf{Reset}}$ and $\overline{\textbf{Int}}$(4:0)) are sampled based on **MasterOut**.

Phase-Locked Loop (PLL)

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

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

8.4 Connecting Clocks to a System without Phase Locking

When the VR4100 processor is used in a system in which the external agent cannot lock its phase to a common **MasterClock**, the output clocks **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

Figure 8-4 is a block diagram of a system without phase lock, using the VR4100 processor with an external agent implemented as a gate array.

When connecting to a gate array device, **TClock** is used with the gate array. The gate array internal buffers **TClock** and 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, and sample processor output.

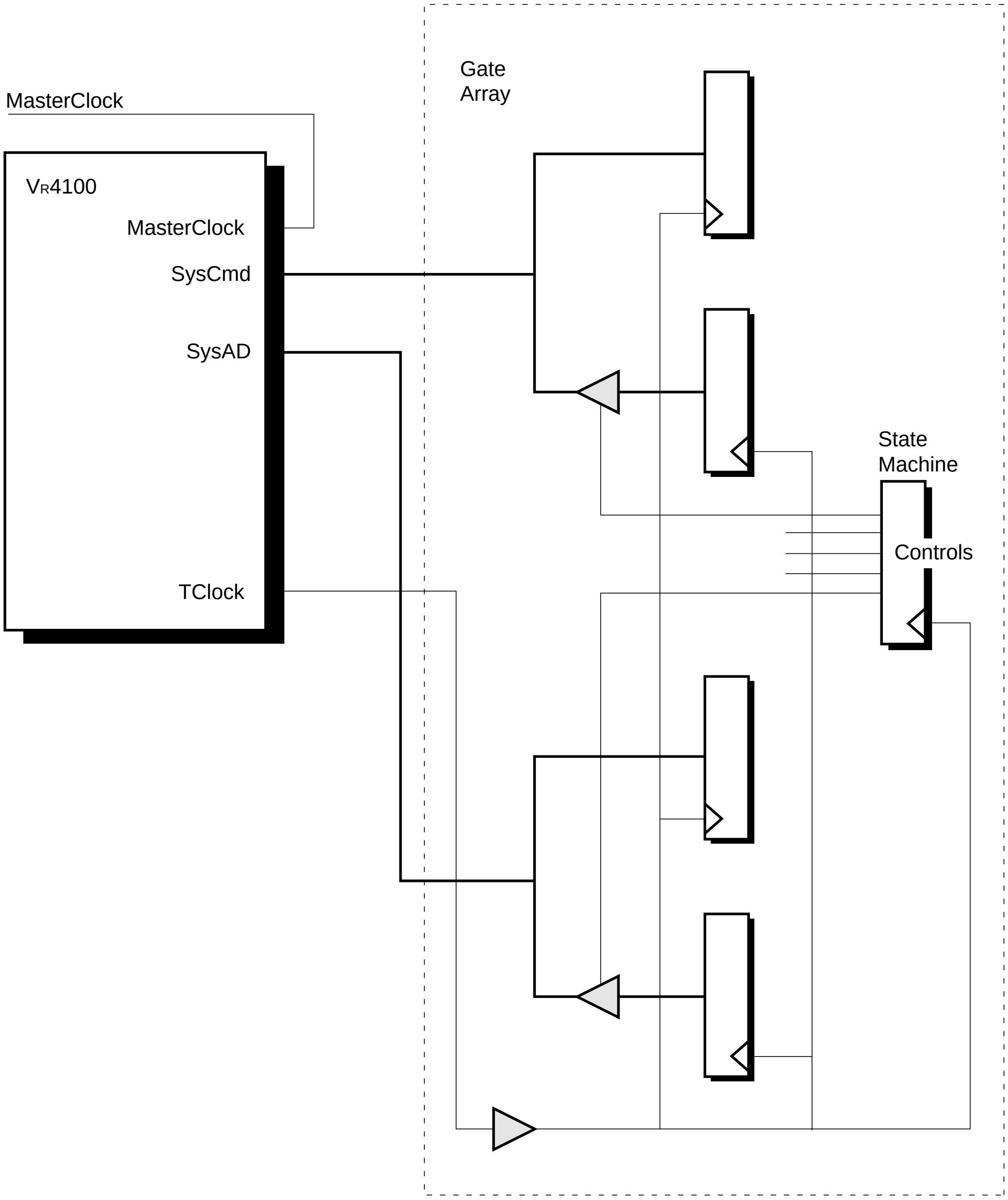


Figure 8-4 Gate-Array System without Phase Lock, using the VR4100 Processor

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 = (TClock period) - (t_{DO} for V_R4100)
+ (Minimum External Clock Buffer Delay)
- (External Sample Register Setup Time)
- (Maximum Clock Jitter for VR4100 Internal Clocks)
- (Maximum Clock Jitter for TClock)

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 VR4100)
- (Maximum External Clock Buffer Delay)
- (Maximum External Output Register Clock-to-Q Delay)
- (Maximum Clock Jitter for TClock)
- (Maximum Clock Jitter for VR4100 Internal Clocks)

Connecting to a CMOS Logic System

When processor output clock are supplied to many devices, use external clock buffers. Calculating the transmission time needs clock buffer delay. ★

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 VR4100)

+ (Minimum Delay Mismatch for External ClockBuffer)

- (Setup Time for External Input Buffer)

- (Maximum Clock Jitter for VR4100 Internal Clocks)

- (Maximum Clock Jitter for TClock)

Figure 8-5 is a block diagram of a system without phase lock, employing the VR4100 processor and an external agent composed of both a gate array and discrete CMOS logic devices.

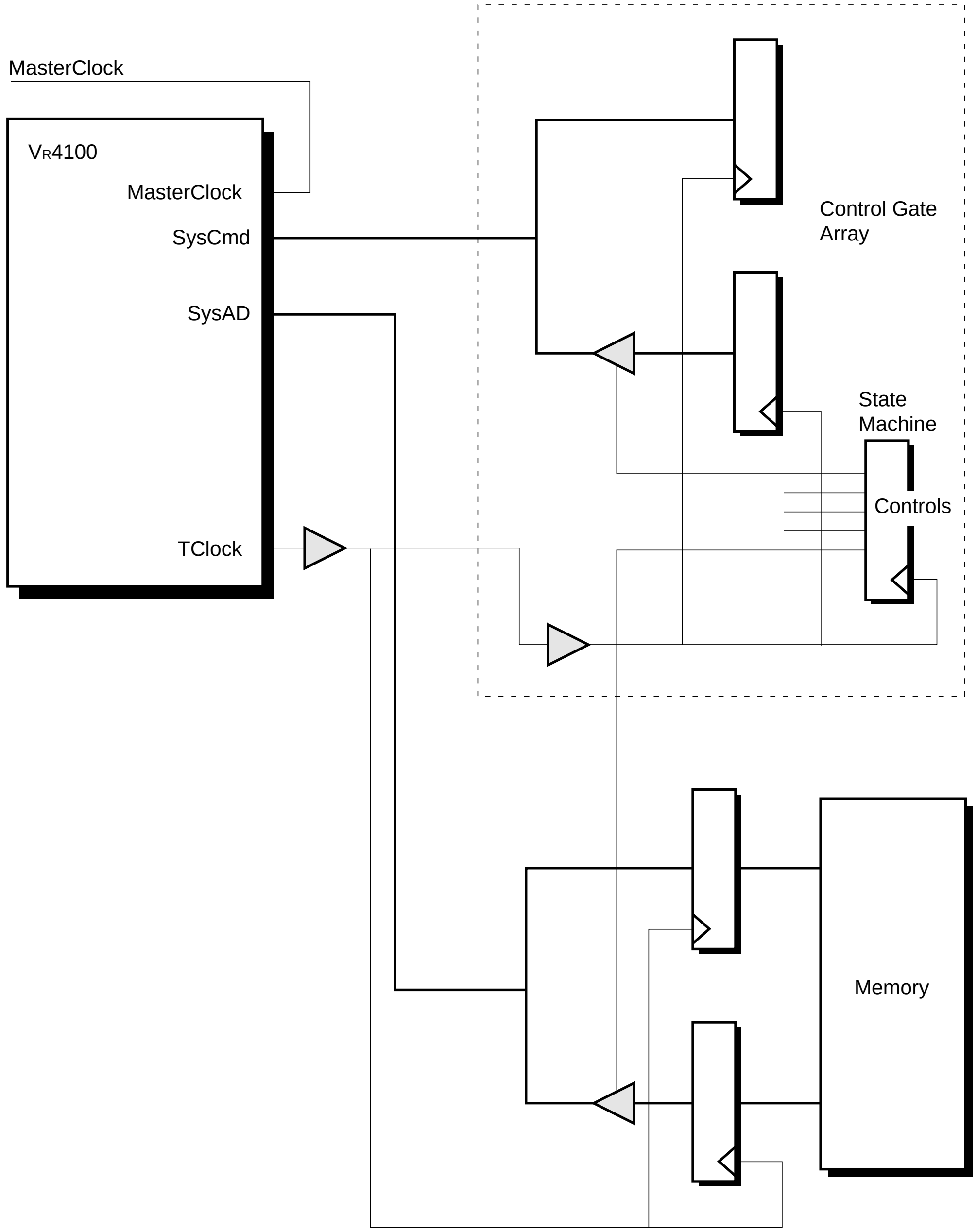


Figure 8-5   Gate Array and CMOS System without Phase Lock, using the VR4100 Processor

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$_{R}$4100) ★

- (Maximum External Output Register Clock-to-Q Delay)
- (Maximum External Clock Buffer Delay)
- (Maximum Clock Jitter for V$_{R}$4100 Internal Clocks)
- (Maximum Clock Jitter for TClock)

[MEMO]

Power Mode

9

This chapter describes in detail the VR4100 Power modes: FullSpeed mode, Standby mode, Suspend mode and Hibernate mode.

9.1 VR4100 Power Mode

The VR4100 processor supports four power modes:

- FullSpeed mode
- Standby mode
- Suspend mode
- Hibernate mode

These modes are described in the next sections.

Power Mode State Machine

Figure 9-1 shows Power mode State machine.

To enter Standby mode, Suspend mode and Hibernate mode from FullSpeed mode, execute special instructions, STANDBY, SUSPEND and HIBERNATE instructions.

To return FullSpeed mode from Standby mode or Suspend mode, generate interrupt or NMI, or execute SoftReset or ColdReset sequence.

To return FullSpeed mode from Hibernate mode, execute ColdReset sequence.

Table 9-1 shows Power mode overview.

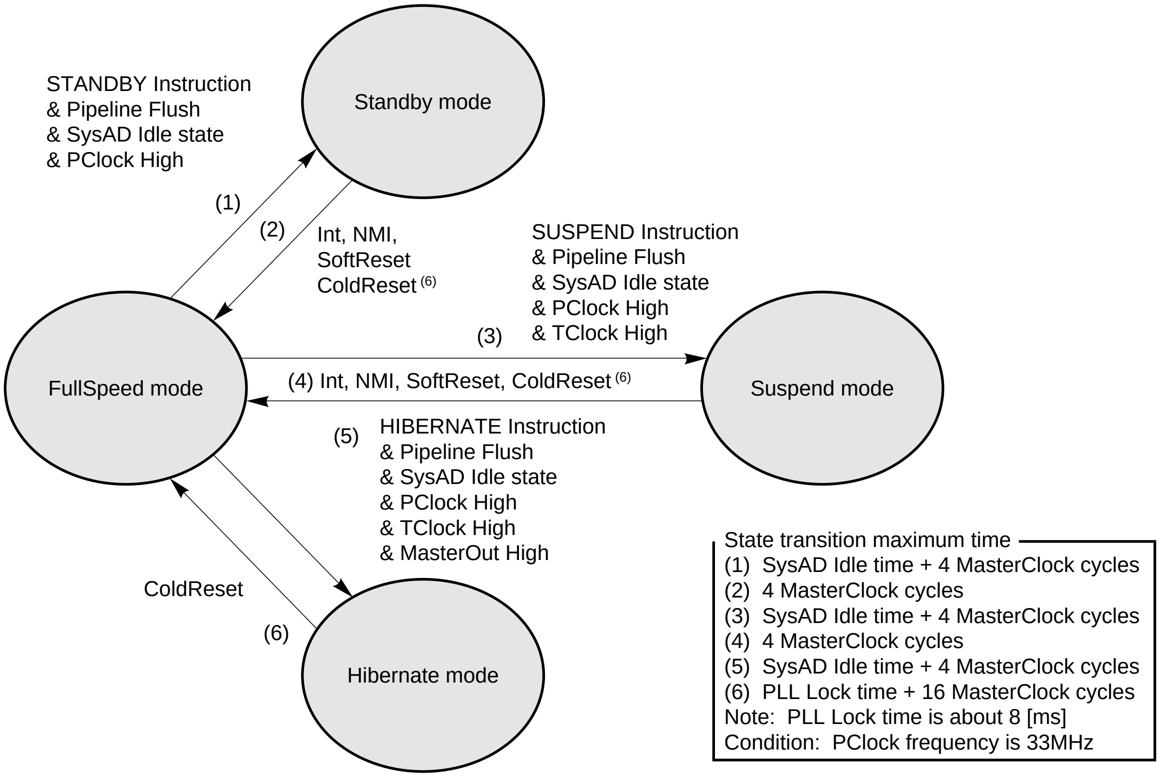


Figure 9-1 Power Mode State Machine

Mode	PLL	MasterOut	Timer/Interrupt	TClock (Bus clock)	PClock (Pipeline clock)	Restart Event
FullSpeed	ON	ON	ON	ON	ON	---
Standby	ON	ON	ON	ON	OFF	Int, NMI, SoftReset, ColdReset
Suspend	ON	ON	ON	OFF	OFF	Int, NMI, SoftReset, ColdReset
Hibernate	OFF	OFF	OFF	OFF	OFF	ColdReset

Table 9-1 Power Mode Overview

FullSpeed Mode

In FullSpeed mode, all internal clocks and all system interface clocks are active. In FullSpeed mode, VR4100 can execute every functions.

Figure 9-2 shows FullSpeed mode Clocking.

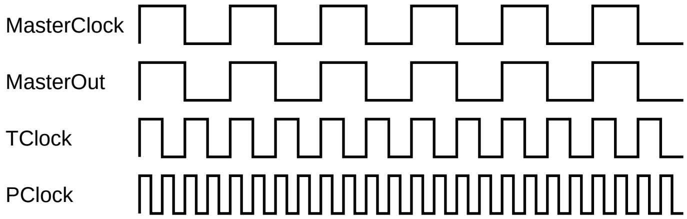


Figure 9-2 FullSpeed Mode Timing Chart: In Div2 Mode

Standby Mode

In Standby mode, all internal clocks, except Timer/Interrupt unit, are frozen at high level.

To enter Standby mode from FullSpeed mode, first execute the STANDBY instruction. When the STANDBY instruction finishes the WB stage, the VR4100 wait by the SysAD bus is idle state, after then the internal clocks will shut down, thus freezing the pipeline. The PLL, Timer/Interrupt clocks and the system interface clocks, TClock and MasterOut, will continue to run.

Once the VR4100 is in Standby mode, any interrupt, including the internally generated timer interrupt, NMI, SoftReset and ColdReset will cause the VR4100 to exit Standby mode and to enter FullSpeed mode.

Note: When a interrupt, NMI, SoftReset or ColdReset is detected after RF pipeline stage of STANDBY instruction but before clock stop, STANDBY instruction is nullified.

Figure 9-3 shows Standby mode timing chart.

★

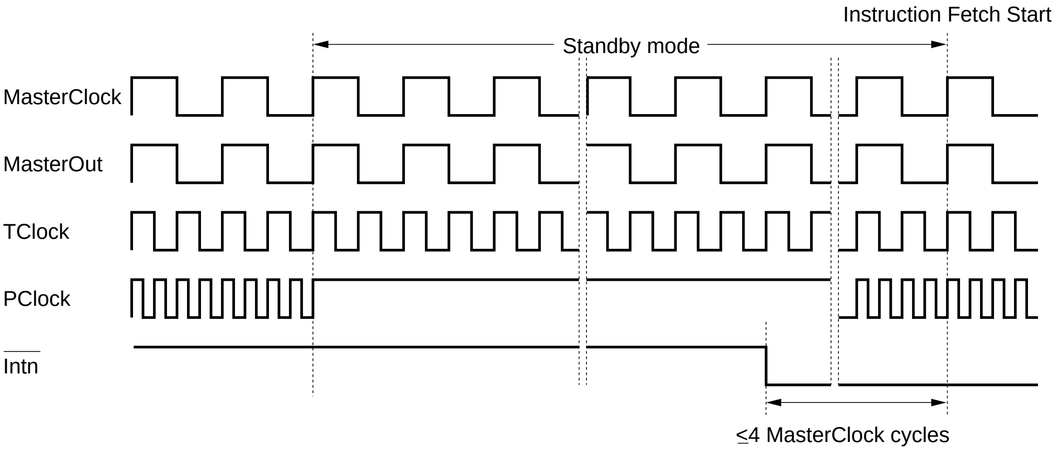


Figure 9-3   Standby Mode Timing Chart: In Div2 Mode

Suspend Mode

In Suspend mode, all internal clocks, except Timer/Interrupt unit, and the TClock are frozen at high level.

To enter Suspend mode from FullSpeed mode, first execute the SUSPEND instruction. When the SUSPEND instruction finishes the WB stage, the VR4100 wait by the SysAD bus is idle state, after then the internal clocks and the TClock will shut down, thus freezing the pipeline. The PLL, Timer/Interrupt clocks and MasterOut, will continue to run.

Once the VR4100 is in Suspend mode, any interrupt, including the internally generated timer interrupt, NMI, SoftReset and ColdReset will cause the VR4100 to exit Suspend mode and to enter FullSpeed mode.

Note: When a interrupt, NMI, SoftReset or ColdReset is detected after RF pipeline stage of SUSPEND instruction but before clock stop, SUSPEND instruction is nullified.

Figure 9-4 shows Suspend mode timing chart.

★

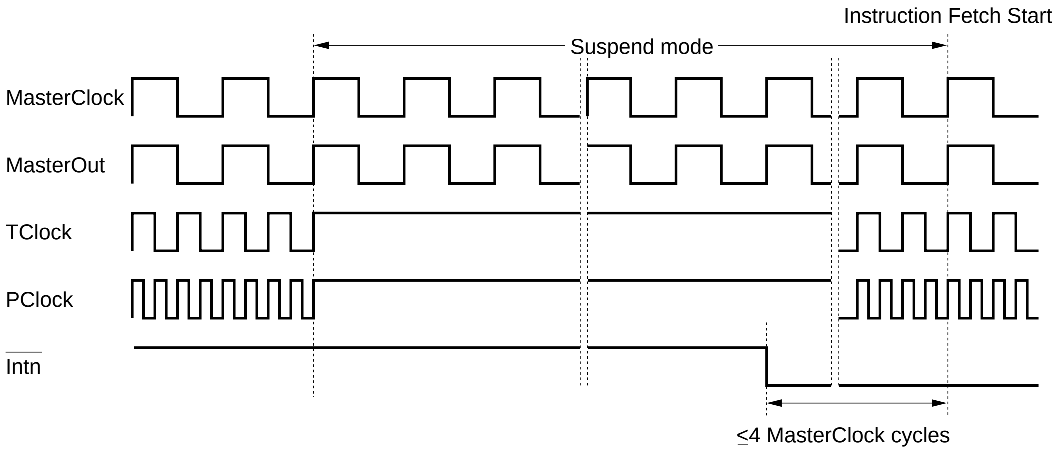


Figure 9-4 Suspend Mode Timing Chart: In Div2 Mode

Hibernate Mode

In Hibernate mode, all internal clock, include Timer/Interrupt unit, and all system interface clocks are frozen at high level.

To enter Hibernate mode from FullSpeed mode, first execute the HIBERNATE instruction. When the HIBERNATE instruction finishes the WB stage, the VR4100 wait by the SysAD bus is idle state, after then the internal clocks and the system interface clocks will shut down, thus freezing the pipeline.

Once the VR4100 is in Hibernate mode, the ColdRest sequence will cause the VR4100 to exit Hibernate mode and to enter FullSpeed mode.

Figure 9-5 shows Hibernate mode timing chart.

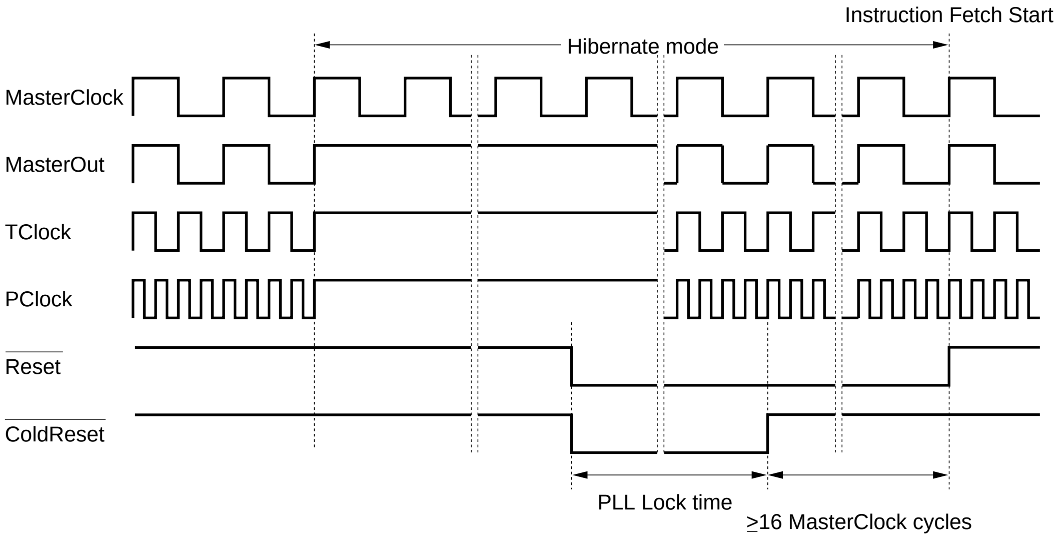


Figure 9-5 Hibernate Mode Timing Chart: In Div2 Mode

[MEMO]

Cache Organization and Operation

10

This chapter describes in detail the cache memory: its place in the VR4100 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 VR4100 system memory hierarchy. In the logical memory hierarchy, the 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.

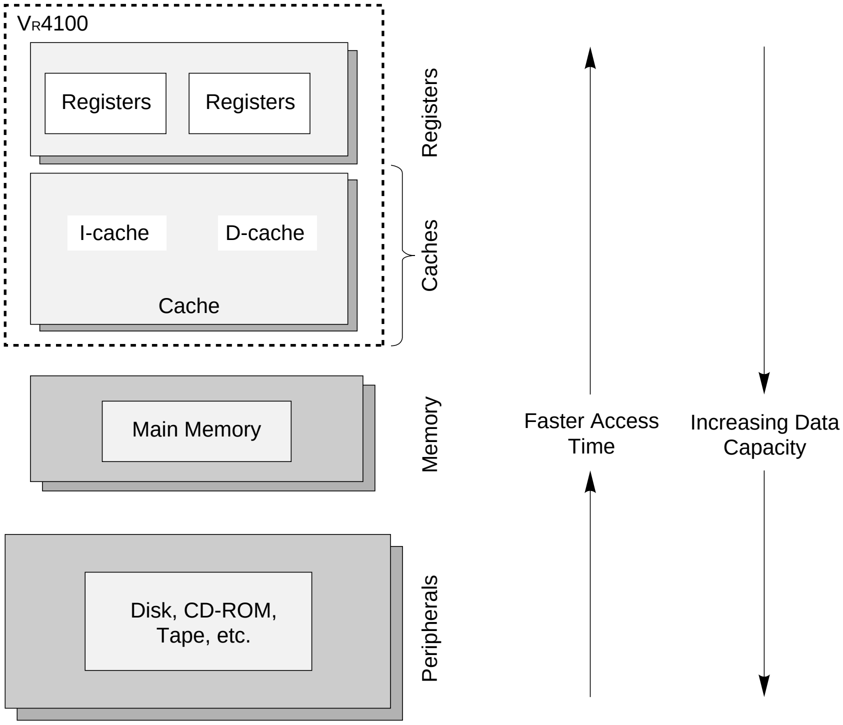


Figure 10-1 Logical Hierarchy of Memory

The VR4100 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 VR4100 cache and memory model.

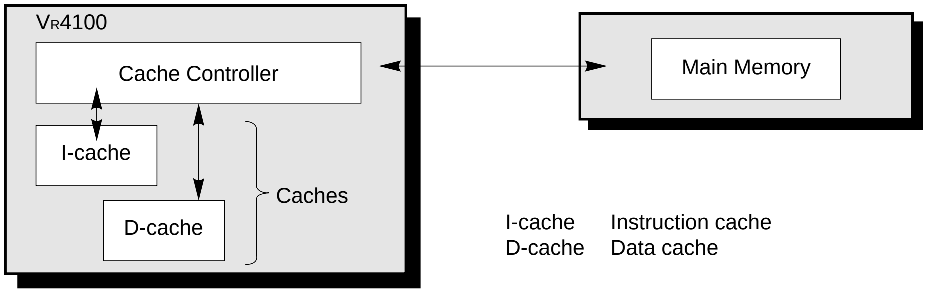


Figure 10-2 VR4100 Cache Support

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/data cache is 4 words (16 bytes).

Cache Sizes

The VR4100 instruction cache is 2 Kbytes; the data cache is 1 Kbytes.

† 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 24-bit tag that contains a 22-bit physical address, a single *Valid* bit, and a single *Parity* bit. Word parity is used on I-cache data (1 bit of parity per word).

The VR4100 processor I-cache has the following characteristics:

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

Figure 10-3 shows the format of a 4-word (16-byte) I-cache line.

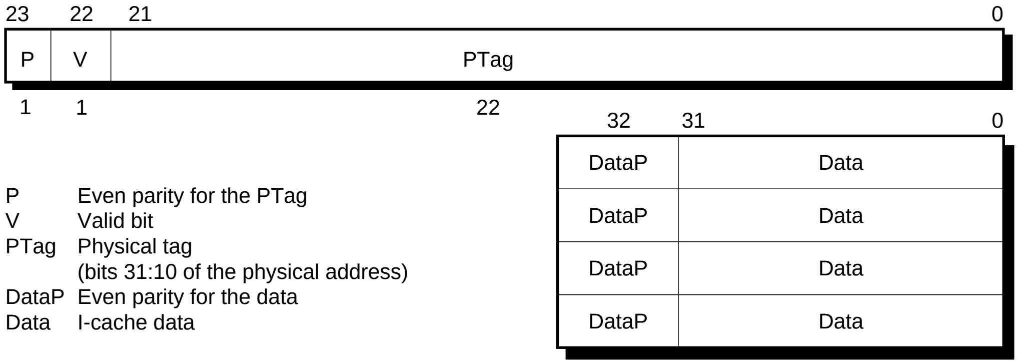

23
22
21
0
P
V
PTag
1
1
22
32
31
0
DataP
Data
DataP
Data
DataP
Data
DataP
Data
P Even parity for the PTag
V Valid bit
PTag Physical tag (bits 31:10 of the physical address)
DataP Even parity for the data
Data I-cache data

Figure 10-3 VR4100 4-Word I-Cache Line Format

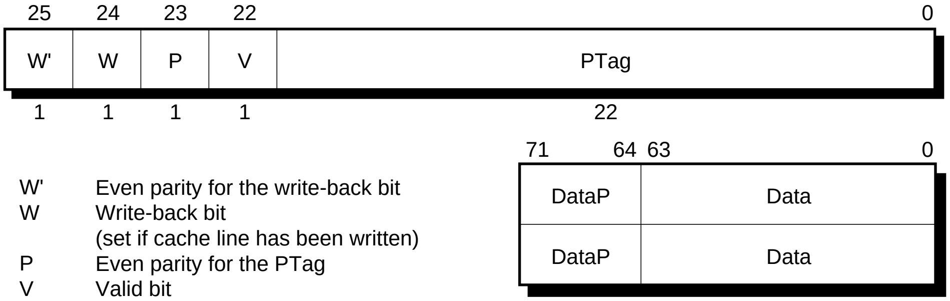
Organization of the Data Cache (D-Cache)

Each line of D-cache data has an associated 26-bit tag that contains a 22-bit physical address, a *Valid* bit, a *Parity* bit, a *Write-back* bit, and a *parity* bit for Write-back.

The VR4100 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.



W'	Even parity for the write-back bit
W	Write-back bit (set if cache line has been written)
P	Even parity for the PTag
V	Valid bit
PTag	Physical tag (bits 31:10 of the physical address)
DataP	Even parity for the data
Data	I-cache data

Figure 10-4 VR4100 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 used to index the instruction and data caches depends on the cache size.

For example, VA (9:4) accesses the 1-Kbyte page tag in the data cache with its 4-word line: VA (9) addresses 1 Kbytes and VA (4) provides quadword resolution.

Similarly, VA (10:4) accesses an 4-word tag in a 2 Kbyte I-cache: VA (4) provides quadword resolution and VA (10) addresses 2 Kbytes.

★

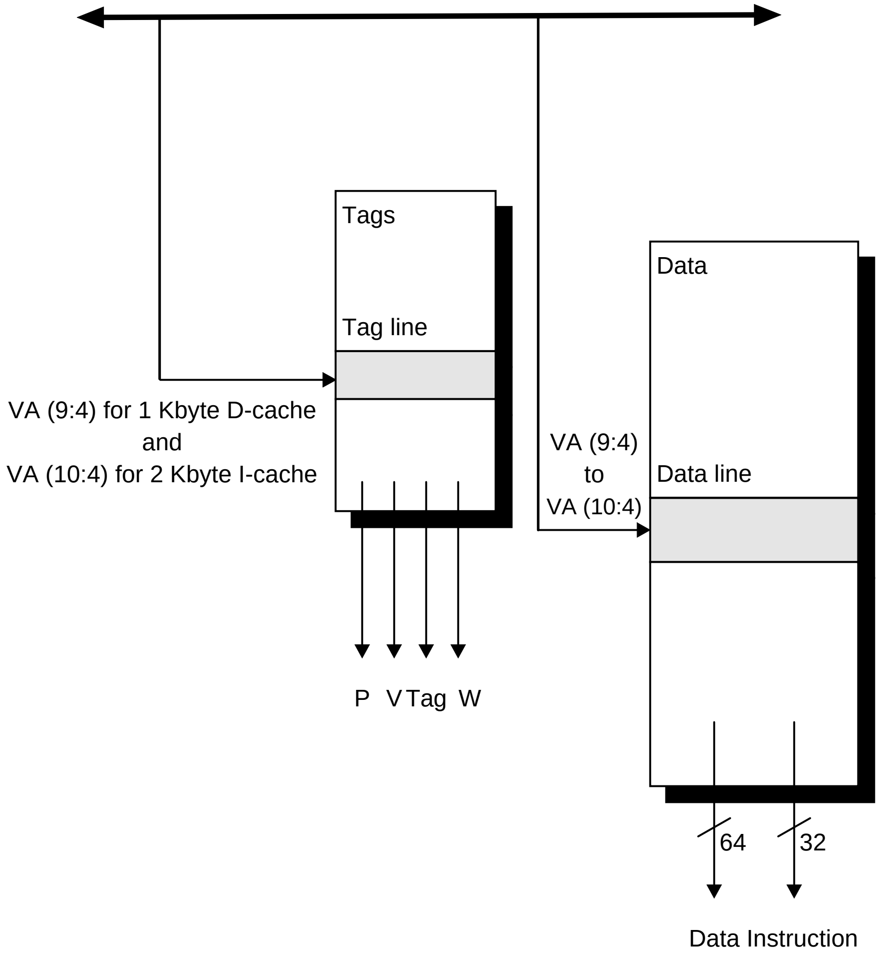


Figure 10-5   Cache Data and Tag Organization

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.

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

Cache Write Policy

The VR4100 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 VR4100 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.

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. CACHE operations are described in Chapter 14.

[†] 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.

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

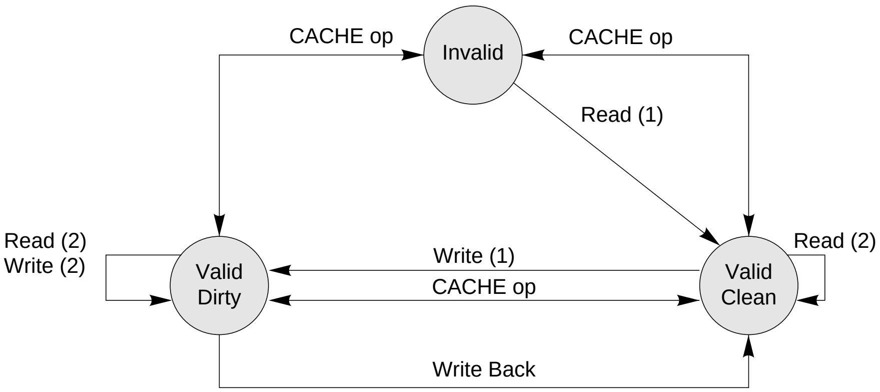


Figure 10-6 Data Cache State Diagram

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.

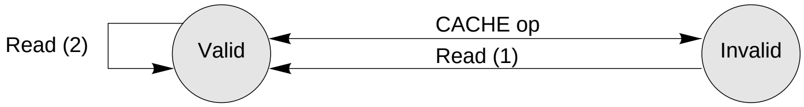


Figure 10-7   Instruction Cache State Diagram

10.6 Cache Data Integrity

The D- and I-cache data RAM arrays are protected by parity. D- and I-cache tag RAM arrays are also protected by parity.

These parity bits are checked for errors on every cache read access. The processor takes a cache error exception if it encounters a parity error during an instruction cache access, a data cache access, or memory read access. The *CacheErr* register indicates the source of the error.

Figure 10-8 to Figure 10-22 shows the parity generation and checking operations for various cache accesses.

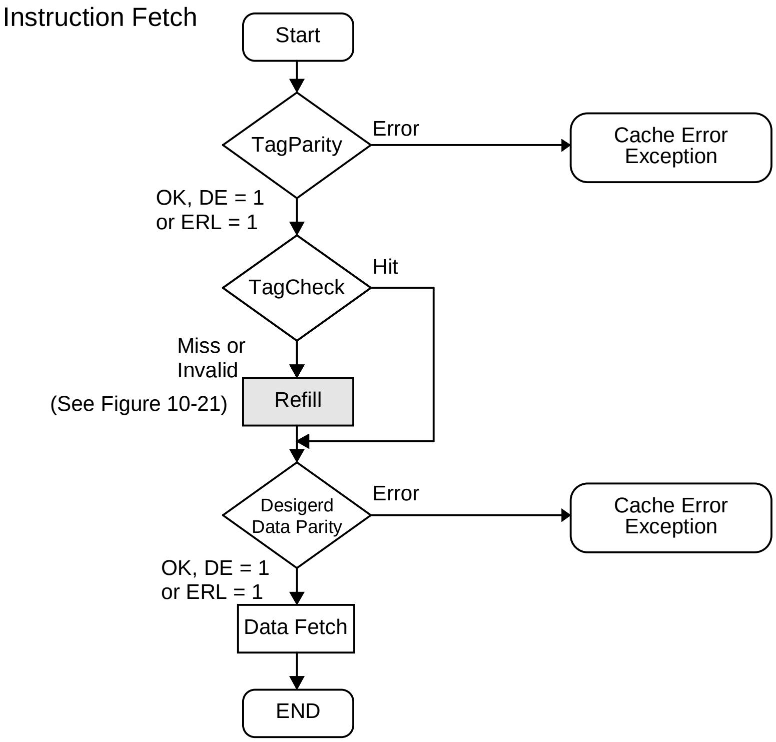


Figure 10-8 Data flow on Instruction Fetch

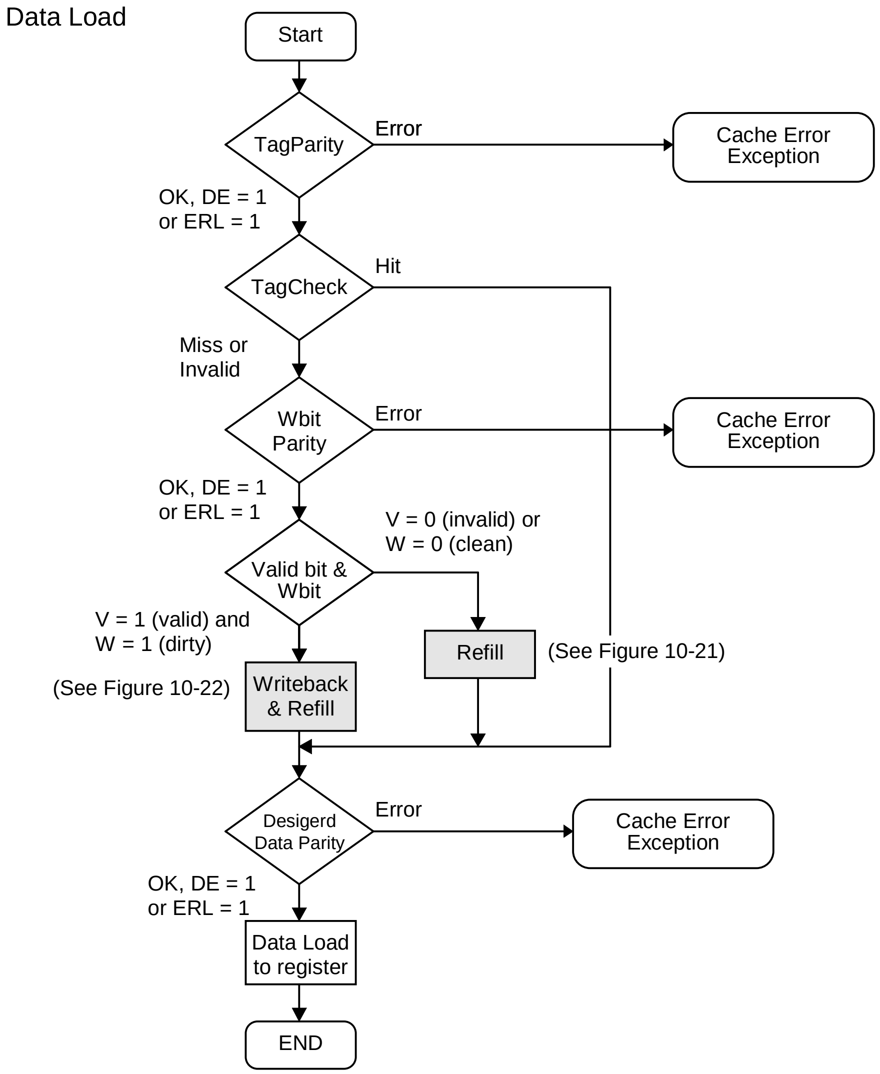


Figure 10-9   Data Integrity on Load Operations

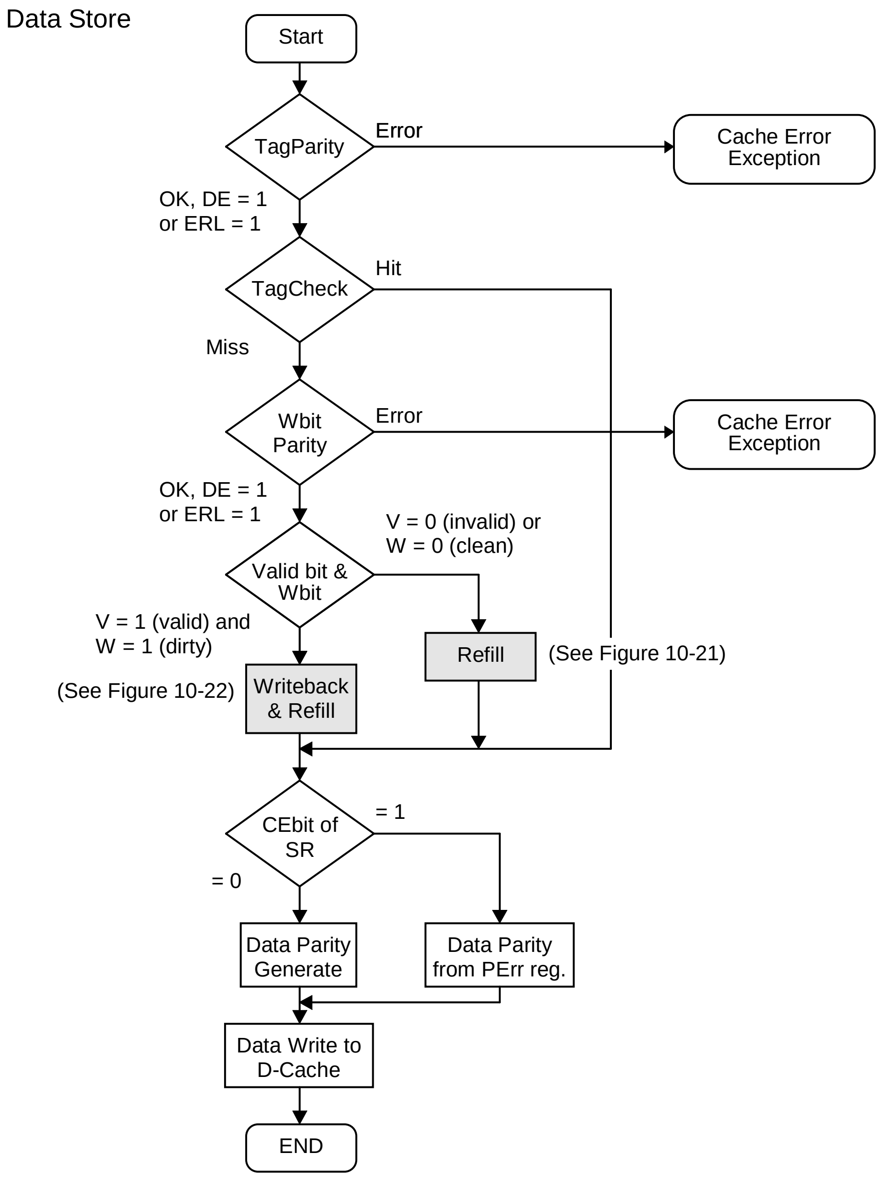


Figure 10-10 Data Integrity on Store Operations

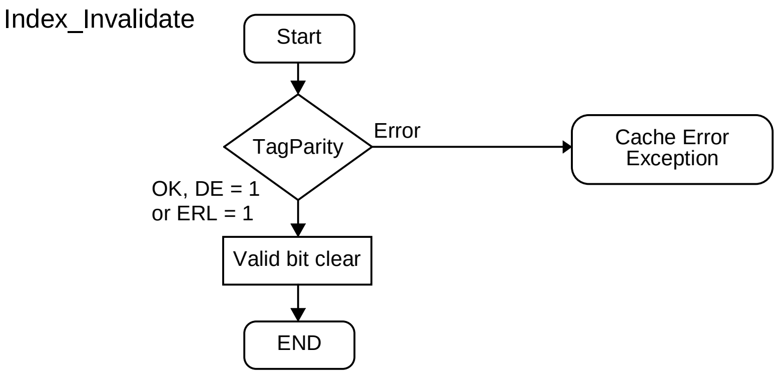


Figure 10-11   Data Integrity on Index_Invalidate Operations

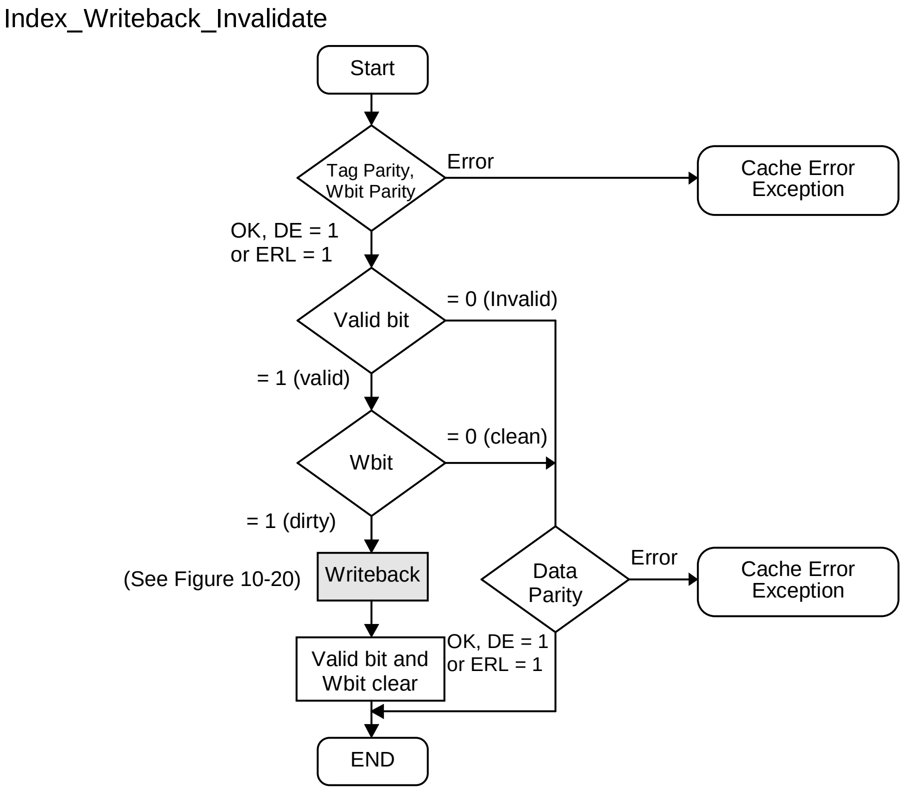


Figure 10-12   Data Integrity on Index_Writeback_Invalidate Operations

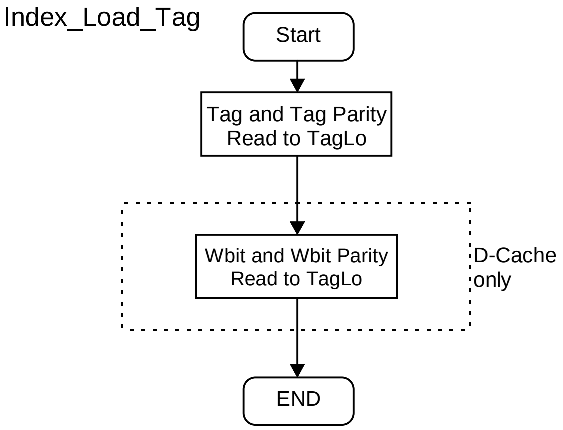

Index_Load_Tag
Start
Tag and Tag Parity Read to TagLo
Wbit and Wbit Parity Read to TagLo
D-Cache only
END

Figure 10-13 Data Integrity on Index_Load_Tag Operations

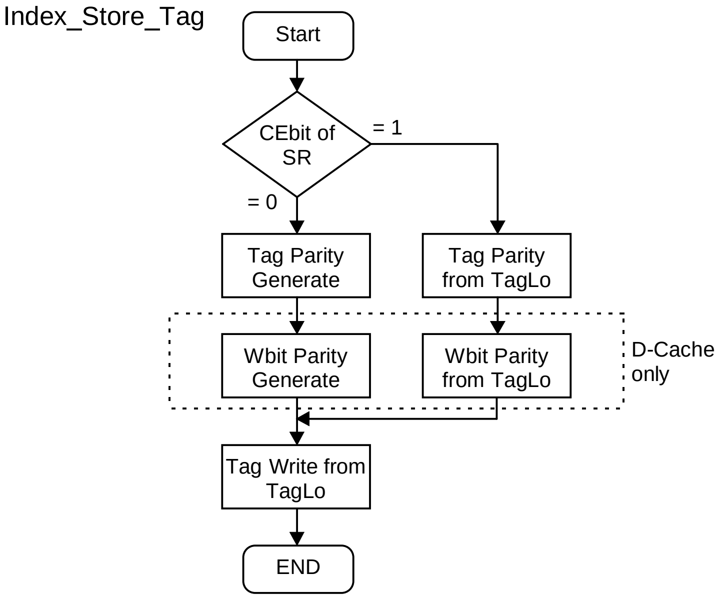


Figure 10-14 Data Integrity on Index_Store_Tag Operations

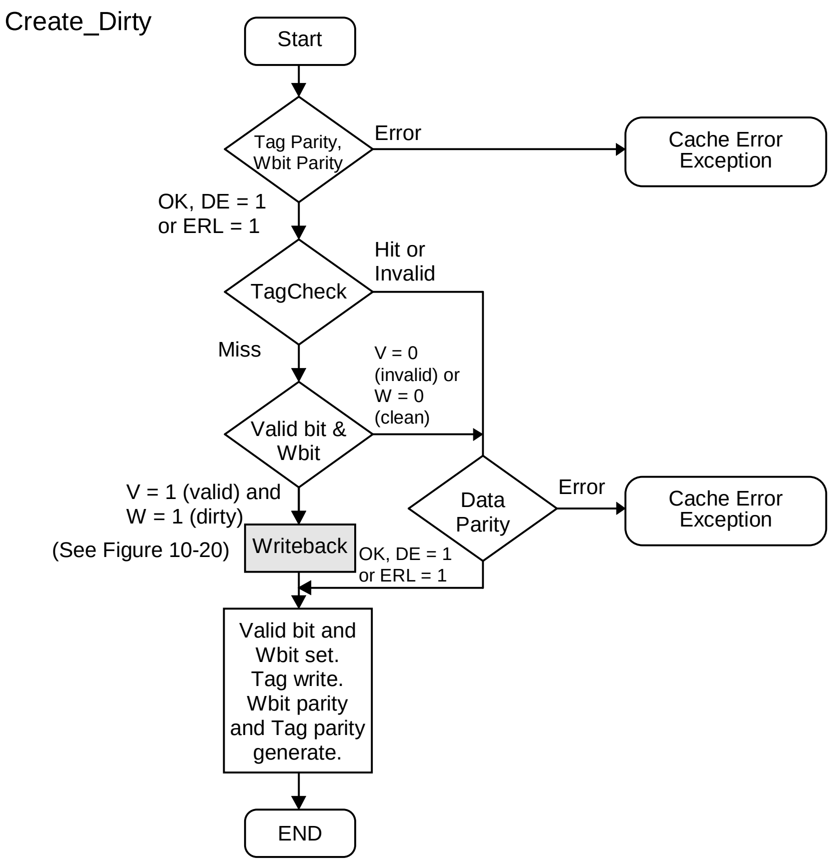


Figure 10-15 Data Integrity on Create_Dirty Operations

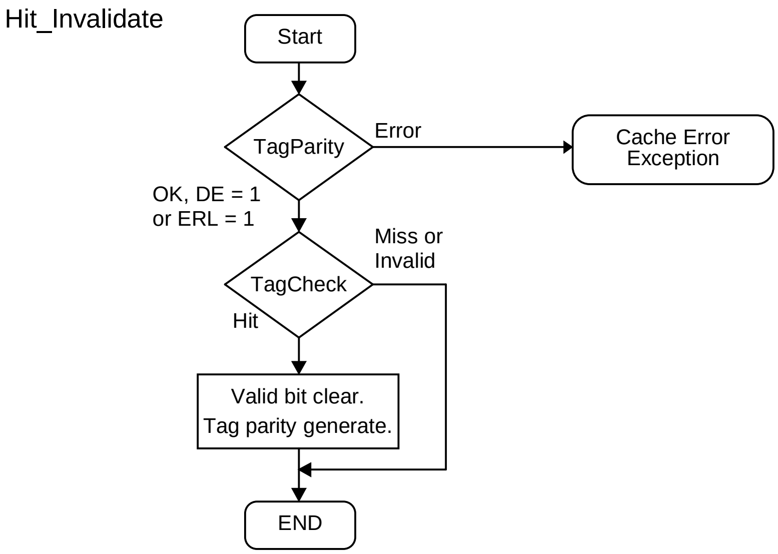


Figure 10-16 Data Integrity on Hit_Invalidate Operations

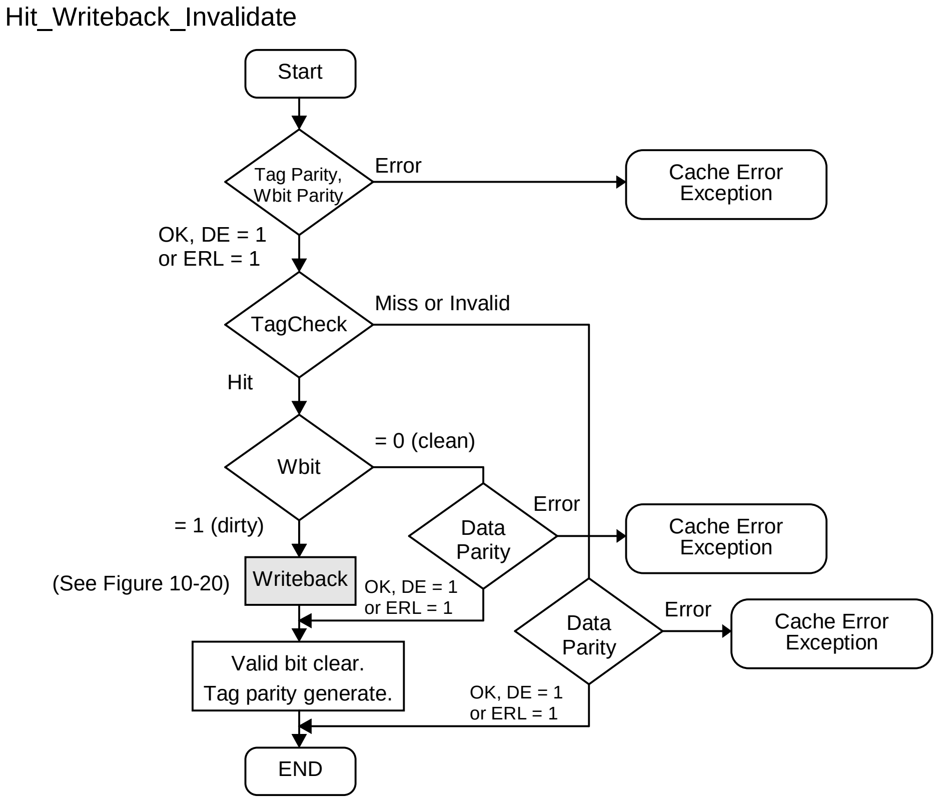


Figure 10-17 Data Integrity on Hit_Writeback_Invalidate Operations

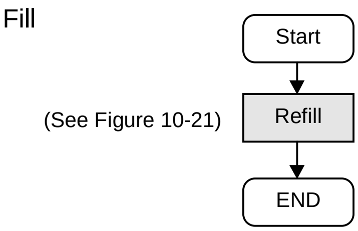


Figure 10-18 Data Integrity on Fill Operations

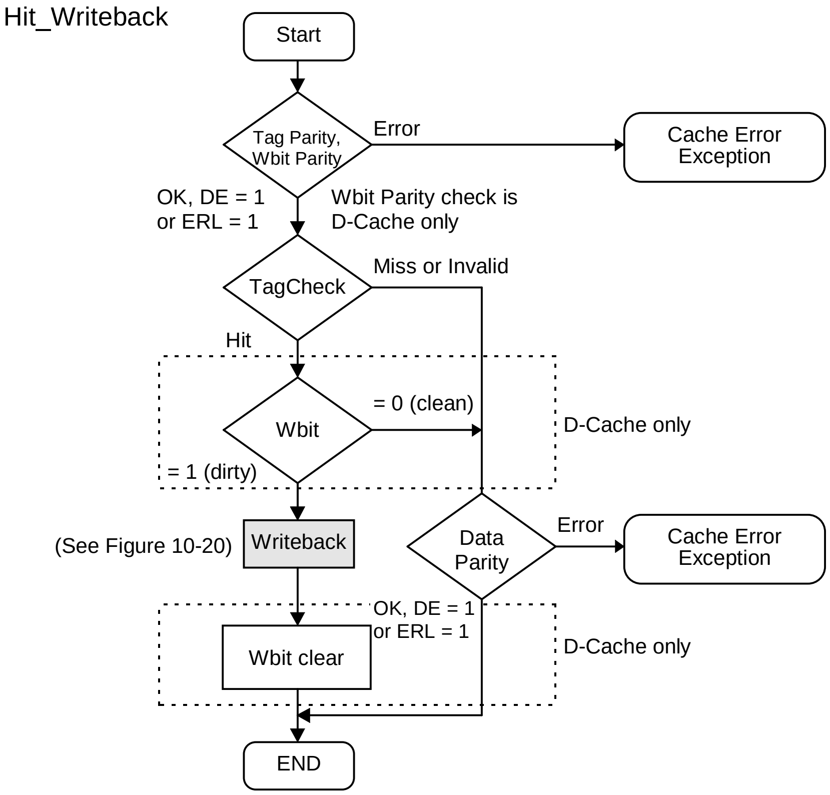


Figure 10-19 Data Integrity on Hit_Writeback Operations

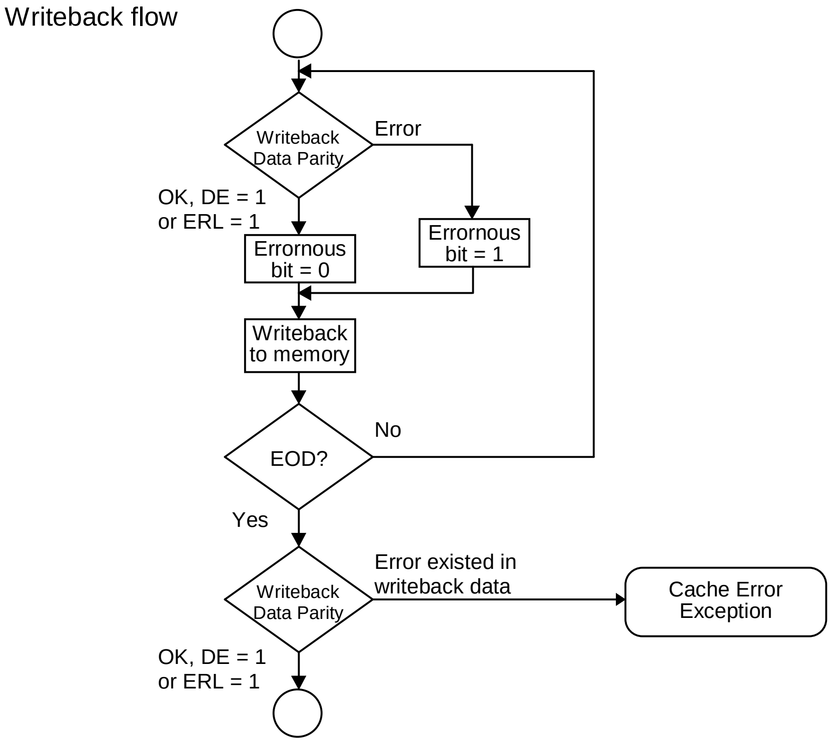


Figure 10-20 Data Integrity on Writeback Flow

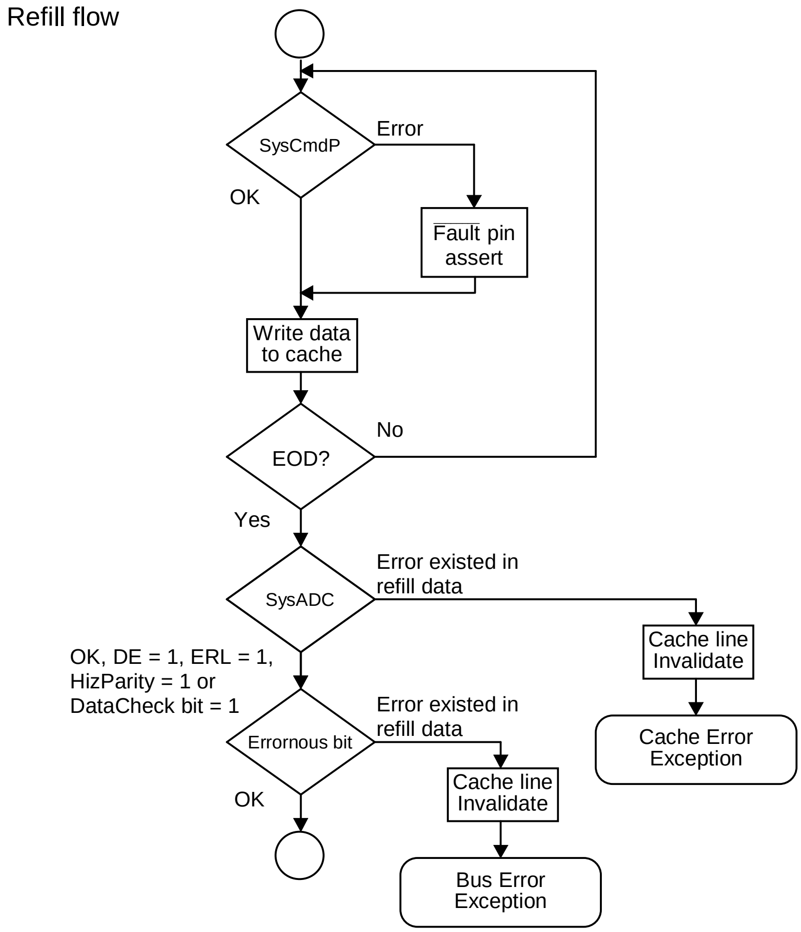


Figure 10-21 Data Integrity on Refill Flow

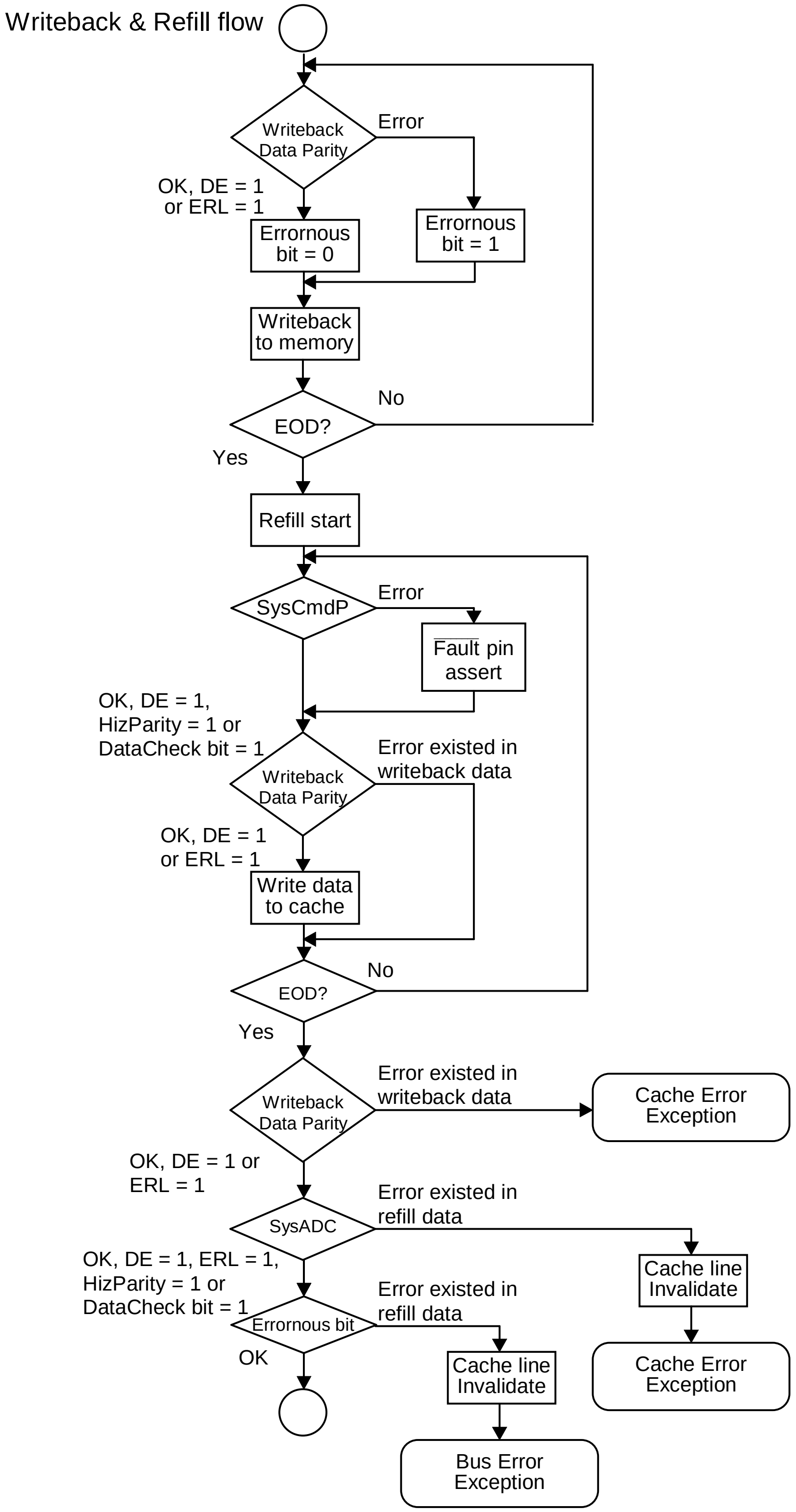


Figure 10-22 Data Integrity on Writeback & Refill Flow

Write-back Procedure: On a store miss write-back, data and tag parity is checked and data parity is transferred to the write buffer. Byte parity is generated for the physical address and transferred to write buffer. If an error is discovered on the data field, the write back is not terminated; the erroneous data is still written out. If an error is discovered in the tag field, the write-back bus cycle is not issued. In both cases a cache error exception is taken.

If a tag parity error occurs during a CACHE operation, the Cache Error exception is taken and the operation is not permitted to complete.

10.7 Manipulation of the Caches by an External Agent

The VR4100 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.

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.
- *Sequence* refers to the precise series of requests that a processor generates to service a system event.
- *Protocol* refers to the cycle-by-cycle signal transitions that occur on the System interface pins to assert a processor or external request.
- *Syntax* refers to the precise definition of bit patterns on encoded buses, such as the command bus.
- *Block* indicates any transfer larger than 8 bytes across the System interface.
- *Single* indicates any transfer of 4 bytes across the System interface.
- *Fetch* refers to the retrieval of information from the instruction cache.
- *Load* refers to the retrieval of information from the data cache.

11.2 System Interface Description

The VR4100 processor supports a 32-bit address/data interface. The System interface consists of:

- 32-bit address and data bus, **SysAD**
- 4-bit SysAD check bus, **SysADC**
- 5-bit command bus, **SysCmd**
- 1-bit SysCmd check bus, **SysCmdP**
- six handshake signals:
  - $\overline{\textbf{ERdy}}$
  - $\overline{\textbf{EReq}}$, $\overline{\textbf{PMaster}}$, $\overline{\textbf{PReq}}$
  - $\overline{\textbf{EValid}}$, $\overline{\textbf{PValid}}$

The processor uses the System interface to access external resources such as cache misses and uncached operations.

Physical Addresses

Physical addresses are driven on **SysAD (31:0)** during address cycles.

Addresses associated with non-block read and write requests are aligned for the size of the data element.

- For word requests, the low order two bits of the address are zero.
- For half-word requests, the low order bit of the address is zero.
- For byte and tribyte requests, the address provided is a byte address.

The address when a block read or block write request is issued is aligned in double word units. Therefore, the low-order 3 bits of the address become 0.

During cache access, the cache block's start address (low-order bits of address) is not necessarily output. Subblock ordering is used as the method for setting the retrieval sequence for block data. For details, refer to Subblock Ordering at the end of this chapter.

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 (see Processor and External Requests, in this chapter, for more information).

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.

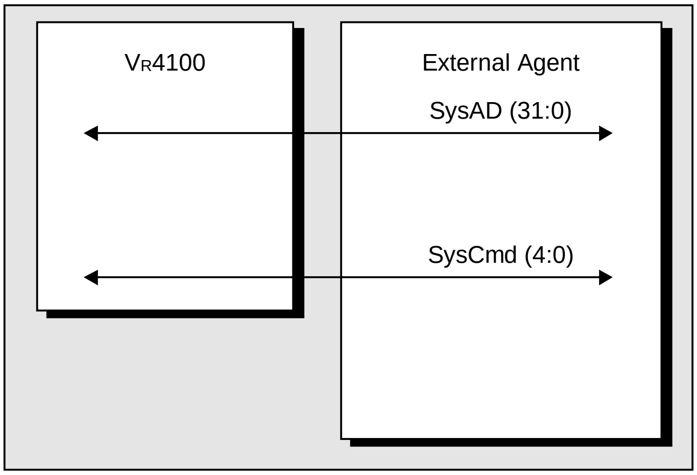


Figure 11-1   System Interface Buses

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. Validity is determined by the state of the $\overline{\textbf{EValid}}$ and $\overline{\textbf{PValid}}$ signals (described in Interface Buses, in this chapter).

When the VR4100 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 $\overline{\textbf{PValid}}$ signal when the **SysAD** and **SysCmd** buses are valid.
- When the processor is slave, an external agent asserts the $\overline{\textbf{EValid}}$ signal when the **SysAD** and **SysCmd** buses are valid.

The **SysADC** bus provides even byte parity for the **SysAD** bus, and the **SysCmdP** signal provides even parity over the five bits of the **SysCmd** bus. Parity is driven by the current master and checked by the corresponding slave.

- During address cycles [**SysCmd (4)** = 0], the remainder of the **SysCmd** bus, **SysCmd (3:0)**, contains a *System interface command* (the encoding of System interface commands is detailed in System Interface Commands and Data Identifiers, in this chapter).
- During data cycles [**SysCmd (4)** = 1], the remainder of the **SysCmd** bus, **SysCmd (3:0)**, contains a *data identifier* (the encoding of data identifiers is detailed in System Interface Commands and Data Identifiers, in this chapter).

Issue Cycles

There are two types of processor issue cycles:

- processor read request
- processor write request

The processor samples the input signal $\overline{\textbf{ERdy}}$ to determine the issue cycle for both processor read/write requests.

As shown in Figure 11-2, $\overline{\textbf{ERdy}}$ must be asserted one cycles prior to the address cycle of the processor request to define the address cycle as the issue cycle.

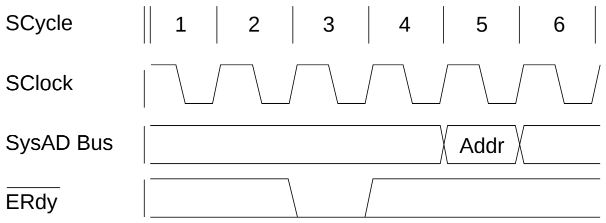


Figure 11-2 State of $\overline{\textbf{ERdy}}$ Signal for Processor Requests

The processor repeats the address cycle for the request until the conditions for a valid issue cycle are met. After the issue cycle, the data transmission begins. There is only one issue cycle for any processor request.

The processor accepts external requests, even while attempting to issue a processor request, by releasing the System interface to slave state in response to an assertion of $\overline{\textbf{EReq}}$ by the external agent.

Note that the rules governing the issue cycle of a processor request are strictly applied to determine the action the processor takes. The processor either:

- completes the issuance of the processor request in its entirety before the external request is accepted, or
- releases the System interface to slave state without completing the issuance of the processor request.

In the latter case, the processor issues the processor request (provided the processor request is still necessary) after the external request is complete. The rules governing an issue cycle again apply to the processor request.

Handshake Signals

The processor manages the flow of requests through the following six control signals:

- $\overline{\textbf{ERdy}}$ is used by the external agent to indicate when it can accept a new read or write transaction.
- $\overline{\textbf{EReq}}$, $\overline{\textbf{PMaster}}$ and $\overline{\textbf{PReq}}$ are used to transfer control of the **SysAD** and **SysCmd** buses. $\overline{\textbf{EReq}}$ is used by an external agent to indicate a need to control the interface. $\overline{\textbf{PMaster}}$ is de-asserted by the processor when it transfers control of the System interface to the external agent. And $\overline{\textbf{PReq}}$ is used by the processor to indicate a need to bus, when the processor is in slave state.
- The VR4100 processor uses $\overline{\textbf{PValid}}$ and the external agent uses $\overline{\textbf{EValid}}$ to indicate valid command/data on the **SysCmd/SysAD** buses.

11.3 System Interface Protocols

Figure 11-3 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.

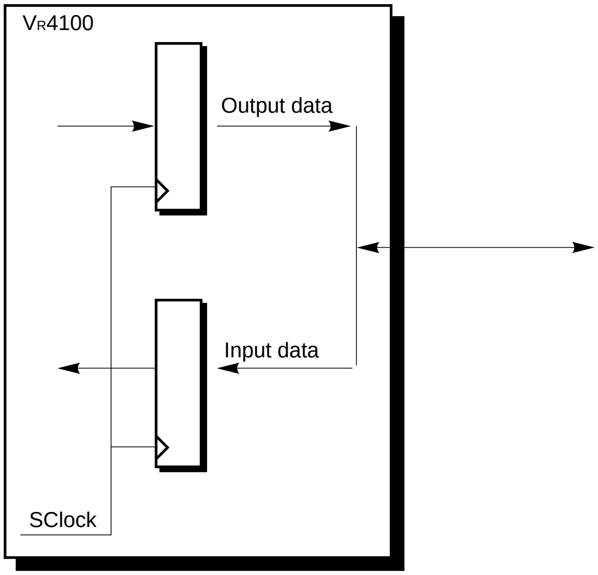


Figure 11-3 System Interface Register-to-Register Operation

Master and Slave States

When the VR4100 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 $\overline{\textbf{PValid}}$ whenever the **SysAD** and **SysCmd** buses are valid.

In slave state, the external agent asserts the signal $\overline{\textbf{EValid}}$ 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; see Chapter 8 for more details.

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 $\overline{\textbf{EReq}}$ and $\overline{\textbf{PMaster}}$. This transition is described by the following procedure:

1. An external agent signals that it wishes to issue an external request by asserting $\overline{\textbf{EReq}}$.
2. When the processor is ready to accept an external request, it releases the System interface from master to slave state by de-asserting $\overline{\textbf{PMaster}}$ continually.
3. The System interface returns to master state as soon as the issue of the external request is complete.

This process is described in External Arbitration Protocol, later in this chapter.

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. $\overline{\mathbf{PMaster}}$ is deasserted automatically after a read request. An uncompelled change to slave state occurs during the next 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 $\overline{\mathbf{EReq}}$.

After the external request is complete and external agent does not asserted $\overline{\mathbf{EReq}}$, the System interface returns to master state.

11.4 Processor and External Requests

There are two broad categories of requests: *processor requests* and *external requests*. These two categories are described in this section.

When a system event occurs, the processor issues a request or a series of requests through the system interface to access some external resource to service this event. For this to occur, the system interface must be connected to an external agent that coordinates the access to system resources. An external agent requesting access to a processor status register generates an *external request*. This access request passes through the System interface.

Processor requests include the following:

- read requests, which provide an address to an external agent
- write requests, which provide an address and a word or block of data to be written to an external agent

External requests include the following:

- read responses, which provide a block or single transfer of data from an external agent in response to read requests
- write requests, which provide 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.

A processor read request is complete after the last transfer of response data has been received from the external agent. A processor write request is complete after the last word of data has been transmitted.

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 pending.

System events and request cycles are shown in Figure 11-4.

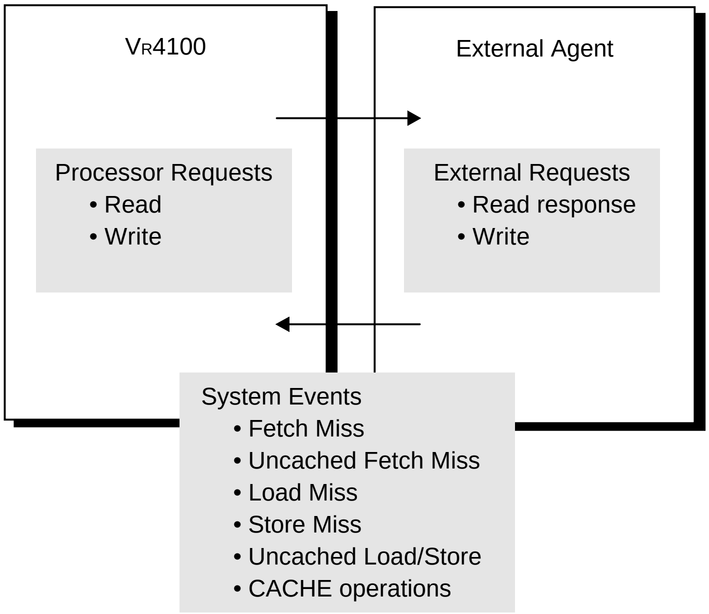


Figure 11-4   Requests and System Events

Processor Requests

A processor request is a request or a series of requests, through the System interface, to access some external resource. As shown in Figure 11-5, processor requests are either read or write requests.

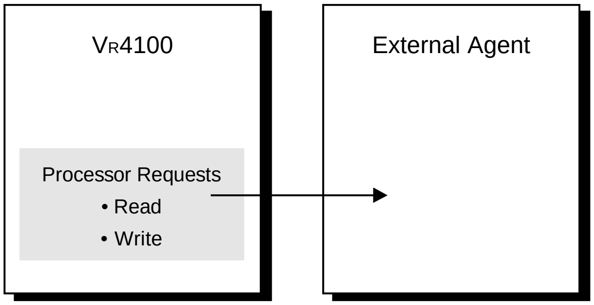


Figure 11-5 Processor Requests

Read request asks for a block, word, or partial word of data either from main memory or from another system resource.

Write request provides a block, word, or partial word of data to be written either to main memory or to another system resource.

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

The processor has the input signals $\overline{\textbf{ERdy}}$ to allow an external agent to manage the flow of processor requests.

The processor request cycle sequence is shown in Figure 11-6.

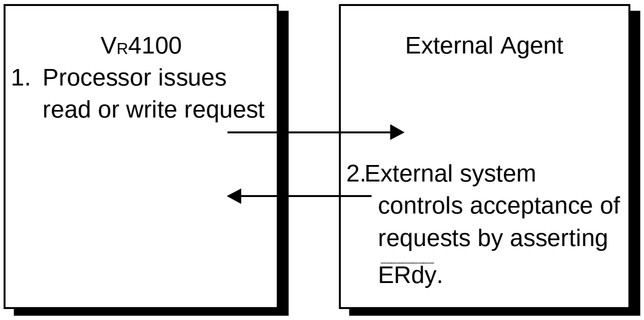

VR4100
1. Processor issues read or write request
External Agent
2.External system controls acceptance of requests by asserting $\overline{\text{ERdy}}$.

Figure 11-6 Processor Request

Processor Read Request

When a processor issues a read request, the external agent must access the specified resource and return the requested data. (Processor read requests are described in this section; external read responses are described in External Requests, later on in this chapter.)

A processor read request can be split from the external agent's return of the requested data; in other words, the external agent can initiate an unrelated external request before it returns the response data for a processor read. A processor read request is completed after the last word of response data has been received from the external agent.

Note that the data identifier (see System Interface Commands and Data Identifiers, in this chapter) associated with the response data can signal that the returned data is erroneous, causing the processor to take a bus error.

Processor read requests that have been issued, but for which data has not yet been returned, are said to be *pending.* A read remains pending until the requested read data is returned.

The external agent must be capable of accepting a new processor read request any time the following two conditions are met:

- There is no present processor read request pending.
- The signal $\overline{\textbf{ERdy}}$ has been asserted for one or more cycles.

Processor Write Request

When a processor issues a write request, the specified resource is accessed and the data is written to it. (Processor write requests are described in this section; external write requests are described in External Requests, later on in this chapter.)

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

The external agent must be capable of accepting a new processor write request any time the following two conditions are met:

- No present processor read request is pending.
- The signal $\overline{\textbf{ERdy}}$ has been asserted for one or more cycles.

External Requests

External requests include read response, and write requests, as shown in Figure 11-7.

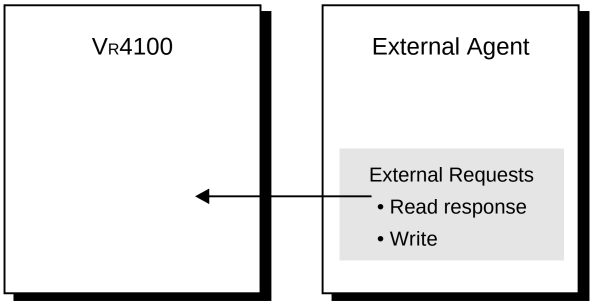


Figure 11-7   External Requests

Read response returns data in response to a processor read request.

Write request provides data to be written to the processor's internal resource.

The processor controls the flow of external requests through the arbitration signals $\overline{\textbf{EReq}}$ and $\overline{\textbf{PMaster}}$, as shown in Figure 11-8. The external agent must acquire mastership of the System interface before it is allowed to issue an external request; the external agent arbitrates for mastership of the System interface by asserting $\overline{\textbf{EReq}}$ and then waiting for the processor to de-assert $\overline{\textbf{PMaster}}$.

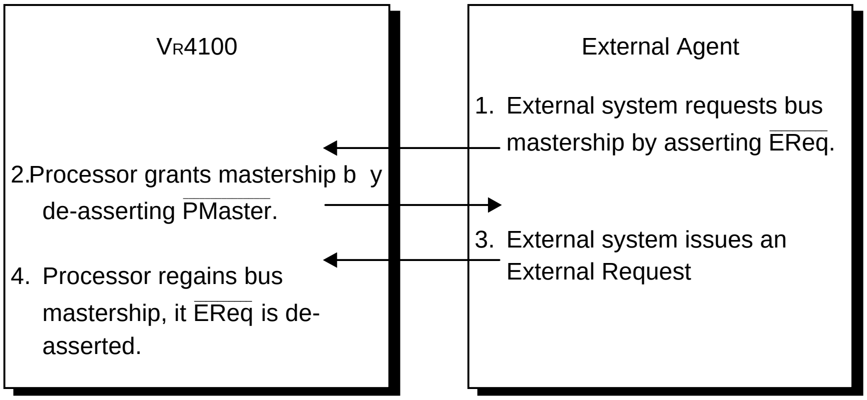

VR4100
External Agent
1. External system requests bus mastership by asserting $\overline{\text{EReq}}$.
2.Processor grants mastership b y de-asserting $\overline{\text{PMaster}}$.
3. External system issues an External Request
4. Processor regains bus mastership, it $\overline{\text{EReq}}$ is de-asserted.

Figure 11-8 External Request

Mastership of the System interface returns to the processor, if the external agent will release the System interface by de-asserting $\overline{\textbf{EReq}}$. The processor does not accept a subsequent external request until it has completed the current request.

If there are no processor requests pending, the processor decides, based on its internal state, whether to accept the external request, or to issue a new processor request. The processor can issue a new processor request even if the external agent is requesting access to the System interface.

The external agent asserts $\overline{\textbf{EReq}}$ indicating that it wishes to begin an external request. The external agent then waits for the processor to signal that it is ready to accept this request by asserting $\overline{\textbf{PMaster}}$. The processor signals that it is ready to accept an external request based on the criteria listed below.

- The processor completes any processor request that is in progress.
- While waiting for the assertion of $\overline{\textbf{ERdy}}$ to issue a processor read request, the processor can accept an external request if the request is delivered to the processor one or more cycles before $\overline{\textbf{ERdy}}$ is asserted. System interface:external requests:overview
- While waiting for the assertion of $\overline{\textbf{ERdy}}$ to issue a processor write request, the processor can accept an external request provided the request is delivered to the processor one or more cycles before $\overline{\textbf{ERdy}}$ is asserted.
- If waiting for the response to a read request after the processor has made an uncompelled change to a slave state, the external agent can issue an external request before providing the read response data.

External Write Request

When an external agent issues a write request, the specified resource is accessed and the data is written to it. An external write request is complete after the word of data has been transmitted to the processor.

The only processor resource available to an external write request is the *Interrupt* register.

Read Response

A *read response* returns data in response to a processor read request, as shown in Figure 11-9. While a read response is technically an external request, it has one characteristic that differentiates it from all other external requests -- it does not perform System interface arbitration. For this reason, read responses are handled separately from all other external requests, and are simply called read responses.

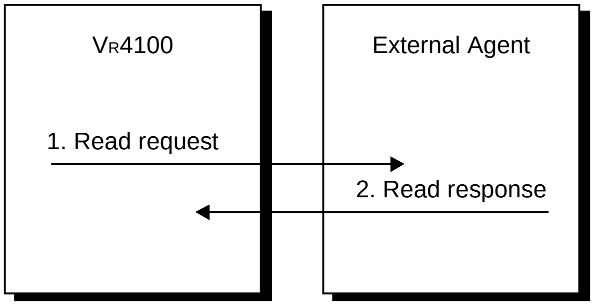


Figure 11-9 Read Response

11.5 Handling Requests

This section details the *sequence*, *protocol*, and *syntax* (See Terminology, in this chapter, for definitions of these terms) of both processor and external requests. The following system events are discussed:

- fetch miss
- uncached fetch miss
- load miss
- store miss
- uncached loads/stores
- CACHE operations

Fetch Miss

When the processor misses in the instruction cache on a fetch, it issues a read request for the cache line and waits for an external agent to provide this data in a read response.

Uncached Fetch Miss ★

When the processor misses in uncached area on a fetch, it issues a single read request for the cache line and waits for an external agent to provide this data in a read response.

Load Miss

When the processor misses in the data cache on a load, it issues a read request for the cache line and waits for an external agent to provide the data in a read response.

If the cache data to be replaced is in the dirty valid state, this data is written out to memory. The read starts after the dirty data is written out.

Store Miss

If the processor store misses in the data cache, it issues a read request to the external agent to retrieve the target cache line. After the target line has been retrieved by the external agent, it is written into the cache, and then updated with the store data.

If the cache data to be replaced is in the dirty valid state, this data is written out to memory. The read starts after the dirty data is written out.

When it is necessary 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. CACHE operations are described in Chapter 14.

Uncached Loads or Stores

When the processor performs an uncached load, it issues a read request, and waits for a single or block 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 or block transfer of data to the external agent.

CACHE Operations

The processor provides a variety of CACHE operations to maintain the state and contents of the primary caches. The processor can issue write requests during the execution of the CACHE operation instructions.

Summary of Parity Check Operation

Parity check operation are summarized in the Table on next page.

Note: When cache writeback, cache data parity error is checked and if error is detected, cache data is sent system bus and set errornouse bit.

When the bus error or parity error are detected in read data, even if that error is not caused desired data word, processor caused exception and cache line marked invalid.

Bus	Uncached Load	Uncached Store	Cache Fill	Cache Writeback	External Write Req.
Processor Data	From System Interface	Not Checked	From System Interface	Not Checked	NA
Cache Data Parity	NA	NA	Generated	Checked	NA
Cache Tag Parity	NA	NA	Generated	Checked	NA
System Interface Address/Data on Address Cycle	Generated Read Address	Generated Write Address	Generated Read Address	Generated Write Address	From External Agent
System Interface Address/Data Parity on Address Cycle	Generated	Generated	Generated	Generated	Not Checked
System Interface Command on Address Cycle	Generated Read Request	Generated Write Request	Generated Read Request	Generated Write Request	From External Agent
System Interface Command Parity on Address Cycle	Generated	Generated	Generated	Generated	Not Checked; Reported to the $\overline{\text{Fault}}$ pin
System Interface Address/Data on Data Cycle	From External Agent	From Processor	From External Agent	From Cache	From External Agent
System Interface Address/Data Parity on Data Cycle	Checked	Generated	Checked	Generated	Checked
System Interface Command on Data Cycle	From External Agent	From Processor	From External Agent	From Processor	From External Agent
System Interface Command Parity on Data Cycle	Not Checked; Reported to the $\overline{\text{Fault}}$ pin	Generated	Not Checked; Reported to the $\overline{\text{Fault}}$ pin	Generated	Not Checked; Reported to the $\overline{\text{Fault}}$ pin

11.6 Processor and External Request Protocols

The following sections contain a cycle-by-cycle description of the bus arbitration protocols for each type of processor and external request. Table 11-1 lists the abbreviations and definitions for each of the buses that are used in the timing diagrams that follow.

Scope	Abbreviation	Meaning
Global	Unsd	Unused
SysAD bus	Addr	Physical address
	Data <n>	Data element number n of a block of data
SysCmd bus	Cmd	An unspecified System interface command
	Read	A processor or external read request command
	Write	A processor or external write request command
	EOD	A data identifier for the last data element
	Data	A data identifier for any data element other than the last data element

Table 11-1 System Interface Requests

Processor Request Protocols

Processor request protocols described in this section include:

- read
- write

Note: In the timing diagrams, the two closely spaced, wavy vertical lines (such as those in SCycle 7, Figure 11-18) indicate one or more identical cycles.

Processor Read Request Protocol

A processor read request is issued by driving a read command on the **SysCmd** bus, driving a read address on the **SysAD** bus, and asserting $\overline{\textbf{PValid}}$. 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 makes an uncompelled change to slave state after the issue cycle of the read request by de-asserting the $\overline{\textbf{PMaster}}$ signal continuous. An external agent then returns the requested data through a read response.

Once in slave state (starting at cycle 5 in Figure 11-10), the external agent can return the requested data through a read response. The read response returns the requested data or, if the requested data could not be successfully retrieved, the read response returns an indication that the returned data is erroneous. If the returned data is erroneous, the processor takes a bus error exception.

Figure 11-10 illustrates a processor read request, coupled with an uncompelled change to slave state, that occurs as the read request is issued.

Note: Timings for the **SysADC** and **SysCmdP** buses are the same as those of the **SysAD** and **SysCmd** buses, respectively.

The following sequence describes the protocol for a processor read request (the numbered steps below corresponds to Figure 11-11).

1. With the System interface in master state, a processor read request is issued by driving a read command on the **SysCmd** bus and a read address on the **SysAD** bus.
2. At the same time, the processor asserts $\overline{\textbf{PValid}}$ for one cycle, indicating valid data is present on the **SysCmd** and the **SysAD** buses.

   Note: Only one processor read request can be pending at a time.
3. If $\overline{\textbf{ERdy}}$ is de-asserted, the issue cycle will be delayed.
4. The processor makes an uncompelled change to slave state at the next cycle of the read request by de-asserting the $\overline{\textbf{PMaster}}$ signal continuous.
5. The processor releases the **SysCmd** and the **SysAD** buses at the same SCycle of the de-assertion of $\overline{\textbf{PMaster}}$.
6. The external agent drives the **SysCmd** and the **SysAD** buses within the de-assertion of $\overline{\textbf{PMaster}}$. ★

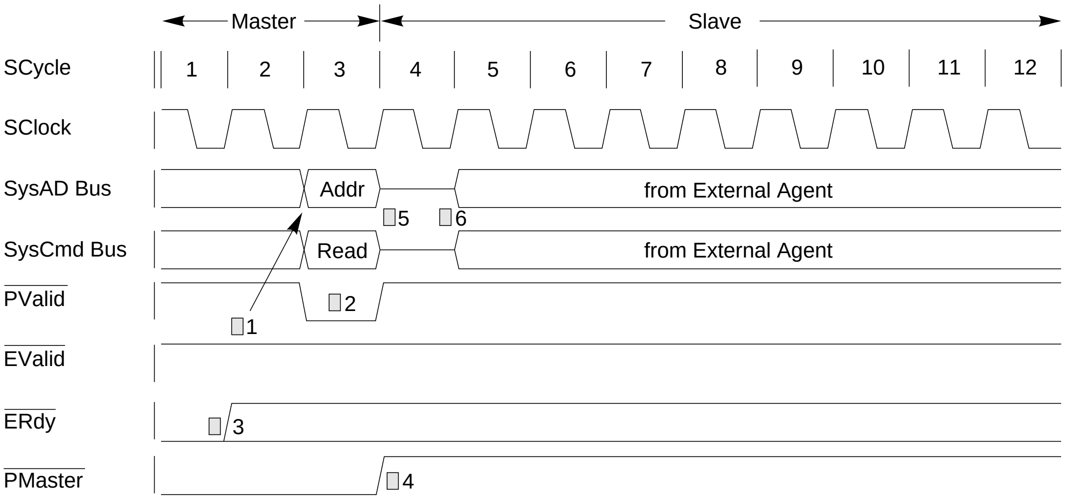


Figure 11-10 Processor Read Request

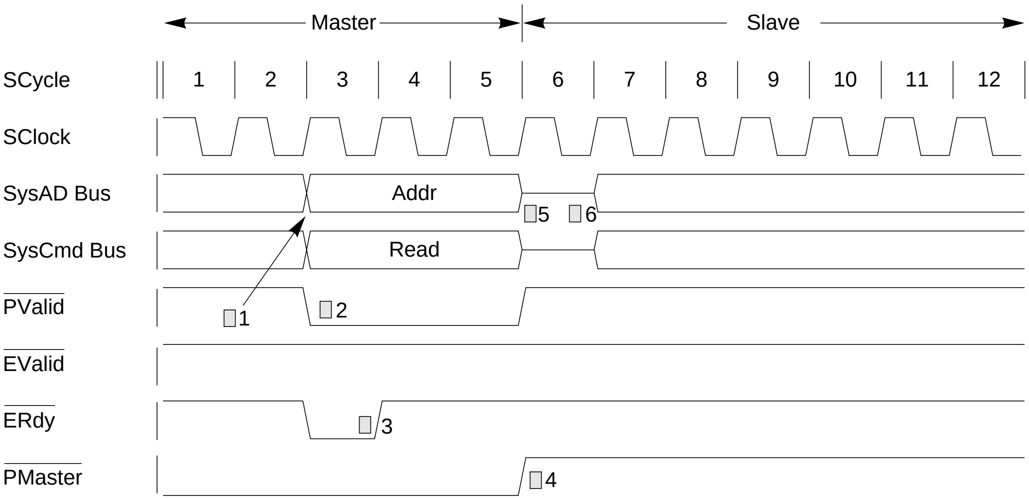


Figure 11-11 Delayed Processor Read Request

Processor Write Request Protocol

A processor write request is issued by driving a write command on the **SysCmd** bus, driving a write address on the **SysAD** bus, and asserting **$\overline{\text{PValid}}$** for one cycle. This is followed by driving the appropriate number of data identifiers on the **SysCmd** bus, driving data on the **SysAD** bus, and asserting **$\overline{\text{PValid}}$**. The processor drives data at the rate specified by the EP field in Config register. The data identifier associated with the last data cycle must contain a last data cycle indication.

1. A processor block write request with DD data rate is illustrated in Figure 11-13. The processor issues a write command on the **SysCmd** bus and a write address on the **SysAD** bus.
2. The processor asserts **$\overline{\text{PValid}}$**.
3. If **$\overline{\text{ERdy}}$** is de-asserted, the issue cycle will be delayed.
4. The processor drives a data identifier on the **SysCmd** bus and data on the **SysAD** bus.
5. The processor asserts **$\overline{\text{PValid}}$** for a number of cycles sufficient to transmit the block of data.
6. The data identifier associated with the last data cycle must contain a last data cycle indication.

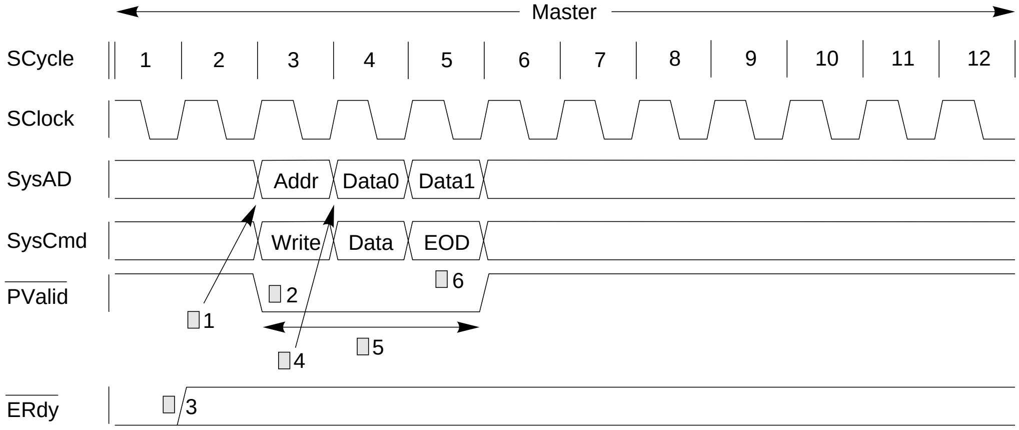


Figure 11-12 Processor Block Write Request with DD Data Rate

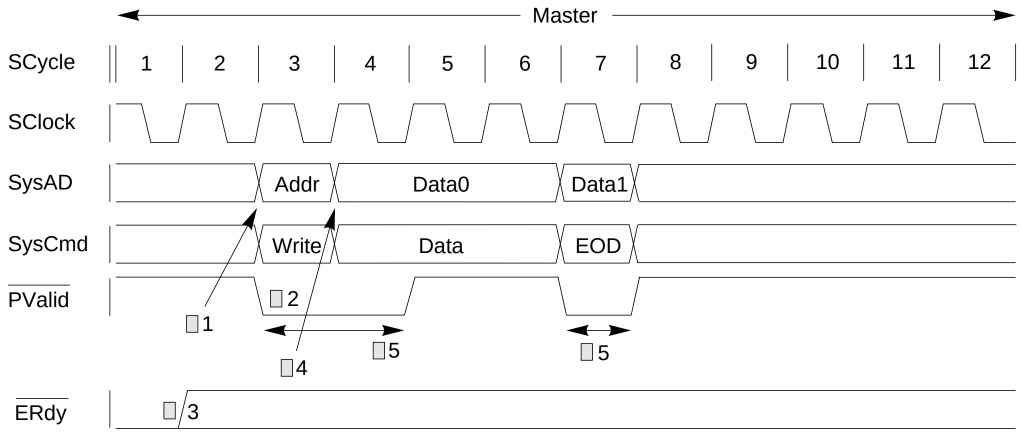


Figure 11-13   Processor Block Write Request with Dxx Data Rate

Processor Request Flow Control

The signal **$\overline{\text{ERdy}}$** may be used by an external agent to control the flow of processor read requests. While **$\overline{\text{ERdy}}$** is de-asserted the processor repeats the current address cycle until the external agent signals it is ready by asserting **$\overline{\text{ERdy}}$**.

1. **$\overline{\text{ERdy}}$** is deasserted 2 cycles before the processor cycle, so it will be delayed and the address cycles are not finished.
2. **$\overline{\text{ERdy}}$** is asserted 2 cycles before the processor cycle, so it will not be delayed and the address cycles are finished.

Use of **$\overline{\text{ERdy}}$** is illustrated in Figure 11-14.

★

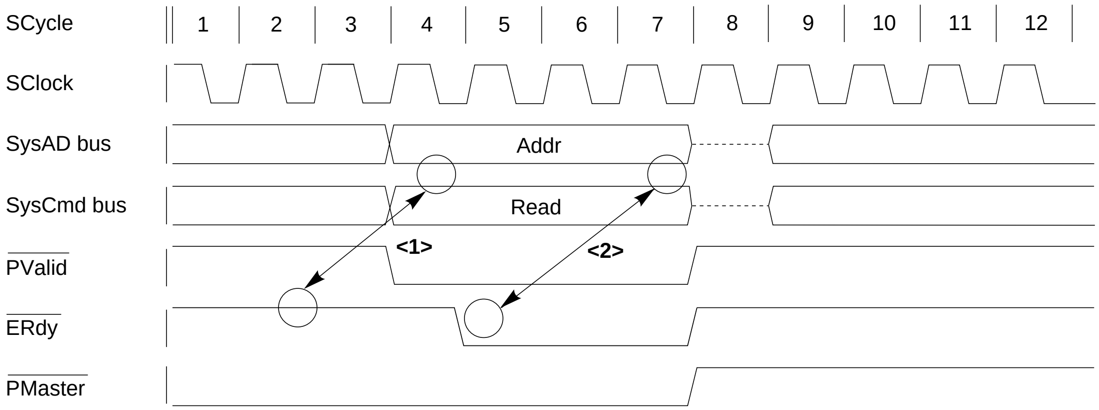


Figure 11-14   Delayed Processor Read Request

In Figures 11-14 and 11-15, **$\overline{\text{ERdy}}$** must be valid two cycles before the issue cycle.

The signal **$\overline{\text{ERdy}}$** is used by an external agent to control the flow of processor write requests, too. While **$\overline{\text{ERdy}}$** is de-asserted the processor repeats the current address cycle until the external agent signals it is ready by asserting **$\overline{\text{ERdy}}$**.

Two processor write requests in which the issue of the second is delayed for the assertion of **$\overline{\text{ERdy}}$** are illustrated in Figure 11-15.

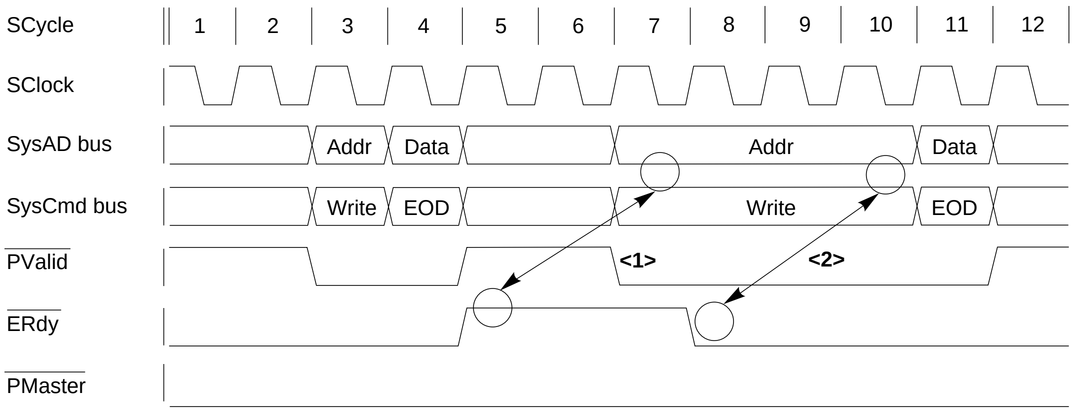


Figure 11-15 Two Processor Write Requests, with the Second Delayed

External Request Protocols

External requests can only be issued with the System interface in slave state. An external agent must assert **EReq** to arbitrate (see External Arbitration Protocol, below) for the System interface, and then wait for the processor to release the System interface to slave state. If the System interface is already in slave state -- that is, the processor has previously performed an uncompelled change to slave state -- the external agent can begin an external request immediately.

After issuing an external request, the external agent must return the System interface to master state, as described below.

Following the description of the arbitration protocol, this section describes the following external request protocols:

- write
- read response
- de-assertion of **EReq**

External Arbitration Protocol

System interface arbitration uses the signals $\overline{\textbf{EReq}}$, $\overline{\textbf{PMaster}}$, and $\overline{\textbf{PReq}}$ to arbitrate for bus mastership.

The transition from processor master to slave state is arbitrated by the processor using the system interface handshake signals $\overline{\textbf{EReq}}$ and $\overline{\textbf{PMaster}}$. When the processor is master, an external agent acquires control of the system interface by asserting $\overline{\textbf{EReq}}$, and waiting for the processor to de-assert $\overline{\textbf{PMaster}}$ when it is ready to relinquish the system interface.

When a processor read request is pending, the processor transitions to slave by de-asserting $\overline{\textbf{PMaster}}$, to allow an external agent to return read response data. The processor remains slave until an external agent issues a read response and also de-asserts $\overline{\textbf{EReq}}$, when it then transitions to master with the assertion of $\overline{\textbf{PMaster}}$.

When the processor is master, an external agent acquires control of the system interface by asserting $\overline{\textbf{EReq}}$, and waiting for the processor to de-assert $\overline{\textbf{PMaster}}$ when it is ready to relinquish the system interface. When the processor is ready to go slave state, it will de-assert $\overline{\textbf{PMaster}}$. If the processor is not asserting $\overline{\textbf{PReq}}$, it is guaranteed to release the bus two cycles after the external agent asserts $\overline{\textbf{EReq}}$. The system interface will return to master state when the issue of the external request is complete and $\overline{\textbf{EReq}}$ is de-asserted. Having asserted $\overline{\textbf{EReq}}$, an external agent must not de-assert $\overline{\textbf{EReq}}$ until the processor de-asserts $\overline{\textbf{PMaster}}$. Once the external agent has become master with an $\overline{\textbf{EReq}}$, it may continue to assert $\overline{\textbf{EReq}}$ until it is ready to relinquish the bus. Normally this would be on the same cycle as it signals end of data. Upon completing its transaction, it may continue to drive the bus and issue more transactions by continuing to hold the $\overline{\textbf{EReq}}$ line. That is, once the external agent has become bus master, it can remain master as long as it wants by simply continuing to assert $\overline{\textbf{EReq}}$.

If the processor is in slave state and needs the bus, it may assert the $\overline{\textbf{PReq}}$ line to let the external agent know that it wants the bus. When the processor sees $\overline{\textbf{EReq}}$ de-asserted, it resumes ownership, asserts the $\overline{\textbf{PMaster}}$ line, and can issue its command. The processor will become master two cycles after $\overline{\textbf{EReq}}$ is de-asserted.

Upon assertion of $\overline{\textbf{Reset}}$ or $\overline{\textbf{ColdReset}}$, the processor becomes bus master and the external agent must become slave.

This protocol guarantees that either the processor or the external agent is always bus master. The master should never 3-state the bus, except when giving up ownership of the bus under the rules of the protocol.

Figure 11-16 is a timing diagram of the arbitration protocol, in which slave and master states are shown. The arbitration cycle consists of the following steps:

1. The external agent asserts **$\overline{\text{EReq}}$** when it wishes to submit an external request.
2. The processor waits until it is ready to handle an external request, whereupon it de-asserts **$\overline{\text{PMaster}}$**.
3. The processor sets the **SysAD** and **SysCmd** buses to 3-state.
4. The external agent must begin driving the **SysAD** bus and the **SysCmd** bus one cycles after the de-assertion of **$\overline{\text{PMaster}}$**.
5. The external agent continues to assert **$\overline{\text{EReq}}$** until it is ready to relinquish the bus.
6. The external agent sets the **SysAD** and the **SysCmd** buses to 3-state at the completion of an external request.

The processor can start issuing a processor request one cycle after the external agent sets the bus to 3-state.

Note: Timings for the **SysADC** and **SysCmdP** buses are the same as those of the **SysAD** and **SysCmd** buses, respectively.

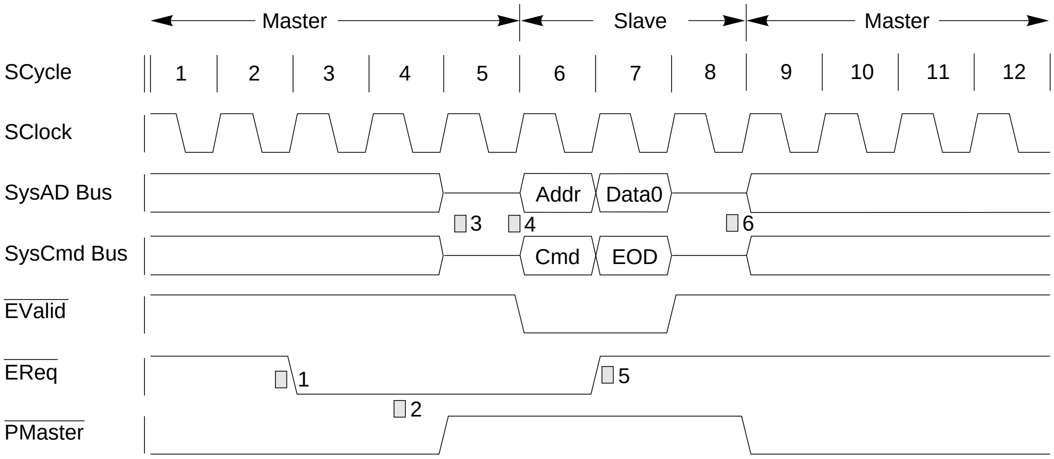


Figure 11-16 External Request Arbitration Protocol

External Write Request Protocol

External write requests are similar to a processor single write except that the signal **EValid** is asserted in place of the signal **PValid**.

An external write request consists of driving a write command on the **SysCmd** bus and a write address on the **SysAD** bus and asserting **EValid** for one cycle. This is followed by driving a data identifier on the **SysCmd** bus and data on the **SysAD** bus and asserting **EValid** for one more cycle. The data identifier associated with the data cycle must contain a last data cycle indication.

After the data cycle is issued, the write request is complete and the external agent releases the **SysCmd** and **SysAD** buses and allows the system interface to return to master state.

An external write request with the processor initially in master state is illustrated in Figure 11-17.

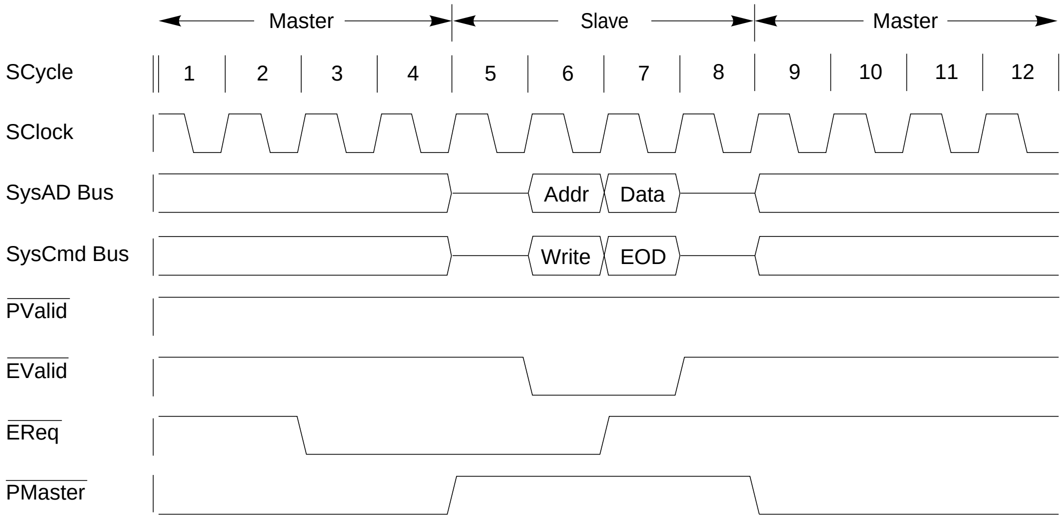


Figure 11-17 External Write Request Protocol

Note: The only writable resources are processor interrupts.

External Read Response Protocol

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, and then returning the data through a single data cycle or a number of data cycles sufficient for the requested data.

The read response is complete and the external agent releases the **SysCmd** and **SysAD** buses after the last data cycle is issued. The external agent then allows the processor to return to master state.

The data identifier associated with a data cycle may indicate that data transmitted during that cycle is erroneous; however, an external agent must return a data block of the correct size whether or not the data is erroneous. If a read response includes one or more erroneous data cycles, the processor takes a bus error.

Read response data can only be delivered to the processor when a processor read request is pending. If a read response is presented to the processor while no processor read is pending, the behavior of the processor is undefined.

A processor single read request followed by a read response is illustrated in Figure 11-18 and Figure 11-20. A read response for a processor block read with the system interface already in slave state is illustrated in Figure 11-20.

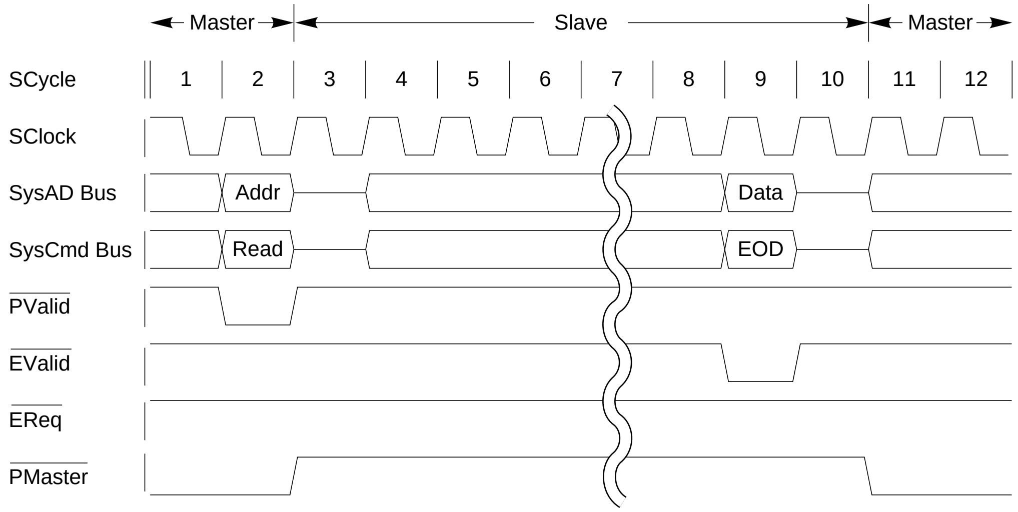


Figure 11-18 Single Read Request Followed by a Read Response

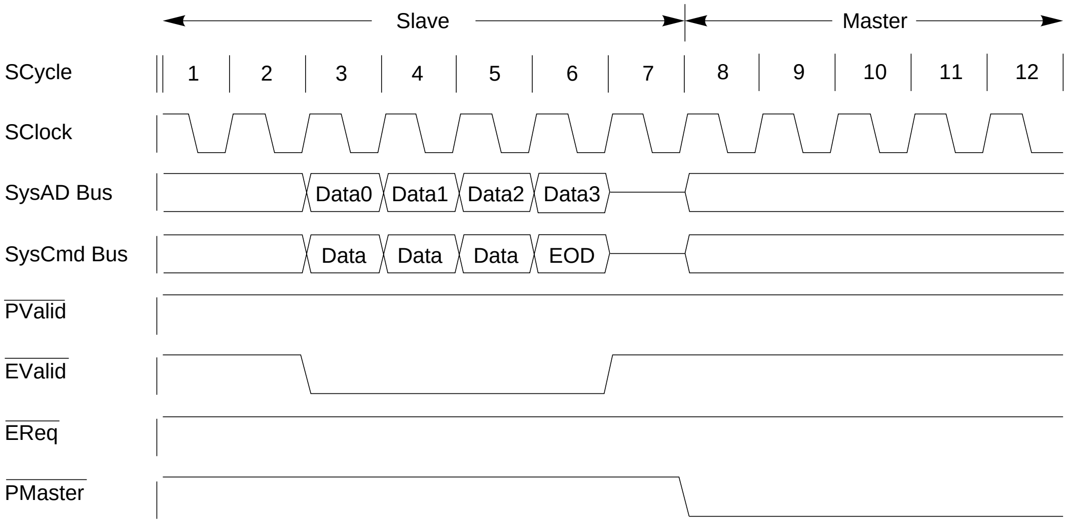


Figure 11-19 Block Read Response, with the System Interface Already in Slave State (I-cache block refill)

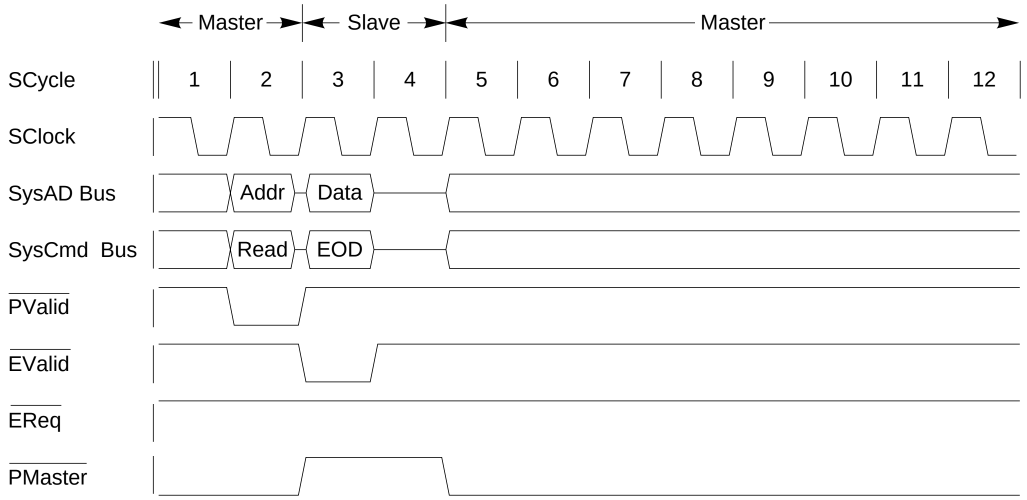


Figure 11-20 Processor Read Request Followed by Read Response

Successive Processor Writes

Successive processor write requests (a processor write followed by another processor write) are followed by next request without any wait states, as shown in Figure 11-21. Processor write request timing is the same as that described in the prior section, Processor Write Request Protocol.

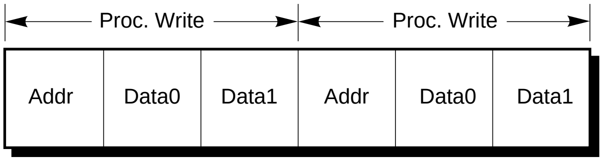
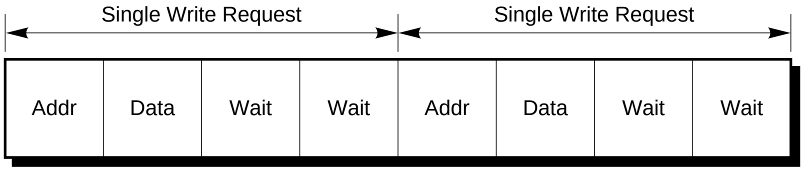


Figure 11-21 Processor Write Request Followed by Another Processor Block Write Request

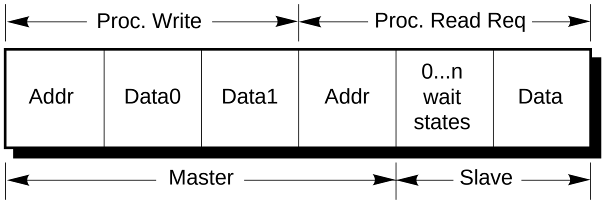


★

Figure 11-22 Processor Single Write Request Followed by Another Processor Write Request

Processor Write Followed by a Processor Read Request

A processor write request followed by a processor read request are shown in Figure 11-23. Timings are the same as those described in the prior sections, Processor Write Request Protocol, and Processor Read Request Protocol.



★

Figure 11-23 Processor Write Request Followed by a Processor Read Request

11.7 Data Flow Control

The system interface supports a maximum data rate of one word per cycle.

An external agent may deliver data to the processor at the maximum data rate of the System interface. The rate at which data is delivered to the processor can be controlled by the external agent, which asserts $\overline{\textbf{EValid}}$ whenever data is available. The processor only accepts cycles as valid when $\overline{\textbf{EValid}}$ is asserted and the **SysCmd** bus contains a data identifier; thereafter, the processor continues to accept data until it receives the data word tagged as the last one.

★ The rate at which the processor transmits data to an external agent is programmable through the EP field in *Config* Register. Data patterns are specified using the letters **D** and **x**, where **D** indicates a data cycle and **x** indicates maintaining last **D** cycle value. For example, a **Dxx** data pattern indicates a data rate of one doubleword every three cycles.

An external read response with a **DDx** data pattern (two doublewords every three cycles) is shown in Figure 11-24.

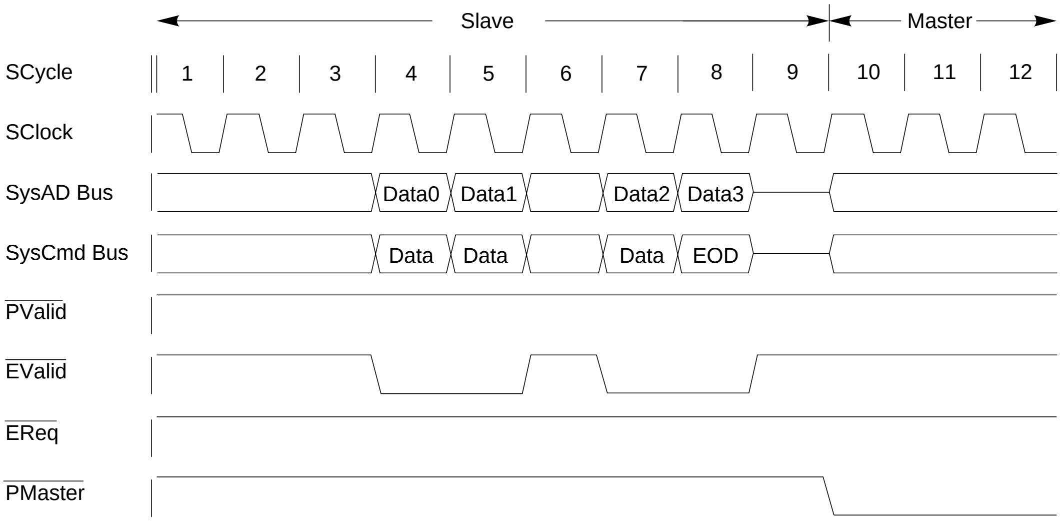


Figure 11-24 External Read Response with DDx Data Rate Pattern

The Accelerated Data mode can be selected by setting the *AD* field of *Config* Register. In AD mode, the processor insert no idle cycle between a cycle and a following cycle.
Figure 11-25 shows Accelerated data cycle.

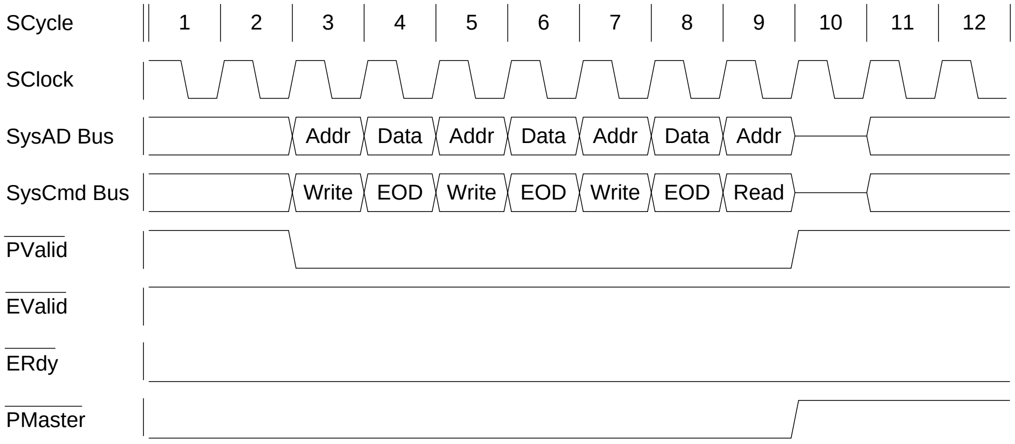


Figure 11-25 Accelerated Data Write Cycle

Independent Transmissions on the SysAD Bus

In most applications, the **SysAD** bus is a point-to-point connection, running from the processor to a bidirectional registered transceiver residing in an external agent. For these applications, the **SysAD** bus has only two possible drivers, the processor or the external agent.

Certain applications may require connection of additional drivers and receivers to the **SysAD** bus, to allow transmissions over the **SysAD** bus that the processor is not involved in. These are called *independent transmissions*. To effect an independent transmission, the external agent must coordinate control of the **SysAD** bus by using arbitration handshake signals and null requests.

An independent transmission on the **SysAD** bus follows this procedure:

1. The external agent requests mastership of the **SysAD** bus, to issue an external request.
2. The processor releases the System interface to slave state.
3. The external agent then allows the independent transmission to take place on the **SysAD** bus, making sure that $\overline{\textbf{EValid}}$ is not asserted while the transmission is occurring.
4. When the transmission is complete, the external agent must de-assert $\overline{\textbf{EReq}}$ to return the System interface to master state.

To implement multiple drivers, separate **Valid** lines are required to allow the non-processor chips to communicate.

System Endianness

★ The endianness of the system is set by the BigEndian Pin: byte order is big endian when the value is “1” and little endian when the value is “0”.

Software can set the reverse endian (*RE*) bit in the *Status* register to reverse the User mode byte ordering.

11.8 System Interface Cycle Time

The processor specifies minimum and maximum cycle counts for various processor transactions and for the processor response time to external requests. Processor requests themselves are constrained by the System interface request protocol, and request cycle counts can be determined by examining the protocol. The following System interface interactions can vary within minimum and maximum cycle counts:

- waiting period for the processor to release the System interface to slave state in response to an external request (*release latency*)

The remainder of this section describes and tabulates the minimum and maximum cycle counts for these System interface interactions.

Release Latency

Release latency is generally defined as the number of cycles the processor can wait to release the System interface to slave state for an external request. When no processor requests are in progress, internal activity can cause the processor to wait some number of cycles before releasing the System interface. Release latency is therefore more specifically defined as the number of cycles that occur between the assertion of $\overline{\textbf{EReq}}$ and the de-assertion of $\overline{\textbf{PMaster}}$.

There are three categories of release latency:

- Category 1: when the external request signal is asserted two cycles before the last cycle of a processor request.
- Category 2: when the external request signal is not asserted during a processor request, or is asserted during the last cycle of a processor request.

Table 11-2 summarizes the minimum and maximum release latencies for requests that fall into categories 1 and 2. Note that the maximum and minimum cycle count values are subject to change.

Category	Minimum PCycle	Maximum PCycle
1	TBD	TBD
2	TBD	TBD

Table 11-2 Release Latency for External Requests

11.9 System Interface Commands and Data Identifiers

System interface commands specify the nature and attributes of any System interface request; this specification is made during the address cycle for the request. System interface data identifiers specify the attributes of data transmitted during a System interface data cycle.

The following sections describe the syntax, that is, the bitwise encoding of System interface commands and data identifiers.

Reserved bits and reserved fields in the command or data identifier should be set to 1 for System interface commands and data identifiers associated with external requests. For System interface commands and data identifiers associated with processor requests, reserved bits and reserved fields in the command and data identifier are undefined.

Command and Data Identifier Syntax

System interface commands and data identifiers are encoded in 5 bits and are transmitted on the ★ **SysCmd** bus from the processor to an external agent, or from an external agent to the processor, during address and data cycles.

Bit 4 (the most-significant bit) of the **SysCmd** bus determines whether the current content of the **SysCmd** bus is a command or a data identifier and, therefore, whether the current cycle is an address cycle or a data cycle. For System interface commands, **SysCmd (4)** must be set to 0. For System interface data identifiers, **SysCmd (4)** must be set to 1.

For commands and data identifiers associated with external requests, reserved bits and reserved fields in the command or data identifier must be set to 1. For system interface commands and data identifiers associated with processor requests, reserved bits and reserved fields in the command or data identifier are undefined.

System Interface Command Syntax

This section describes the **SysCmd** bus encoding for System interface commands. Figure 11-26 shows a common encoding used for all System interface commands.

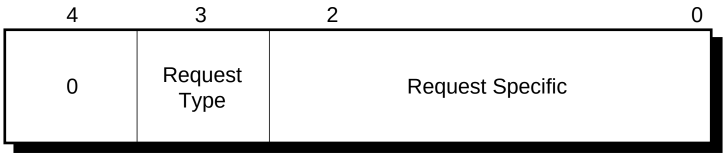


Figure 11-26   System Interface Command Syntax Bit Definition

SysCmd (4) must be set to 0 for all System interface commands.

SysCmd (3) specify the System interface request type which may be read, write, or null, as listed in Table 11-3.

SysCmd (3)	Command
0	Read Request
1	Write Request

Table 11-3 Encoding of SysCmd (3) for System Interface Commands

SysCmd (2:0) are specific to each type of request and are defined in each of the following sections.

Read Requests

For read requests, the encoding of the **SysCmd (4:0)** bits specify the specific attributes of the read. Figure 11-27 shows the format of a **SysCmd** read request.

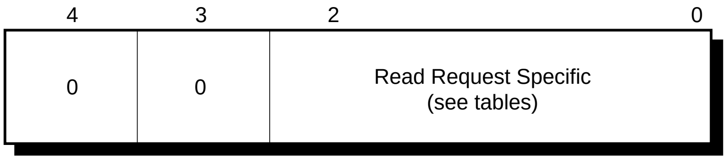


Figure 11-27 Read Request SysCmd Bus Bit Definition

Table 11-4 through Table 11-6 list the encodings of **SysCmd (2:0)** for read requests.

SysCmd (2)	Read Attributes
0	Single read
1	Block read

Table 11-4 Encoding of SysCmd (2) for Read Requests

SysCmd (1:0)	Read Block Size
0	2 words
1	4 words
2	Reserved
3	Reserved

Table 11-5 Encoding of SysCmd (1:0) for Block Read Request

SysCmd (1:0)	Read Data Size
0	1 byte valid (Byte)
1	2 bytes valid (Halfword)
2	3 bytes valid (Tribyte)
3	4 bytes valid (Word)

Table 11-6 Encoding of SysCmd (1:0) for Single Read Request

Write Requests

The encoding of **SysCmd (2:0)** for write commands is shown below.

Figure 11-28 shows the format of a **SysCmd** write request.

Table 11-7 lists the write attributes encoded in bits **SysCmd (2)**. Table 11-9 lists the block write replacement attributes encoded in bits **SysCmd (1:0)**. Table 11-8 lists the single write request bit encodings in **SysCmd (1:0)**.

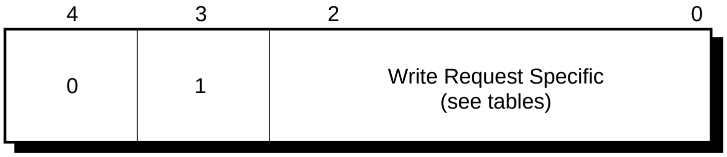


Figure 11-28   Write Request SysCmd Bus Bit Definition

SysCmd (2)	Write Attributes
0	Single write
1	Block write

Table 11-7   Encoding of SysCmd (2) for Write Requests

SysCmd (1:0)	Write Data Size
0	1 byte valid (Byte)
1	2 bytes valid (Halfword)
2	3 bytes valid (Tribyte)
3	4 bytes valid (Word)

Table 11-8   Encoding of SysCmd (1:0) for Single Write Request

SysCmd (1:0)	Write Block Size
0	2 words
1	4 words
2	Reserved
3	Reserved

Table 11-9   Encoding of SysCmd (1:0) for Block Write Request

System Interface Data Identifier Syntax

This section defines the encoding of the **SysCmd** bus for System interface data identifiers. Figure 11-29 shows a common encoding used for all System interface data identifiers.

4	3	2	1	0
1	Last Data	Resp Data	Err Data	Data Chk

Figure 11-29 Data Identifier SysCmd Bus Bit Definition

SysCmd (4) must be set to 1 for all System interface data identifiers.

Data Identifier Bit Definitions

SysCmd (3) marks the last data element.

SysCmd (2) indicates whether or not the data is response data, for both processor and external data identifiers. Response data is data returned in response to a read request.

SysCmd (1) indicates whether or not the data element is error free. Erroneous data contains an uncorrectable error and is returned to the processor, forcing a bus error. The processor delivers data with the good data bit de-asserted if a parity error is detected for a transmitted data item.

SysCmd (0) indicates to the processor whether or not to check the data and check bits for this data element.

Table 11-10 lists the encodings of **SysCmd (3:0)** for processor data identifiers. Table 11-11 lists the encodings 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 (will always be set to 1; there is no processor response data)
SysCmd (1)	Good Data Indication
0	Data is error free
1	Data is erroneous
SysCmd (0)	Reserved

Table 11-10 Processor Data Identifier Encoding of SysCmd (3:0)

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)	Good Data Indication
0	Data is error free
1	Data is erroneous
SysCmd (0)	Data Checking Enable
0	Check the data and check bits
1	Do not check the data and check bits

Table 11-11 External Data Identifier Encoding of SysCmd (3:0)

11.10 System Interface Addresses

System interface addresses are full 32-bit physical addresses presented on the **SysAD** bus during address cycles.

Addressing Conventions

Addresses associated with word, or partial word transactions are aligned for the size of the data element. The system uses the following address conventions:

- Addresses associated with block requests are aligned to double-word boundaries; that is, the low-order 3 bits of address are 0.
- Word requests set the low-order 2 bits of address to 0.
- Halfword requests set the low-order bit of address to 0.
- Byte, tribyte, requests use the byte address.

Subblock Ordering

All block read and write bus cycles are managed in subblock order. ★

Table 11-12 show the block read data address ordering. Table 11-13 show the block write data ordering. First read/write address are marked bold font.

Note: CACHE instruction for I cache used same ordering of D cache access.

target address lower 4 bits	I cache refill address lower 4 bits	D cache refill address lower 4 bits	uncache double word load address lower 4 bits
0	C - 8 - 4 - 0	8 - C - 0 - 4	0 - 4
4	8 - C - 0 - 4	C - 8 - 4 - 0	NA
8	4 - 0 - C - 8	0 - 4 - 8 - C	8 - C
C	0 - 4 - 8 - C	4 - 0 - C - 8	NA

Table 11-12 Block Read Data Address Ordering

target address lower 4 bits	I cache writeback address lower 4 bits	D cache writeback address lower 4 bits	uncache double word store address lower 4 bits
0	C - 8 - 4 - 0	8 - C - 0 - 4	0 - 4
4	8 - C - 0 - 4	C - 8 - 4 - 0	NA
8	4 - 0 - C - 8	0 - 4 - 8 - C	8 - C
C	0 - 4 - 8 - C	4 - 0 - C - 8	NA

Table 11-13 Block Write Data Address Ordering

[MEMO]

VR4100 Processor Interrupts

12

Four types of interrupt are available on the VR4100. These are:

- one non-maskable interrupt, NMI
- five external interrupts
- two software interrupts
- one timer interrupt

These are described in this chapter.

12.1 Nonmaskable Interrupt

The nonmaskable interrupt is signaled by asserting the $\overline{\textbf{NMI}}$ 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 **MasterOut**, as shown in Figure 12-1. 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.

Figure 12-1 shows the internal derivation of the **NMI** signal. The $\overline{\textbf{NMI}}$ pin is latched into an internal register by the rising edge of **MasterOut**. Bit 6 of the *Interrupt* register is then ORed with the inverted value of $\overline{\textbf{NMI}}$ to form the nonmaskable interrupt.Interrupt registerregisters, CPU:Interrupt

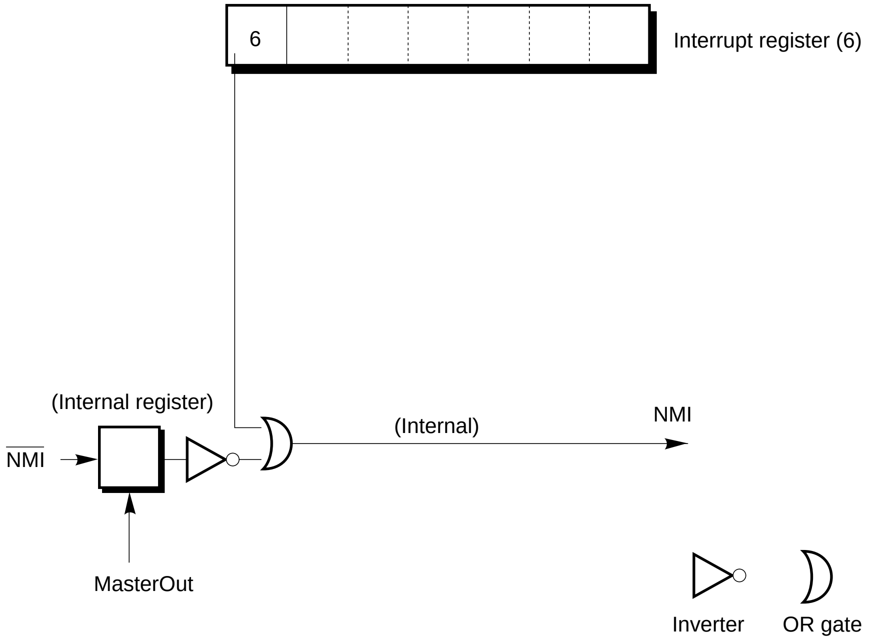


Figure 12-1 VR4100 Nonmaskable Interrupt Signal

12.2 External Interrupts

External interrupts are set by asserting the external interrupt pins **$\overline{\text{Int}}$ (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.

12.3 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 through the *IM*, *IE*, and *EXL* fields of the *Status* register.

12.4 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.

12.5 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 12-2 shows the mechanics of an external write to the *Interrupt* register, along with the nonmaskable interrupt described earlier.

Note: If the SysAD Parity Error or Bus Error detected on the external write cycles, contents of Interrupt registers are never changed.

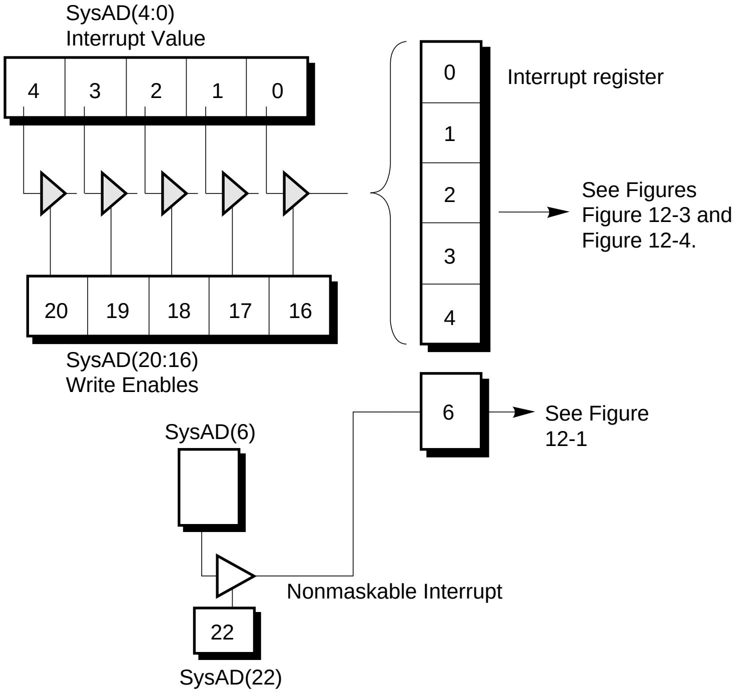


Figure 12-2   Interrupt Register Bits and Enables

Figure 12-3 shows how the VR4100 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 $\overline{\mathbf{Int}}$ **[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.

★

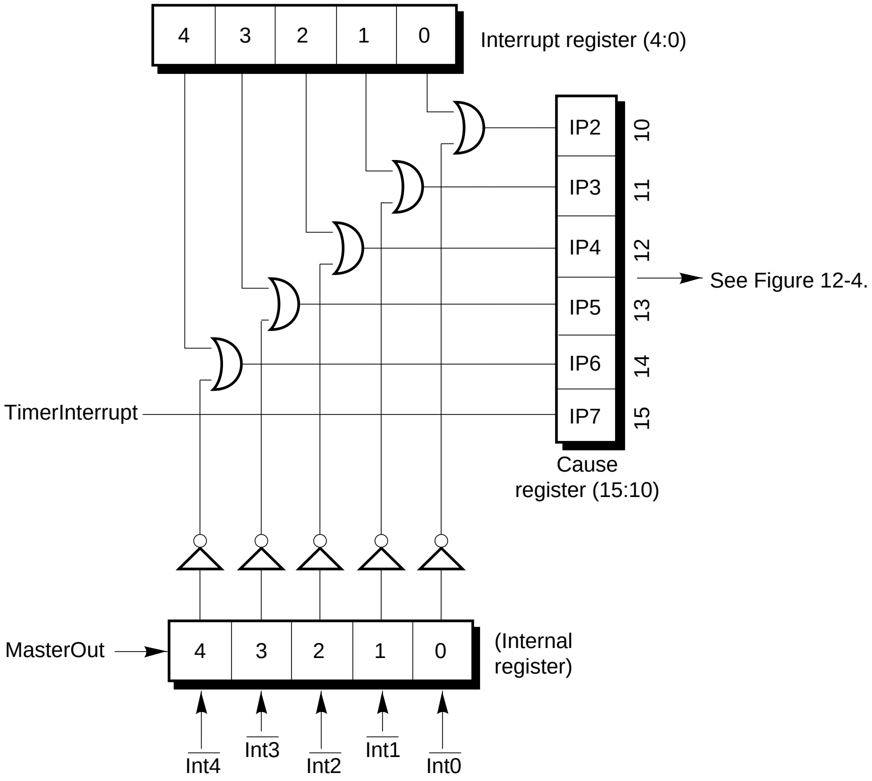


Figure 12-3 VR4100 Hardware Interrupt Signals

Figure 12-4 shows the masking of the VR4100 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 VR4100 interrupt signal. The *EXL* bit in the *Status* register also enables these interrupts.

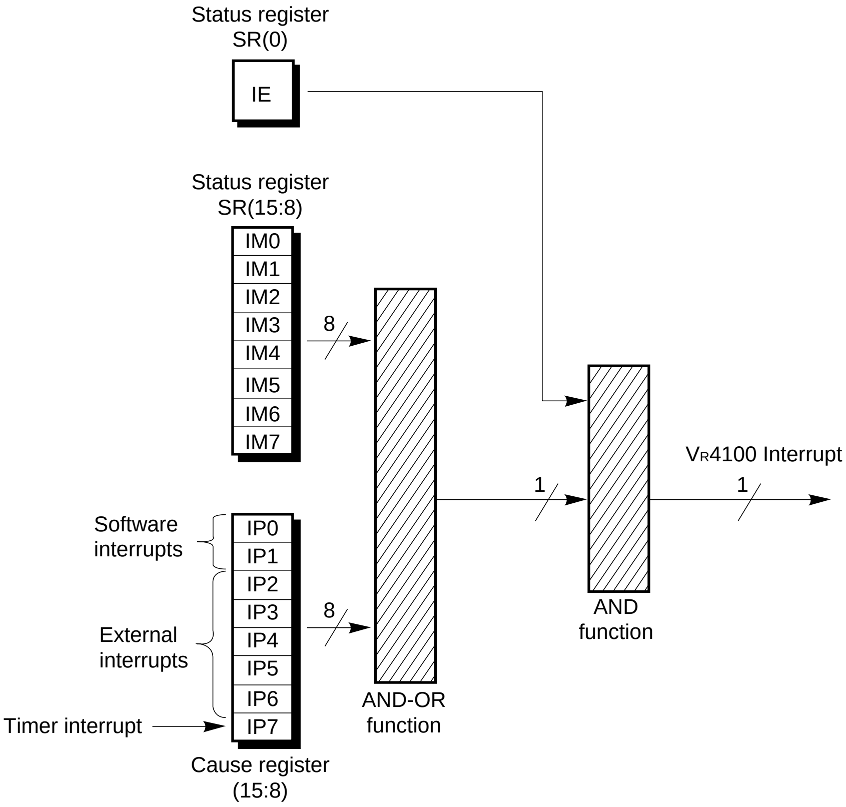


Figure 12-4   Masking of the VR4100 Interrupts

Electrical Characteristics

13

This chapter describes the processor electrical characteristics.

13.1 Electrical Characteristics

LVCMOS

V_R4100 will meet and exceed the JEDEC standard for low-voltage CMOS-compatible VLSI digital circuit (LVCMOS).

Maximum Ratings

(Operation beyond the limits set forth in this table may impair the useful life of the device.)

(T_A = 25 ^oC)

Parameter	Symbol	Test Conditions	Minimum	Typical	Maximum	Units
Supply Voltage	V_{DD}		-0.5		4.0	V
Input Voltage	V_I		-0.5		min (4.0, V_{DD} + 0.3)	V
Storage Temperature	Tstg		-65		150	^oC

Table 13-1 Maximum Ratings

Recommended Operating Conditions

Parameter	Symbol	Test Conditions	Minimum	Typical	Maximum	Units
Operating Ambient Temperature	T_A		-10		+70	^oC
Supply Voltage	V_{DD}		+2.2		+3.6	V

Table 13-2 Recommended Operating Conditions

DC Characteristics

(T_A = -10 to +70 °C, V_{DD} = 2.7 to 3.6 V)

Parameter	Symbol	Conditions	Minimum	Typical	Maximum	Units
Output HIGH Voltage	V_{OH}	I_{OH} = -2 mA I_{OH} = -20 µA	0.8V_{DD} V_{DD} - 0.1			V
Output LOW Voltage	V_{OL}	I_{OL} = 2 mA I_{OL} = 20 µA			0.4 0.1	V
Clock Output HIGH Voltage[3]	V_{OHC}	I_{OH} = -2 mA I_{OH} = -20 µA	0.8V_{DD} V_{DD} - 0.1			V
Clock Output LOW Voltage[3]	V_{OLC}	I_{OL} = 2 mA I_{OL} = 20 µA			0.4 0.1	V
Input HIGH Voltage[2]	V_{IH}		2.0		V_{DD} + 0.3	V
Input LOW Voltage[2]	V_{IL}		-0.3		0.3V_{DD}	V
MasterClock Input HIGH Voltage	V_{KH}		0.7V_{DD}		V_{DD} + 0.3	V
MasterClock Input LOW Voltage	V_{KL}		-0.3		0.3V_{DD}	V
Operating Current (the inside chip)	I_{DD}	V_{DD} = 3.3 V T_A = 25 °C		1f[1]	2f[1]	mA
Input Leakage	I_{LI}	V_{DD} = 3.6 V V_I = V_{DD}, 0 V			±5	µA
Output Leakage	I_{LO}	V_{DD} = 3.6 V V_O = V_{DD}, 0 V			±5	µA

Notes: (1) f = PClock Frequency (MHz).
(2) Except for MasterClock input.
(3) Applies to TClock and MasterOut output.

Table 13-3   DC Characteristics

Capacitance

(T_A = 25 °C, V_{DD} = 3.6 V)

Parameter	Symbol	Conditions	Minimum	Typical	Maximum	Units
Input Capacitance	C_I	f_C = 1 MHz			15	pF
Input/Output Capacitance	C_{IO}	f_C = 1 MHz			15	pF

Table 13-4   Capacitance

AC Characteristics

Note: All output timings are given assuming 40 pF of capacitive load. Output timings should be derated where appropriate as per the table below.

MasterClock and Clock Parameters

Parameter	Symbol	Conditions	Minimum	Typical	Maximum	Units
MasterClock High	t_{KKH}	Transition ≤ 5 ns	45			ns
MasterClock Low	t_{KKL}	Transition ≤ 5 ns	45			ns
MasterClock Freq[1]			1		10	MHz
MasterClock Period	t_{CYK}		100		1000	ns
Clock Jitter	t_{Jitter}				±2	ns
MasterClock Rise Time	t_{KR}				5	ns
MasterClock Fall Time	t_{KF}				5	ns
Note: (1) Operation of V_R4100 is only guaranteed with the Phase Lock Loop enabled. When the PLL work on self oscillation mode, this minimum frequency is 0 MHz (DC).						

Table 13-5   MasterClock and Clock Parameters (T_A = -10 to +70 °C,V_{DD} = 2.7 to 3.6V)

Parameter	Symbol	Conditions	Minimum	Typical	Maximum	Units
MasterClock High	t_{KKH}	Transition ≤ 5 ns	90			ns
MasterClock Low	t_{KKL}	Transition ≤ 5 ns	90			ns
MasterClock Freq[1]			1		5	MHz
MasterClock Period	t_{CYK}		200		1000	ns
Clock Jitter	t_{Jitter}				±2	ns
MasterClock Rise Time	t_{KR}				10	ns
MasterClock Fall Time	t_{KF}				10	ns

Note: (1) Operation of VR4100 is only guaranteed with the Phase Lock Loop enabled. When the PLL work on self oscillation mode, this minimum frequency is 0 MHz (DC).

Table 13-6 MasterClock and Clock Parameters ($2.2 \leq V_{DD} < 2.7$)

System Interface Parameters

Parameter	Symbol	Conditions	Minimum	Typical	Maximum	Units
Data Active Delay[1]	t_{DO}		3		15	ns
Data Inactive Delay[1]	t_{DM}		3			ns
Data Setup[1]	t_{DS}		5			ns
Data Hold[1]	t_{DH}		2			ns
MasterOut Rise Time	t_{MOR}				6	ns
MasterOut Fall Time	t_{MOF}				6	ns
MasterOut High Time	t_{MOH}		50			ns
MasterOut Low Time	t_{MOL}		50			ns
TClock Rise Time	t_{TCR}				3	ns
TClock Fall Time	t_{TCF}				3	ns
TClock High Time	t_{TCH}		10			ns
TClock Low Time	t_{TCL}		10			ns

Notes: (1) See the *Figure 13-5 SysAD Data*.
(2) Capacitive load for all output timings is 40 pF.
(3) Data Output, Data Setup, and Data Hold apply to all logic signals driven out of or driven into the VR4100 on the system interface. Clocks are specified separately.
(4) TClock.

Table 13-7 System Interface Parameters (V_{DD} = 2.7 to 3.6 V)

Parameter	Symbol	Conditions	Minimum	Typical	Maximum	Units
Data Active Delay[1]	t_{DO}		3		15	ns
Data Inactive Delay[1]	t_{DM}		3			ns
Data Setup[1]	t_{DS}		5			ns
Data Hold[1]	t_{DH}		2			ns
MasterOut Rise Time	t_{MOR}				TBD	ns
MasterOut Fall Time	t_{MOF}				TBD	ns
MasterOut High Time	t_{MOH}		TBD			ns
MasterOut Low Time	t_{MOL}		TBD			ns
TClock Rise Time	t_{TCR}				TBD	ns
TClock Fall Time	t_{TCF}				TBD	ns
TClock High Time	t_{TCH}		TBD			ns
TClock Low Time	t_{TCL}		TBD			ns

Notes: (1) See the *Figure 13-5 SysAD Data*.
(2) Capacitive load for all output timings is 40 pF.
(3) Data Output, Data Setup, and Data Hold apply to all logic signals driven out of or driven into the V_R4100 on the system interface. Clocks are specified separately.
(4) TClock.

Table 13-8 System Interface Parameters $(2.2 \le V_{DD} < 2.7)$

Capacitive Load Deration

Parameter	Symbol	Minimum	Typical	Maximum	Units
Load Deration	CLD			5	ns/20 pF

Table 13-9 Capacitive Load Deration

Timing Diagrams

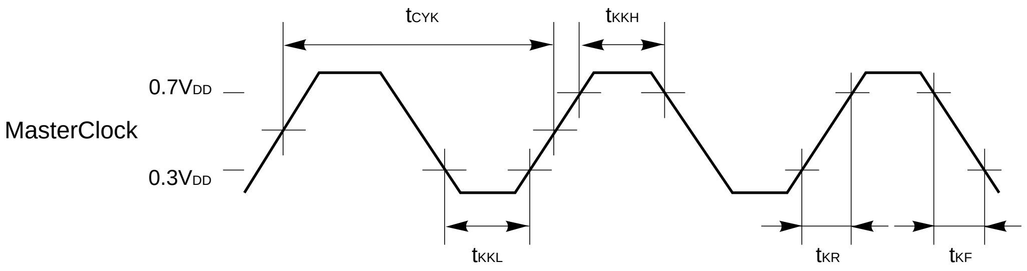


Figure 13-1 MasterClock

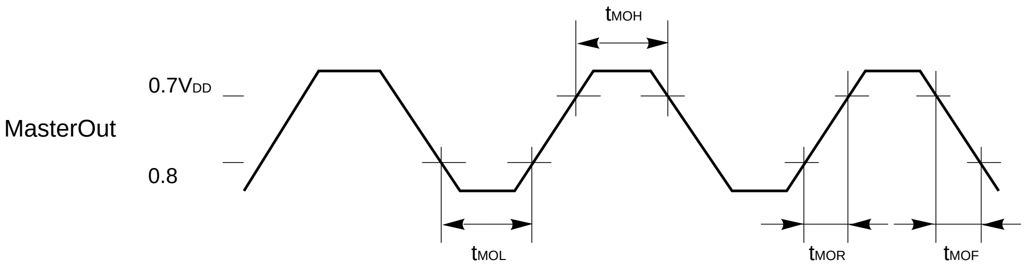


Figure 13-2 MasterOut

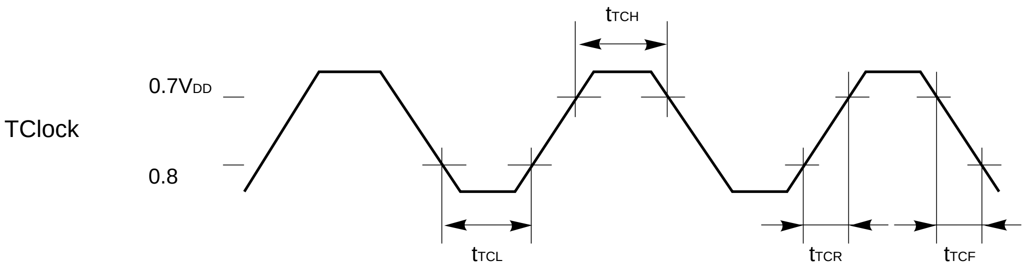


Figure 13-3 TClock

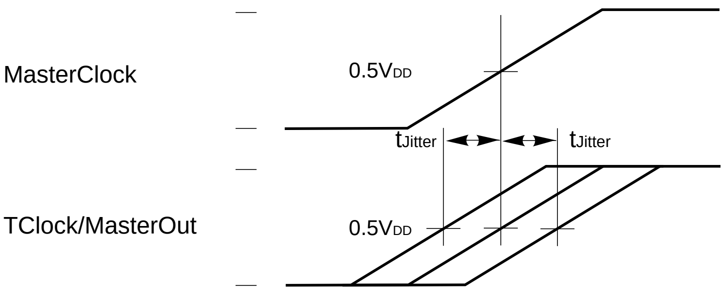


Figure 13-4   Clock Jitter

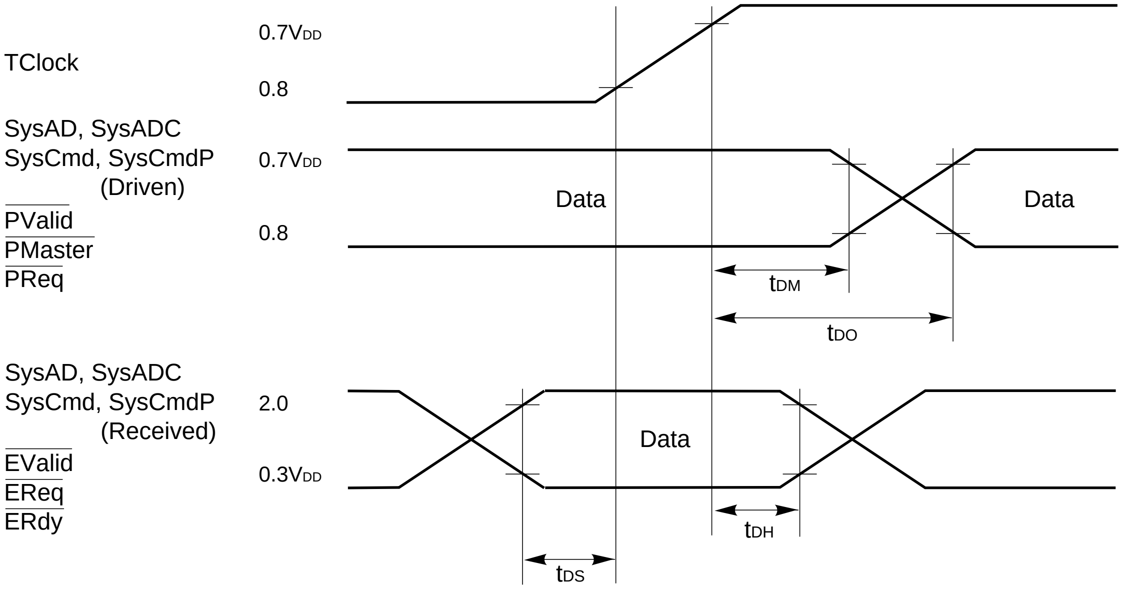


Figure 13-5   System Interface

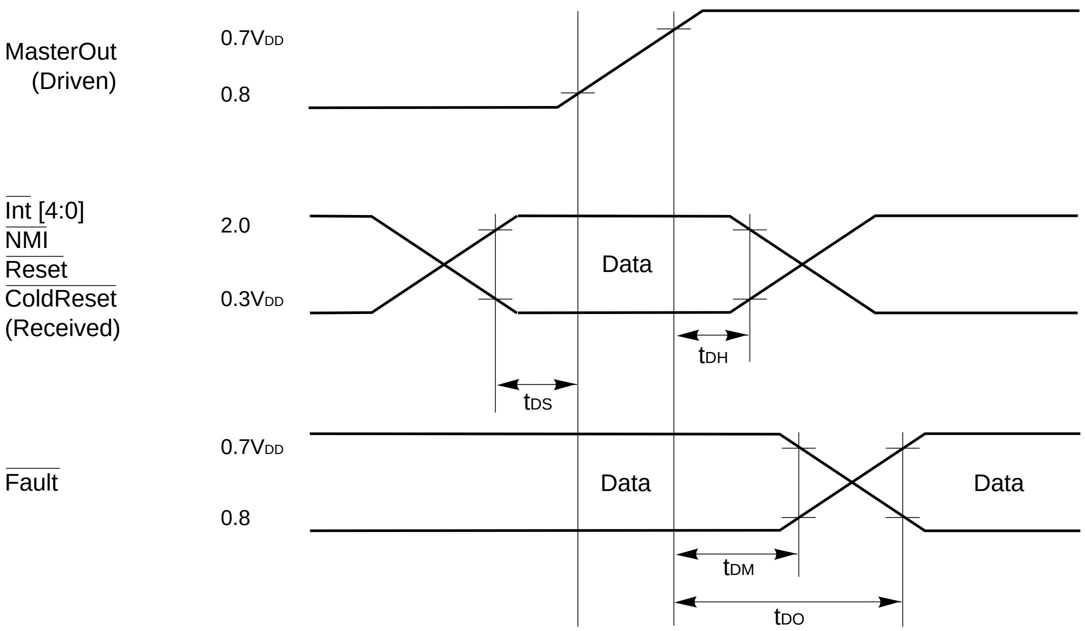


Figure 13-6 $\overline{\text{Int}}$ [4:0], $\overline{\text{NMI}}$, $\overline{\text{Reset}}$, $\overline{\text{ColdReset}}$ and $\overline{\text{Fault}}$ Interface

CPU Instruction Set Details

14

This chapter provides a detailed description of the operation of each VR4100 instruction in both 32- and 64-bit modes. The instructions are listed in alphabetical order.

Exceptions that may occur due to the execution of each instruction are listed after the description of each instruction. Descriptions of the immediate cause and manner of handling exceptions are omitted from the instruction descriptions in this chapter.

Figures at the end of this chapter list the bit encoding for the constant fields of each instruction, and the bit encoding for each individual instruction is included with that instruction.

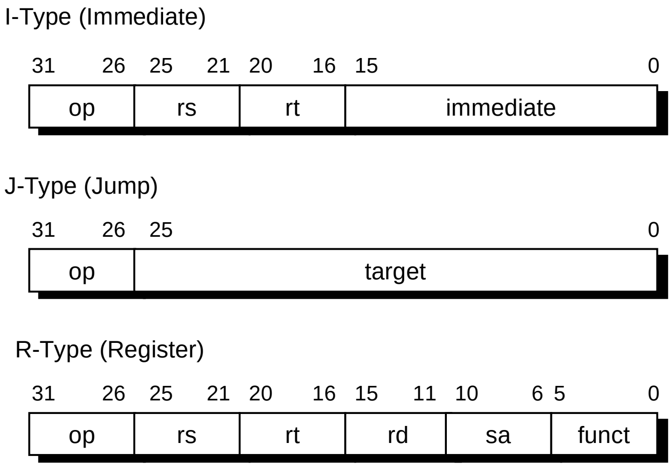
14.1 Instruction Classes

CPU instructions are divided into the following classes:

- **Load** and **Store** instructions move data between memory and general registers. They are all I-type instructions, since the only addressing mode supported is *base register + 16-bit immediate offset*.
- **Computational** instructions perform arithmetic, logical and shift operations on values in registers. They occur in both R-type (both operands are registers) and I-type (one operand is a 16-bit immediate) formats.
- **Jump** and **Branch** instructions change the control flow of a program. Jumps are always made to absolute 26-bit word addresses (J-type format), or register addresses (R-type), for returns and dispatches. Branches have 16-bit offsets relative to the program counter (I-type). **Jump and Link** instructions save their return address in register *31*.
- ★ **Coprocessor zero** (CP0) instructions manipulate the memory management and exception handling facilities of the processor.
- **Special** instructions perform a variety of tasks, including movement of data between special and general registers, trap, and breakpoint. They are always R-type.

14.2 Instruction Formats

Every CPU instruction consists of a single word (32 bits) aligned on a word boundary and the major instruction formats are shown in Figure 14-1.



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

Figure 14-1   CPU Instruction Formats

14.3 Instruction Notation Conventions

In this chapter, all variable subfields in an instruction format (such as *rs, rt, immediate*, etc.) are shown in lowercase names.

For the sake of clarity, we sometimes use an alias for a variable subfield in the formats of specific instructions. For example, we use *rs = base* in the format for load and store instructions. Such an alias is always lower case, since it refers to a variable subfield.

Figures with the actual bit encoding for all the mnemonics are located at the end of this chapter, and the bit encoding also accompanies each instruction.

In the instruction descriptions that follow, the *Operation* section describes the operation performed by each instruction using a high-level language notation. The VR4100 can operate as either a 32- or 64-bit microprocessor and the operation for both modes is included with the instruction description.

Special symbols used in the notation are described in Table 14-1.

Symbol	Meaning
<-	Assignment.
||	Bit string concatenation.
x^y	Replication of bit value *x* into a *y*-bit string. Note: *x* is always a single-bit value.
$x_{y:z}$	Selection of bits *y* through *z* of bit string *x*. Little-endian bit notation is always used. If *y* is less than *z*, this expression is an empty (zero length) bit string.
+	2's complement or floating-point addition.
-	2's complement or floating-point subtraction.
*	2's complement or floating-point multiplication.
div	2's complement integer division.
mod	2's complement modulo.
/	Floating-point division.
<	2's complement less than comparison.
and	Bit-wise logical AND.
or	Bit-wise logical OR.
xor	Bit-wise logical XOR.
nor	Bit-wise logical NOR.
GPR [*x*]	General-Register *x*. The content of GPR [0] is always zero. Attempts to alter the content of GPR [0] have no effect.
CPR [*z, x*]	Coprocessor unit *z*, general register *x*.
CCR [*z, x*]	Coprocessor unit *z*, control register *x*.
COC [*z*]	Coprocessor unit *z* condition signal.
BigEndianMem	Big-endian mode as configured at reset (0 -> Little, 1 -> Big). Specifies the endianness of the memory interface (see LoadMemory and StoreMemory), and the endianness of Kernel and Supervisor mode execution.
ReverseEndian	Signal to reverse the endianness of load and store instructions. This feature is available in User mode only, and is effected by setting the *RE* bit of the *Status* register. Thus, ReverseEndian may be computed as (SR_{25} and User mode).
BigEndianCPU	The endianness for load and store instructions (0 -> Little, 1 -> Big). In User mode, this endianness may be reversed by setting SR_{25}. Thus, BigEndianCPU may be computed as BigEndianMem XOR ReverseEndian.
T + *i*:	Indicates the time steps between operations. Each of the statements within a time step are defined to be executed in sequential order (as modified by conditional and loop constructs). Operations which are marked *T + i*: are executed at instruction cycle *i* relative to the start of execution of the instruction. Thus, an instruction which starts at time *j* executes operations marked T + *i:* at time *i + j*. The interpretation of the order of execution between two instructions or two operations which execute at the same time should be pessimistic; the order is not defined.

Table 14-1 CPU Instruction Operation Notations

Instruction Notation Examples

The following examples illustrate the application of some of the instruction notation conventions:

Example #1:

$$\text{GPR [rt]} \leftarrow \text{immediate} \,\|\, 0^{16}$$

Sixteen zero bits are concatenated with an immediate value (typically 16 bits), and the 32-bit string (with the lower 16 bits set to zero) is assigned to General-Purpose Register *rt*.

Example #2:

$$(\text{immediate}_{15})^{16} \,\|\, \text{immediate}_{15...0}$$

Bit 15 (the sign bit) of an immediate value is extended for 16 bit positions, and the result is concatenated with bits 15 through 0 of the immediate value to form a 32-bit sign extended value.

14.4 Load and Store Instructions

In the VR4100 implementation, the instruction immediately following a load may use the loaded contents of the register. In such cases, the hardware *interlocks*, requiring additional real cycles, so scheduling load delay slots is still desirable, although not required for functional code. ★

In the load and store descriptions, the functions listed in Table 14-2 are used to summarize the handling of virtual addresses and physical memory.

Function	Meaning
Address Translation	Uses the TLB to find the physical address given the virtual address. The function fails and an exception is taken if the required translation is not present in the TLB.
Load Memory	Uses the cache and main memory to find the contents of the word containing the specified physical address. The low-order three bits of the address and the *Access Type* field indicates which of each of the four bytes within the data word need to be returned. If the cache is enabled for this access, the entire word is returned and loaded into the cache.
Store Memory	Uses the cache, write buffer, and main memory to store the word or part of word specified as data in the word containing the specified physical address. The low-order three bits of the address and the *Access Type* field indicates which of each of the four bytes within the data word should be stored.

Table 14-2   Load and Store Common Functions

As shown in Table 14-3, the *Access Type* field indicates the size of the data item to be loaded or stored. Regardless of access type or byte-numbering order (endianness), the address specifies the byte which has the smallest byte address in the addressed field. For a big-endian machine, this is the leftmost byte and contains the sign for a 2's complement number; for a little-endian machine, this is the rightmost byte.

Access Type Mnemonic	Value	Meaning
DOUBLEWORD	7	8 bytes (64 bits)
SEPYIBYTE	6	7 bytes (56 bits)
SEXTIBYTE	5	6 bytes (48 bits)
QUINTIBYTE	4	5 bytes (40 bits)
WORD	3	4 bytes (32 bits)
TRIPLEBYTE	2	3 bytes (24 bits)
HALFWORD	1	2 bytes (16 bits)
BYTE	0	1 byte (8 bits)

Table 14-3 Access Type Specifications for Loads/Stores

The bytes within the addressed doubleword which are used can be determined directly from the access type and the three low-order bits of the address.

14.5 Jump and Branch Instructions

All jump and branch instructions have an architectural delay of exactly one instruction. That is, the instruction immediately following a jump or branch (that is, occupying the delay slot) is always executed while the target instruction is being fetched from storage. A delay slot may not itself be occupied by a jump or branch instruction; however, this error is not detected and the results of such an operation are undefined.

If an exception or interrupt prevents the completion of a legal instruction during a delay slot, the hardware sets the *EPC* register to point at the jump or branch instruction that precedes it. When the code is restarted, both the jump or branch instructions and the instruction in the delay slot are reexecuted.

Because jump and branch instructions may be restarted after exceptions or interrupts, they must be restartable. Therefore, when a jump or branch instruction stores a return link value, register *31* (the register in which the link is stored) may not be used as a source register.

Since instructions must be word-aligned, a **Jump Register** or **Jump and Link Register** instruction must use a register which contains an address whose two low-order bits are zero. If these low-order bits are not zero, an address exception will occur when the jump target instruction is subsequently fetched.

14.6 System Control Coprocessor (CP0) Instructions

There are some special limitations imposed on operations involving CP0 that is incorporated within the CPU. Although load and store instructions to transfer data to/from coprocessors and to move control to/from coprocessor instructions are generally permitted by the MIPS architecture, CP0 is given a somewhat protected status since it has responsibility for exception handling and memory management. Therefore, the move to/from coprocessor instructions are the only valid mechanism for writing to and reading from the CP0 registers.

Several CP0 instructions are defined to directly read, write, and probe TLB entries and to modify the operating modes in preparation for returning to User mode or interrupt-enabled states.

ADD Add ADD

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	ADD 100000
6	5	5	5	5	6

Format:

ADD rd, rs, rt

Description:

The contents of general register *rs* and the contents of general register *rt* are added to form the result. The result is placed into general register *rd*. In 64-bit mode, the operands must be valid sign-extended, 32-bit values.

An overflow exception occurs if the carries out of bits 30 and 31 differ (2● s complement overflow). The destination register *rd* is not modified when an integer overflow exception occurs.

Operation:

```
T:   temp <- GPR [rs] + GPR [rt]
     GPR [rd] <- $(\mathrm{temp}_{31})^{32}$ || $\mathrm{temp}_{31...0}$
```

Exceptions:

Integer overflow exception

ADDI Add Immediate ADDI

31 26	25 21	20 16	15 0
ADDI 0 0 1 0 0 0	rs	rt	immediate
6	5	5	16

Format:

ADDI rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and added to the contents of general register *rs* to form the result. The result is placed into general register *rt.* In 64-bit mode, the operand must be valid sign-extended, 32-bit values.

An overflow exception occurs if carries out of bits 30 and 31 differ (2● s complement overflow). The destination register *rt* is not modified when an integer overflow exception occurs.

Operation:

```
T:    temp <- GPR [rs] + $(immediate_{15})^{48}$ || $immediate_{15...0}$
      GPR [rt] <- $(temp_{31})^{32}$ || $temp_{31...0}$
```

Exceptions:

Integer overflow exception

ADDIU Add Immediate Unsigned ADDIU

31 26	25 21	20 16	15 0
ADDIU 0 0 1 0 0 1	rs	rt	immediate
6	5	5	16

Format:

ADDIU rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and added to the contents of general register *rs* to form the result. The result is placed into general register *rt.* No integer overflow exception occurs under any circumstances. In 64-bit mode, the operand must be valid sign-extended, 32-bit values.

The only difference between this instruction and the ADDI instruction is that ADDIU never causes an overflow exception.

Operation:

```
T:   temp <- GPR [rs] + $(\text{immediate}_{15})^{48}$ || $\text{immediate}_{15...0}$
     GPR [rt] <- $(\text{temp}_{31})^{32}$ || $\text{temp}_{31...0}$
```

Exceptions:

None

ADDU Add Unsigned ADDU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	ADDU 1 0 0 0 0 1
6	5	5	5	5	6

Format:

ADDU rd, rs, rt

Description:

The contents of general register *rs* and the contents of general register *rt* are added to form the result. The result is placed into general register *rd*. No overflow exception occurs under any circumstances. In 64-bit mode, the operands must be valid sign-extended, 32-bit values.

The only difference between this instruction and the ADD instruction is that ADDU never causes an overflow exception.

Operation:

```
T:    temp <- GPR [rs] + GPR [rt]
      GPR [rd] <- $(\text{temp}_{31})^{32}$ || $\text{temp}_{31...0}$
```

Exceptions:

None

AND And AND

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	AND 1 0 0 1 0 0
6	5	5	5	5	6

Format:

AND rd, rs, rt

Description:

The contents of general register *rs* are combined with the contents of general register *rt* in a bit-wise logical AND operation. The result is placed into general register *rd*.

Operation:

```
T:    GPR [rd] <- GPR [rs] and GPR [rt]
```

Exceptions:

None

ANDI And Immediate ANDI

31 26	25 21	20 16	15 0
ANDI 0 0 1 1 0 0	rs	rt	immediate
6	5	5	16

Format:

ANDI rt, rs, immediate

Description:

The 16-bit *immediate* is zero-extended and combined with the contents of general register *rs* in a bit-wise logical AND operation. The result is placed into general register *rt*.

Operation:

```
T:    GPR [rt] <- $0^{48}$ || (immediate and GPR $[rs]_{15...0}$)
```

Exceptions:

None

BC0F Branch On Coprocessor 0 False BC0F

31 26	25 21	20 16	15 0
COP0 0 1 0 0 0 0	BC 0 1 0 0 0	BCF 0 0 0 0 0	offset
6	5	5	16

Format:

BC0F offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If coprocessor 0's condition signal (CpCond: Status register bit-18 CHfield), as sampled during the previous instruction, is false, then the program branches to the target address with a delay of one instruction.

Because the condition line is sampled during the previous instruction, there must be at least one instruction between this instruction and a coprocessor instruction that changes the condition line.

Operation:

```
T-1:  condition <- not $SR_{18}$
T:    target <- $(\text{offset}_{15})^{46}$ || offset || $0^{2}$
T+1:  if condition then
             PC <- PC + target
      endif
```

Exceptions:

Coprocessor unusable exception

BC0FL Branch On Coprocessor 0 False Likely BC0FL

31 26	25 21	20 16	15 0
COP0 0 1 0 0 0 0	BC 0 1 0 0 0	BCFL 0 0 0 1 0	offset
6	5	5	16

Format:

BC0FL offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of coprocessor 0's condition line, as sampled during the previous instruction, is false, the target address is branched to with a delay of one instruction.

If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

Because the condition line is sampled during the previous instruction, there must be at least one instruction between this instruction and a coprocessor instruction that changes the condition line.

Operation:

```
T-1:  condition <- not $SR_{18}$
T:    target <- $(offset_{15})^{46}$ || offset || $0^2$
T+1:  if condition then
          PC <- PC + target
      else
          NullifyCurrentInstruction
      endif
```

Exceptions:

Coprocessor unusable exception

BC0T Branch On Coprocessor 0 True BC0T

31 26	25 21	20 16	15 0
COP0 0 1 0 0 0 0	BC 0 1 0 0 0	BCT 0 0 0 0 1	offset
6	5	5	16

Format:

BC0T offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the coprocessor 0's condition signal (CpCond: Status register bit-18 CHfield) is true, then the program branches to the target address, with a delay of one instruction.

Because the condition line is sampled during the previous instruction, there must be at least one instruction between this instruction and a coprocessor instruction that changes the condition line.

Operation:

```
T-1:  condition <- $SR_{18}$
T:    target <- $(offset_{15})^{46}$ || offset || $0^{2}$
T+1:  if condition then
          PC <- PC + target
      endif
```

Exceptions:

Coprocessor unusable exception

BC0TL Branch On Coprocessor 0 True Likely BC0TL

31 26	25 21	20 16	15 0
COP0 0 1 0 0 0 0	BC 0 1 0 0 0	BCTL 0 0 0 1 1	offset
6	5	5	16

Format:

BC0TL offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of coprocessor 0's condition line, as sampled during the previous instruction, is true, the target address is branched to with a delay of one instruction.

If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

Because the condition line is sampled during the previous instruction, there must be at least one instruction between this instruction and a coprocessor instruction that changes the condition line.

Operation:

```
T-1:  condition <- $SR_{18}$
T:    target <- $(offset_{15})^{46}$ || offset || $0^2$
T+1:  if condition then
          PC <- PC + target
      else
          NullifyCurrentInstruction
      endif
```

Exceptions:

Coprocessor unusable exception

BEQ Branch On Equal BEQ

31 26	25 21	20 16	15 0
BEQ 0 0 0 1 0 0	rs	rt	offset
6	5	5	16

Format:

BEQ rs, rt, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. The contents of general register *rs* and the contents of general register *rt* are compared. If the two registers are equal, then the program branches to the target address, with a delay of one instruction.

Operation:

```
T:     target <- $(\text{offset}_{15})^{46}$ || offset || $0^2$
       condition <- (GPR [rs] = GPR [rt])
T+1:   if condition then
           PC <- PC + target
       endif
```

Exceptions:

None

BEQL Branch On Equal Likely BEQL

31 26	25 21	20 16	15 0
BEQL 0 1 0 1 0 0	rs	rt	offset
6	5	5	16

Format:

BEQL rs, rt, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. The contents of general register *rs* and the contents of general register *rt* are compared. If the two registers are equal, the target address is branched to, with a delay of one instruction. If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

Operation:

```
T:     target <- $(\text{offset}_{15})^{46}$ || offset || $0^2$
       condition <- (GPR [rs] = GPR [rt])
T+1:   if condition then
           PC <- PC + target
       else
           NullifyCurrentInstruction
       endif
```

Exceptions:

None

BGEZ Branch On Greater Than Or Equal To Zero BGEZ

31 26	25 21	20 16	15 0
REGIMM 0 0 0 0 0 1	rs	BGEZ 0 0 0 0 1	offset
6	5	5	16

Format:

BGEZ rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of general register *rs* have the sign bit cleared, then the program branches to the target address, with a delay of one instruction.

Operation:

```
T:    target <- $(\text{offset}_{15})^{46}$ || offset || $0^2$
      condition <- (GPR [rs]$_{63}$ = 0)
T+1:  if condition then
          PC <- PC + target
      endif
```

Exceptions:

None

Note: The results of arithmetic operations are sign-extended from 32 to 64 bits and stored destination registers. Therefore, GPR [rs]$_{63}$ generally indicates the sign of data on 32-bit mode the same as GPR [rs]$_{31}$.

BGEZAL Branch On Greater Than Or Equal To Zero And Link BGEZAL

31 26	25 21	20 16	15 0
REGIMM 0 0 0 0 0 1	rs	BGEZAL 1 0 0 0 1	offset
6	5	5	16

Format:

BGEZAL rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset,* shifted left two bits and sign-extended. Unconditionally, the address of the instruction after the delay slot is placed in the link register, *r31*. If the contents of general register *rs* have the sign bit cleared, then the program branches to the target address, with a delay of one instruction.

General register *rs* may not be general register *31*, because such an instruction is not restartable. An attempt to execute this instruction is not trapped, however.

Operation:

```
T:    target <- $(\text{offset}_{15})^{46}$ || offset || $0^2$
      condition <- (GPR [rs]$_{63}$ = 0)
      GPR [31] <- PC + 8
T+1:  if condition then
          PC <- PC + target
      endif
```

Exceptions:

None

Note: The results of arithmetic operations are sign-extended from 32 to 64 bits and stored destination registers. Therefore, GPR [rs]$_{63}$ generally indicates the sign of data on 32-bit mode the same as GPR [rs]$_{31}$.

BGEZALL Branch On Greater Than Or Equal To Zero And Link Likely BGEZALL

31 26	25 21	20 16	15 0
REGIMM 0 0 0 0 0 1	rs	BGEZALL 1 0 0 1 1	offset
6	5	5	16

Format:

BGEZALL rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset,* shifted left two bits and sign-extended. Unconditionally, the address of the instruction after the delay slot is placed in the link register, *r31*. If the contents of general register *rs* have the sign bit cleared, then the program branches to the target address, with a delay of one instruction. General register *rs* may not be general register *31*, because such an instruction is not restartable. An attempt to execute this instruction is not trapped, however. If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

Operation:

```
T:     target <- $(\text{offset}_{15})^{46}$ || offset || $0^{2}$
       condition <- (GPR $[rs]_{63}$ = 0)
       GPR [31] <- PC + 8
T+1:   if condition then
           PC <- PC + target
       else
           NullifyCurrentInstruction
       endif
```

Exceptions:

None

Note: The results of arithmetic operations are sign-extended from 32 to 64 bits and stored destination registers. Therefore, GPR $[rs]_{63}$ generally indicates the sign of data on 32-bit mode the same as GPR $[rs]_{31}$.

BGEZL Branch On Greater Than Or Equal To Zero Likely BGEZL

31 26	25 21	20 16	15 0
REGIMM 0 0 0 0 0 1	rs	BGEZL 0 0 0 1 1	offset
6	5	5	16

Format:

BGEZL rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of general register *rs* have the sign bit cleared, then the program branches to the target address, with a delay of one instruction. If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

Operation:

```
T:    target <- $(\text{offset}_{15})^{46}$ || offset || $0^{2}$
      condition <- (GPR $[\text{rs}]_{63}$ = 0)
T+1:  if condition then
          PC <- PC + target
      else
          NullifyCurrentInstruction
      endif
```

Exceptions:

None

Note: The results of arithmetic operations are sign-extended from 32 to 64 bits and stored destination registers. Therefore, GPR $[\text{rs}]_{63}$ generally indicates the sign of data on 32-bit mode the same as GPR $[\text{rs}]_{31}$.

BGTZ Branch On Greater Than Zero BGTZ

31 26	25 21	20 16	15 0
BGTZ 0 0 0 1 1 1	rs	0 0 0 0 0 0	offset
6	5	5	16

Format:

BGTZ rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. The contents of general register *rs* are compared to zero. If the contents of general register *rs* have the sign bit cleared and are not equal to zero, then the program branches to the target address, with a delay of one instruction.

Operation:

```
T:    target <- $(\text{offset}_{15})^{46}$ || offset || $0^2$
      condition <- (GPR $[\text{rs}]_{63}$ = 0) and (GPR [rs] $\neq 0^{64}$)
T+1:  if condition then
          PC <- PC + target
      endif
```

Exceptions:

None

Note: The results of arithmetic operations are sign-extended from 32 to 64 bits and stored destination registers. Therefore, GPR $[\text{rs}]_{63}$ generally indicates the sign of data on 32-bit mode the same as GPR $[\text{rs}]_{31}$.

BGTZL Branch On Greater Than Zero Likely BGTZL

31 26	25 21	20 16	15 0
BGTZL 0 1 0 1 1 1	rs	0 0 0 0 0 0	offset
6	5	5	16

Format:

BGTZL rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. The contents of general register *rs* are compared to zero. If the contents of general register *rs* have the sign bit cleared and are not equal to zero, then the program branches to the target address, with a delay of one instruction. If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

Operation:

```
T:      target <- $(\text{offset}_{15})^{46}$ || offset || $0^2$
        condition <- (GPR $[rs]_{63}$ = 0) and (GPR [rs] $\neq 0^{64}$)
T+1:    if condition then
            PC <- PC + target
        else
            NullifyCurrentInstruction
        endif
```

Exceptions:

None

Note: The results of arithmetic operations are sign-extended from 32 to 64 bits and stored destination registers. Therefore, GPR [rs]$_{63}$ generally indicates the sign of data on 32-bit mode the same as GPR [rs]$_{31}$.

BLEZ Branch On Less Than Or Equal To Zero BLEZ

31 26	25 21	20 16	15 0
BLEZ 0 0 0 1 1 0	rs	0 0 0 0 0 0	offset
6	5	5	16

Format:

BLEZ rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. The contents of general register *rs* are compared to zero. If the contents of general register *rs* have the sign bit set, or are equal to zero, then the program branches to the target address, with a delay of one instruction.

Operation:

```
T:    target <- $(\text{offset}_{15})^{46}$ || offset || $0^2$
      condition <- (GPR $[\text{rs}]_{63}$ = 1) or (GPR [rs] = $0^{64}$)
T+1:  if condition then
          PC <- PC + target
      endif
```

Exceptions:

None

Note: The results of arithmetic operations are sign-extended from 32 to 64 bits and stored destination registers. Therefore, GPR $[\text{rs}]_{63}$ generally indicates the sign of data on 32-bit mode the same as GPR $[\text{rs}]_{31}$.

BLEZL Branch On Less Than Or Equal To Zero Likely BLEZL

31 26	25 21	20 16	15 0
BLEZL 0 1 0 1 1 0	rs	0 0 0 0 0 0	offset
6	5	5	16

Format:

BLEZL rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. The contents of general register *rs* is compared to zero. If the contents of general register *rs* have the sign bit set, or are equal to zero, then the program branches to the target address, with a delay of one instruction.

If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

Operation:

```
T:    target <- $(\text{offset}_{15})^{46}$ || offset || $0^2$
      condition <- (GPR $[rs]_{63}$ = 1) or (GPR [rs] = $0^{64}$)
T+1:  if condition then
          PC <- PC + target
      else
          NullifyCurrentInstruction
      endif
```

Exceptions:

None

Note: The results of arithmetic operations are sign-extended from 32 to 64 bits and stored destination registers. Therefore, GPR $[rs]_{63}$ generally indicates the sign of data on 32-bit mode the same as GPR $[rs]_{31}$.

BLTZ Branch On Less Than Zero BLTZ

31 26	25 21	20 16	15 0
REGIMM 0 0 0 0 0 1	rs	BLTZ 0 0 0 0 0	offset
6	5	5	16

Format:

BLTZ rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of general register *rs* have the sign bit set, then the program branches to the target address, with a delay of one instruction.

Operation:

```
T:    target <- $(\text{offset}_{15})^{46}$ || offset || $0^{2}$
      condition <- (GPR $[\text{rs}]_{63}$ = 1)
T+1:  if condition then
          PC <- PC + target
      endif
```

Exceptions:

None

Note: The results of arithmetic operations are sign-extended from 32 to 64 bits and stored destination registers. Therefore, GPR $[\text{rs}]_{63}$ generally indicates the sign of data on 32-bit mode the same as GPR $[\text{rs}]_{31}$.

BLTZAL Branch On Less Than Zero And Link BLTZAL

31 26	25 21	20 16	15 0
REGIMM 0 0 0 0 0 1	rs	BLTZAL 1 0 0 0 0	offset
6	5	5	16

Format:

BLTZAL rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset,* shifted left two bits and sign-extended. Unconditionally, the address of the instruction after the delay slot is placed in the link register, *r31*. If the contents of general register *rs* have the sign bit set, then the program branches to the target address, with a delay of one instruction.

General register *rs* may not be general register *31*, because such an instruction is not restartable. An attempt to execute this instruction with register *31* specified as *rs* is not trapped, however.

Operation:

```
T:    target <- $(\text{offset}_{15})^{46}$ || offset || $0^2$
      condition <- (GPR $[\text{rs}]_{63}$ = 1)
      GPR [31] <- PC + 8
T+1:  if condition then
          PC <- PC + target
      endif
```

Exceptions:

None

Note: The results of arithmetic operations are sign-extended from 32 to 64 bits and stored destination registers. Therefore, GPR $[\text{rs}]_{63}$ generally indicates the sign of data on 32-bit mode the same as GPR $[\text{rs}]_{31}$.

BLTZALL Branch On Less Than Zero And Link Likely BLTZALL

31 26	25 21	20 16	15 0
REGIMM 0 0 0 0 0 1	rs	BLTZALL 1 0 0 1 0	offset
6	5	5	16

Format:

BLTZALL rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset,* shifted left two bits and sign-extended. Unconditionally, the address of the instruction after the delay slot is placed in the link register, *r31*. If the contents of general register *rs* have the sign bit set, then the program branches to the target address, with a delay of one instruction.

General register *rs* may not be general register *31*, because such an instruction is not restartable. An attempt to execute this instruction with register *31* specified as *rs* is not trapped, however. If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

Operation:

```
T:    target <- $(\text{offset}_{15})^{46}$ || offset || $0^2$
      condition <- (GPR [rs]$_{63}$ = 1)
      GPR [31] <- PC + 8
T+1:  if condition then
          PC <- PC + target
      else
          NullifyCurrentInstruction
      endif
```

Exceptions:

None

Note: The results of arithmetic operations are sign-extended from 32 to 64 bits and stored destination registers. Therefore, GPR [rs]$_{63}$ generally indicates the sign of data on 32-bit mode the same as GPR [rs]$_{31}$.

BLTZL Branch On Less Than Zero Likely BLTZL

31 26	25 21	20 16	15 0
REGIMM 0 0 0 0 0 1	rs	BLTZL 0 0 0 1 0	offset
6	5	5	16

Format:

BLTZ rs, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of general register *rs* have the sign bit set, then the program branches to the target address, with a delay of one instruction. If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

Operation:

```
T:    target <- $(\text{offset}_{15})^{46}$ || offset || $0^2$
      condition <- (GPR $[rs]_{63}$ = 1)
T+1:  if condition then
          PC <- PC + target
      else
          NullifyCurrentInstruction
      endif
```

Exceptions:

None

Note: The results of arithmetic operations are sign-extended from 32 to 64 bits and stored destination registers. Therefore, GPR $[rs]_{63}$ generally indicates the sign of data on 32-bit mode the same as GPR $[rs]_{31}$.

BNE Branch On Not Equal BNE

31 26	25 21	20 16	15 0
BNE 0 0 0 1 0 1	rs	rt	offset
6	5	5	16

Format:

BNE rs, rt, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset,* shifted left two bits and sign-extended. The contents of general register *rs* and the contents of general register *rt* are compared. If the two registers are not equal, then the program branches to the target address, with a delay of one instruction.

Operation:

```
T:    target <- $(\text{offset}_{15})^{46}$ || offset || $0^2$
      condition <- (GPR [rs] $\neq$ GPR [rt])
T+1:  if condition then
          PC <- PC + target
      endif
```

Exceptions:

None

BNEL Branch On Not Equal Likely BNEL

31 26	25 21	20 16	15 0
BNEL 0 1 0 1 0 1	rs	rt	offset
6	5	5	16

Format:

BNEL rs, rt, offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset,* shifted left two bits and sign-extended. The contents of general register *rs* and the contents of general register *rt* are compared. If the two registers are not equal, then the program branches to the target address, with a delay of one instruction.

If the conditional branch is not taken, the instruction in the branch delay slot is nullified.

Operation:

```
T:    target <- $(\text{offset}_{15})^{46}$ || offset || $0^{2}$
      condition <- (GPR [rs] $\neq$ GPR [rt])
T+1:  if condition then
          PC <- PC + target
      else
          NullifyCurrentInstruction
      endif
```

Exceptions:

None

BREAK Breakpoint BREAK

31 26	25 6	5 0
SPECIAL 0 0 0 0 0 0	code	BREAK 0 0 1 1 0 1
6	20	6

Format:

BREAK

Description:

A breakpoint trap occurs, immediately and unconditionally transferring control to the exception handler.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
T:     BreakpointException
```

Exceptions:

Breakpoint exception

CACHE Cache CACHE

31 … 26	25 … 21	20 … 16	15 … 0
CACHE 1 0 1 1 1 1	base	op	offset
6	5	5	16

Format:

CACHE op, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The virtual address is translated to a physical address using the TLB, and the 5-bit sub-opcode specifies a cache operation for that address.

If CP0 is not usable (User or Supervisor mode) and the CP0 enable bit in the *Status* register is clear, a coprocessor unusable exception is taken. The operation of this instruction on any operation/cache combination not listed below, or on a secondary cache, is undefined. The operation of this instruction on uncached addresses is also undefined.

The Index operation uses part of the virtual address to specify a cache block.

For a primary cache of $2^{\text{CACHEBITS}}$ bytes with 2^{LINEBITS} bytes per tag, $\text{vAddr}_{\text{CACHEBITS...LINEBITS}}$ specifies the block.

Index Load Tag also uses $\text{vAddr}_{\text{LINEBITS...3}}$ to select the doubleword for reading parity. When the *CE* bit of the *Status* register is set, Fill Cache op uses the PErr register to store parity values into the cache.

The Hit operation accesses the specified cache as normal data references, and performs the specified operation if the cache block contains valid data with the specified physical address (a hit). If the cache block is invalid or contains a different address (a miss), no operation is performed.

CACHE Cache (Continued) CACHE

Write back from a primary cache goes to memory. The address to be written is specified by the cache tag and not the translated physical address.

TLB Refill and TLB Invalid exceptions can occur on any operation. For Index operations (where the physical address is used to index the cache but need not match the cache tag) unmapped addresses may be used to avoid TLB exceptions. This operation never causes a TLB Modified exception.

Bits 17...16 of the instruction specify the cache as follows:

Code	Name	Cache
0	I	Primary instruction
1	D	Primary data
2 - 3	NA	Undefined

CACHE Cache (Continued) CACHE

Bits 20...18 (this value is listed under the **Code** column) of the instruction specify the operation as follows:

Code	Cache	Name	Operation
0	I	Index Invalidate	Set the cache state of the cache block to Invalid.
0	D	Index Write-Back Invalidate	Examine the cache state and W bit of the primary data cache block at the index specified by the virtual address. If the state is not Invalid and the W bit is set, then write back the block to memory. The address to write is taken from the primary cache tag. Set cache state of primary cache block to Invalid.
1	I, D	Index Load Tag	Read the tag for the cache block at the specified index and place it into the TagLo CP0 registers, ignoring parity errors. Also load the data parity bits into the ECC register.
2	I, D	Index Store Tag	Write the tag for the cache block at the specified index from the TagLo and TagHi CP0 registers.
3	D	Create Dirty Exclusive	This operation is used to avoid loading data needlessly from memory when writing new contents into an entire cache block. If the cache block does not contain the specified address, and the block is dirty, write it back to the memory. In all cases, set the cache state to Dirty.
4	I, D	Hit Invalidate	If the cache block contains the specified address, mark the cache block invalid.
5	D	Hit WriteBack Invalidate	If the cache block contains the specified address, write back the data if it is dirty, and mark the cache block invalid.
5	I	Fill	Fill the primary instruction cache block from memory. If the CE bit of the Status register is set, the contents of the ECC register is used instead of the computed parity bits for addressed doubleword when written to the instruction cache.
6	D	Hit WriteBack	If the cache block contains the specified address, and the W bit is set, write back the data to memory and clear the W bit.
6	I	Hit WriteBack	If the cache block contains the specified address, write back the data unconditionally.

CACHE Cache (Continued) CACHE

Operation:

```
T:    vAddr <- $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15...0})$ + GPR [base]
      (pAddr, uncached) <- AddressTranslation (vAddr, DATA)
      CacheOp (op, vAddr, pAddr)
```

Exceptions:

Coprocessor unusable exception

TLB Refill exception

TLB Invalid exception

Bus Error exception

Address Error exception

Cache Error exception

DADD Doubleword Add DADD

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	DADD 1 0 1 1 0 0
6	5	5	5	5	6

Format:

DADD rd, rs, rt

Description:

The contents of general register *rs* and the contents of general register *rt* are added to form the result. The result is placed into general register *rd*.

An overflow exception occurs if the carries out of bits 62 and 63 differ (2's complement overflow). The destination register *rd* is not modified when an integer overflow exception occurs.

This operation is only defined for the VR4100 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
    T:     GPR [rd] <- GPR [rs] + GPR [rt]
```

Exceptions:

Integer overflow exception

Reserved instruction exception (VR4100 in 32-bit mode)

DADDI — Doubleword Add Immediate — DADDI

31 … 26	25 … 21	20 … 16	15 … 0
DADDI 0 1 1 0 0 0	rs	rt	immediate
6	5	5	16

Format:

DADDI rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and added to the contents of general register *rs* to form the result. The result is placed into general register *rt.*

An overflow exception occurs if carries out of bits 62 and 63 differ (2's complement overflow). The destination register *rt* is not modified when an integer overflow exception occurs.

This operation is only defined for the VR4100 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
T:    GPR [rt] <- GPR [rs] + $(\text{immediate}_{15})^{48}$ || $\text{immediate}_{15...0}$
```

Exceptions:

Integer overflow exception

Reserved instruction exception (VR4100 in 32-bit mode)

DADDIU Doubleword Add Immediate Unsigned DADDIU

31 26	25 21	20 16	15 0
DADDIU 0 1 1 0 0 1	rs	rt	immediate
6	5	5	16

Format:

DADDIU rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and added to the contents of general register *rs* to form the result. The result is placed into general register *rt.* No integer overflow exception occurs under any circumstances.

The only difference between this instruction and the DADDI instruction is that DADDIU never causes an overflow exception.

This operation is only defined for the VR4100 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
T:    GPR [rt] <- GPR [rs] + $(\text{immediate}_{15})^{48}$ || $\text{immediate}_{15...0}$
```

Exceptions:

Reserved instruction exception (VR4100 in 32-bit mode)

DADDU Doubleword Add Unsigned DADDU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	DADDU 1 0 1 1 0 1
6	5	5	5	5	6

Format:

DADDU rd, rs, rt

Description:

The contents of general register *rs* and the contents of general register *rt* are added to form the result. The result is placed into general register *rd*.

No overflow exception occurs under any circumstances.

The only difference between this instruction and the DADD instruction is that DADDU never causes an overflow exception.

This operation is only defined for the VR4100 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
T:    GPR [rd] <- GPR [rs] + GPR [rt]
```

Exceptions:

Reserved instruction exception (VR4100 in 32-bit mode)

DDIV   Doubleword Divide   DDIV

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	0 0 0 0 0 0 0 0 0 0 0	DDIV 0 1 1 1 1 0
6	5	5	10	6

Format:

DDIV rs, rt

Description:

The contents of general register *rs* are divided by the contents of general register *rt,* treating both operands as 2's complement values. No overflow exception occurs under any circumstances, and the result of this operation is undefined when the divisor is zero.

This instruction is typically followed by additional instructions to check for a zero divisor and for overflow.

When the operation completes, the quotient word of the double result is loaded into special register *LO*, and the remainder word of the double result is loaded into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. Correct operation requires separating reads of *HI* or *LO* from writes by two or more instructions. This is defined in this manner to take account of the R4000 hazards (for code compatibility) as well as the VR4100● s own hazards.

This operation is only defined for the VR4100 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
T-2:  LO  <- undefined
      HI  <- undefined
T-1:  LO  <- undefined
      HI  <- undefined
T:    LO  <- GPR [rs] div GPR [rt]
      HI  <- GPR [rs] mod GPR [rt]
```

Exceptions:

Reserved instruction exception

DDIVU Doubleword Divide Unsigned DDIVU

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	0 0 0 0 0 0 0 0 0 0 0	DDIVU 0 1 1 1 1 1
6	5	5	10	6

Format:

DDIVU rs, rt

Description:

The contents of general register *rs* are divided by the contents of general register *rt,* treating both operands as unsigned values. No integer overflow exception occurs under any circumstances, and the result of this operation is undefined when the divisor is zero.

This instruction may be followed by additional instructions to check for a zero divisor, inserted by the programmer.

When the operation completes, the quotient word of the double result is loaded into special register *LO*, and the remainder word of the double result is loaded into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. Correct operation requires separating reads of *HI* or *LO* from writes by two or more instructions.

This operation is only defined for the VR4100 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
T-2:  LO  <- undefined
      HI  <- undefined
T-1:  LO  <- undefined
      HI  <- undefined
T:    LO  <- (0 || GPR [rs]) div (0 || GPR [rt])
      HI  <- (0 || GPR [rs]) mod (0 || GPR [rt])
```

Exceptions:

Reserved instruction exception (VR4100 in 32-bit mode)

DIV Divide DIV

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	0 0 0 0 0 0 0 0 0 0 0	DIV 0 1 1 0 1 0
6	5	5	10	6

Format:

DIV rs, rt

Description:

The contents of general register *rs* are divided by the contents of general register *rt,* treating both operands as 2's complement values. No overflow exception occurs under any circumstances, and the result of this operation is undefined when the divisor is zero.

In 64-bit mode, the operands must be valid sign-extended, 32-bit values.

This instruction is typically followed by additional instructions to check for a zero divisor and for overflow.

When the operation completes, the quotient word of the double result is loaded into special register *LO*, and the remainder word of the double result is loaded into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. Correct operation requires separating reads of *HI* or *LO* from writes by two or more instructions.

DIV Divide (Continued) DIV

Operation:

```
T-2:  LO  <- undefined
      HI  <- undefined
T-1:  LO  <- undefined
      HI  <- undefined
T:    q   <- GPR [rs]$_{31...0}$ div GPR [rt]$_{31...0}$
      r   <- GPR [rs]$_{31...0}$ mod GPR [rt]$_{31...0}$
      LO  <- $(q_{31})^{32}$ || $q_{31...0}$
      HI  <- $(r_{31})^{32}$ || $r_{31...0}$
```

Exceptions:

None

DIVU Divide Unsigned DIVU

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	0 0 0 0 0 0 0 0 0 0 0	DIVU 0 1 1 0 1 1
6	5	5	10	6

Format:

DIVU rs, rt

Description:

The contents of general register *rs* are divided by the contents of general register *rt,* treating both operands as unsigned values. No integer overflow exception occurs under any circumstances, and the result of this operation is undefined when the divisor is zero.

In 64-bit mode, the operands must be valid sign-extended, 32-bit values.

This instruction is typically followed by additional instructions to check for a zero divisor.

When the operation completes, the quotient word of the double result is loaded into special register *LO*, and the remainder word of the double result is loaded into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. Correct operation requires separating reads of *HI* or *LO* from writes by two or more instructions.

DIVU Divide Unsigned (Continued) DIVU

Operation:

```
T-2:  LO  <- undefined
      HI  <- undefined
T-1:  LO  <- undefined
      HI  <- undefined
T:    q   <- $(0 \| GPR [rs]_{31...0})$ div $(0 \| GPR [rt]_{31...0})$
      r   <- $(0 \| GPR [rs]_{31...0})$ mod $(0 \| GPR [rt]_{31...0})$
      LO  <- $(q_{31})^{32} \| q_{31...0}$
      HI  <- $(r_{31})^{32} \| r_{31...0}$
```

Exceptions:

None

DMADD16 Doubleword Multiply and Add 16-bit integer DMADD16

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	0 0 0 0 0 0 0 0 0 0 0	DMADD16 1 0 1 0 0 1
6	5	5	10	6

Format:

DMADD16 rs, rt

Description:

The contents of general registers *rs* and *rt* are multiplied, treating both operands as 16-bit 2's complement values. The operand[62:15] must be valid 15-bit, sign-extended values. If not, the result is unpredictable.

This multiplied result and the 64-bit data joined of special register *LO* is added to form the result as a signed integer.

No integer overflow exception occurs under any circumstances.

When the operation completes, the double result is loaded into special register *LO*.

The following Table are hazard cycles between DMADD16 and other instructions.

MULT/MULTU -> DMADD16	1 Cycle
DMULT/DMULTU -> DMADD16	4 Cycles
DIV/DIVU -> DMADD16	36 Cycles
DDIV/DDIVU -> DMADD16	68 Cycles
MFHI/MFLO -> DMADD16	2 Cycles
MADD16 -> DMADD16	0 Cycle
DMADD16 -> DMADD16	0 Cycle

DMADD16 Doubleword Multiply and Add 16-bit integer DMADD16 (Continued)

DMADD16 -> MULT/MULTU	0 Cycle
DMADD16 -> DMULT/DMULTU	0 Cycle
DMADD16 -> DIV/DIVU	1 Cycle
DMADD16 -> DDIV/DDIVU	1 Cycle
DMADD16 -> MFHI/MFLO	0 Cycle
DMADD16 -> DMADD16	0 Cycle
DMADD16 -> MADD16	0 Cycle

Operation:

```
T-2:  LO   <- undefined
      HI   <- undefined
T-1:  LO   <- undefined
      HI   <- undefined
T:    temp <- GPR [rs] * GPR [rt]
      LO   <- temp + LO
      HI   <- undefined
```

Exceptions:

Reserved Instruction Exception (VR4100 in 32-bit user mode
VR4100 in 32-bit supervisor mode)

DMFC0 Doubleword Move From System Control Coprocessor DMFC0

31 26	25 21	20 16	15 11	10 0
COP0 0 1 0 0 0 0	DMF 0 0 0 0 1	rt	rd	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format:

DMFC0 rt, rd

Description:

The contents of coprocessor register *rd* of the CP0 are loaded into general register *rt.*

This operation is defined for the VR4100 operating in 64-bit mode and in 32-bit kernel mode. Execution of this instruction in 32-bit user or supervisor mode causes a reserved instruction exception. All 64-bits of the general register destination are written from the coprocessor register source. The operation of DMFC0 on a 32-bit coprocessor 0 register is undefined.

Operation:

```
T:     data <- CPR [0, rd]
T+1:   GPR [rt] <- data
```

Exceptions:

Coprocessor unusable exception	(user mode and supervisor mode if CP0 not enabled)
Reserved instruction exception	(VR4100 in 32-bit user mode VR4100 in 32-bit supervisor mode)

DMTC0 Doubleword Move To System Control Coprocessor DMTC0

31 26	25 21	20 16	15 11	10 0
COP0 0 1 0 0 0 0	DMT 0 0 1 0 1	rt	rd	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format:

DMTC0 rt, rd

Description:

The contents of general register *rt* are loaded into coprocessor register *rd* of the CP0.

This operation is defined for the VR4100 operating in 64-bit mode or in 32-bit kernel mode. Execution of this instruction in 32-bit user or supervisor mode causes a reserved instruction exception.

All 64-bits of the coprocessor 0 register are written from the general register source. The operation of DMTC0 on a 32-bit coprocessor 0 register is undefined.

Because the state of the virtual address translation system may be altered by this instruction, the operation of load instructions, store instructions, and TLB operations immediately prior to and after this instruction are undefined.

Operation:

```
T:    data <- GPR [rt]
T+1:  CPR [0, rd] <- data
```

Exceptions:

Coprocessor unusable exception: (In user and supervisor mode if CP0 not enabled)

Reserved exception: (VR4100 in 32-bit user mode
VR4100 in 32-bit supervisor mode)

DMULT Doubleword Multiply DMULT

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	0 0 0 0 0 0 0 0 0 0 0	DMULT 0 1 1 1 0 0
6	5	5	10	6

Format:

DMULT rs, rt

Description:

The contents of general registers *rs* and *rt* are multiplied, treating both operands as 2● s complement values. No integer overflow exception occurs under any circumstances.

When the operation completes, the low-order word of the double result is loaded into special register *LO*, and the high-order word of the double result is loaded into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results of these instructions are undefined. Correct operation requires separating reads of *HI* or *LO* from writes by a minimum of two other instructions.

This operation is only defined for the VR4100 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
T-2:  LO   <- undefined
      HI   <- undefined
T-1:  LO   <- undefined
      HI   <- undefined
T:    t    <- GPR [rs] * GPR [rt]
      LO   <- $t_{63...0}$
      HI   <- $t_{127...64}$
```

Exceptions:

Reserved instruction exception (VR4100 in 32-bit mode)

DMULTU Doubleword Multiply Unsigned DMULTU

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	0 0 0 0 0 0 0 0 0 0 0	DMULTU 0 1 1 1 0 1
6	5	5	10	6

Format:

DMULTU rs, rt

Description:

The contents of general register *rs* and the contents of general register *rt* are multiplied, treating both operands as unsigned values. No overflow exception occurs under any circumstances.

When the operation completes, the low-order word of the double result is loaded into special register *LO*, and the high-order word of the double result is loaded into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results of these instructions are undefined. Correct operation requires separating reads of *HI* or *LO* from writes by a minimum of two instructions.

This operation is only defined for the VR4100 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
T-2:  LO   <- undefined
      HI   <- undefined
T-1:  LO   <- undefined
      HI   <- undefined
T:    t    <- (0 || GPR [rs]) * (0 || GPR [rt])
      LO   <- $t_{63...0}$
      HI   <- $t_{127...64}$
```

Exceptions:

Reserved instruction exception (VR4100 in 32-bit mode)

DSLL Doubleword Shift Left Logical DSLL

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0	rt	rd	sa	DSLL 1 1 1 0 0 0
6	5	5	5	5	6

Format:

DSLL rd, rt, sa

Description:

The contents of general register *rt* are shifted left by *sa* bits, inserting zeros into the low-order bits. The result is placed in register *rd.*

This operation is only defined for the VR4100 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
T:     s <- 0 || sa
       GPR [rd] <- GPR [rt]$_{(63 - s)...0}$ || $0^{s}$
```

Exceptions:

Reserved instruction exception (VR4100 in 32-bit mode)

DSLLV Doubleword Shift Left Logical Variable DSLLV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	DSLLV 0 1 0 1 0 0
6	5	5	5	5	6

Format:

DSLLV rd, rt, rs

Description:

The contents of general register *rt* are shifted left by the number of bits specified by the low-order six bits contained in general register *rs*, inserting zeros into the low-order bits. The result is placed in register *rd*.

This operation is only defined for the VR4100 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
T:    s <- GPR [rs]$_{5...0}$
      GPR [rd] <- GPR [rt]$_{(63 - s)...0}$ || $0^{s}$
```

Exceptions:

Reserved instruction exception (VR4100 in 32-bit mode)

DSLL32 Doubleword Shift Left Logical + 32 DSLL32

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0	rt	rd	sa	DSLL32 1 1 1 1 0 0
6	5	5	5	5	6

Format:

DSLL32 rd, rt, sa

Description:

The contents of general register *rt* are shifted left by *32* + *sa* bits, inserting zeros into the low-order bits. The result is placed in register *rd*.

This operation is only defined for the VR4100 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
T:    s <- 1 || sa
      GPR [rd] <- GPR [rt]$_{(63 - s)...0}$ || $0^s$
```

Exceptions:

Reserved instruction exception (VR4100 in 32-bit mode)

DSRA Doubleword Shift Right Arithmetic DSRA

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0	rt	rd	sa	DSRA 1 1 1 0 1 1
6	5	5	5	5	6

Format:

DSRA rd, rt, sa

Description:

The contents of general register *rt* are shifted right by *sa* bits, sign-extending the high-order bits. The result is placed in register *rd*.

This operation is only defined for the VR4100 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
T:   s <- 0 || sa
     GPR [rd] <- $(\mathrm{GPR}\ [rt]_{63})^{s}$ || GPR [rt] $_{63...s}$
```

Exceptions:

Reserved instruction exception (VR4100 in 32-bit mode)

DSRAV Doubleword Shift Right Arithmetic Variable DSRAV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	DSRAV 0 1 0 1 1 1
6	5	5	5	5	6

Format:

DSRAV rd, rt, rs

Description:

The contents of general register *rt* are shifted right by the number of bits specified by the low-order six bits of general register *rs*, sign-extending the high-order bits. The result is placed in register *rd*.

This operation is only defined for the VR4100 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
T:    s <- GPR [rs]$_{5...0}$
      GPR [rd] <- (GPR [rt]$_{63}$)$^{s}$ || GPR [rt] $_{63...s}$
```

Exceptions:

Reserved instruction exception (VR4100 in 32-bit mode)

DSRA32 Doubleword Shift Right Arithmetic + 32 DSRA32

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0	rt	rd	sa	DSRA32 1 1 1 1 1 1
6	5	5	5	5	6

Format:

DSRA32 rd, rt, sa

Description:

The contents of general register *rt* are shifted right by *32* + *sa* bits, sign-extending the high-order bits. The result is placed in register *rd*.

This operation is only defined for the VR4100 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
T:   s <- 1 || sa
     GPR [rd] <- $(\text{GPR [rt]}_{63})^{s}$ || $\text{GPR [rt]}_{63...s}$
```

Exceptions:

Reserved instruction exception (VR4100 in 32-bit mode)

DSRL Doubleword Shift Right Logical DSRL

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0	rt	rd	sa	DSRL 1 1 1 0 1 0
6	5	5	5	5	6

Format:

DSRL rd, rt, sa

Description:

The contents of general register *rt* are shifted right by *sa* bits, inserting zeros into the high-order bits. The result is placed in register *rd*.

This operation is only defined for the VR4100 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
T:    s <- 0 || sa
      GPR [rd] <- $0^s$ || GPR [rt]$_{63...s}$
```

Exceptions:

Reserved instruction exception (VR4100 in 32-bit mode)

DSRLV Doubleword Shift Right Logical Variable DSRLV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	DSRLV 0 1 0 1 1 0
6	5	5	5	5	6

Format:

DSRLV rd, rt, rs

Description:

The contents of general register *rt* are shifted right by the number of bits specified by the low-order six bits of general register *rs,* inserting zeros into the high-order bits. The result is placed in register *rd*. This operation is only defined for the VR4100 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
T:    s <- GPR [rs]$_{5...0}$
      GPR [rd] <- $0^{s}$ || GPR [rt]$_{63...s}$
```

Exceptions:

Reserved instruction exception (VR4100 in 32-bit mode)

DSRL32 Doubleword Shift Right Logical + 32 DSRL32

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0	rt	rd	sa	DSRL32 1 1 1 1 1 0
6	5	5	5	5	6

Format:

DSRL32 rd, rt, sa

Description:

The contents of general register *rt* are shifted right by *32* + *sa* bits, inserting zeros into the high-order bits. The result is placed in register *rd*.

This operation is only defined for the VR4100 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
T:   s <- 1 || sa
     GPR [rd] <- $0^{s}$ || GPR [rt]$_{63...s}$
```

Exceptions:

Reserved instruction exception (VR4100 in 32-bit mode)

DSUB Doubleword Subtract DSUB

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	DSUB 1 0 1 1 1 0
6	5	5	5	5	6

Format:

DSUB rd, rs, rt

Description:

The contents of general register *rt* are subtracted from the contents of general register *rs* to form a result. The result is placed into general register *rd.*

The only difference between this instruction and the DSUBU instruction is that DSUBU never traps on overflow.

An integer overflow exception takes place if the carries out of bits 62 and 63 differ 2's complement overflow). The destination register *rd* is not modified when an integer overflow exception occurs.

This operation is only defined for the VR4100 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
T:    GPR [rd] <- GPR [rs] - GPR [rt]
```

Exceptions:

Integer overflow exception

Reserved instruction exception (VR4100 in 32-bit mode)

DSUBU Doubleword Subtract Unsigned DSUBU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	DSUBU 1 0 1 1 1 1
6	5	5	5	5	6

Format:

DSUBU rd, rs, rt

Description:

The contents of general register *rt* are subtracted from the contents of general register *rs* to form a result. The result is placed into general register *rd*.

The only difference between this instruction and the DSUB instruction is that DSUBU never traps on overflow. No integer overflow exception occurs under any circumstances.

This operation is only defined for the VR4100 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
T:    GPR [rd] <- GPR [rs] - GPR [rt]
```

Exceptions:

Reserved instruction exception (VR4100 in 32-bit mode)

ERET   Exception Return   ERET

31 26	25	24 6	5 0
COP0 0 1 0 0 0 0	CO 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	ERET 0 1 1 0 0 0
6	1	19	6

Format:

ERET

Description:

ERET is the VR4100 instruction for returning from an interrupt, exception, or error trap. Unlike a branch or jump instruction, ERET does not execute the next instruction.

ERET must not itself be placed in a branch delay slot.

If the processor is servicing an error trap (SR_2 = 1), then load the PC from the *ErrorEPC* and clear the *ERL* bit of the *Status* register (SR_2). Otherwise (SR_2 = 0), load the PC from the *EPC*, and clear the *EXL* bit of the *Status* register (SR_1).

Operation:

```
T:    if $SR_2$ = 1 then
          PC <- ErrorEPC
          SR <- $SR_{31...3}$ || 0 || $SR_{1...0}$
      else
          PC <-EPC
          SR <- $SR_{31...2}$ || 0 || $SR_0$
      endif
```

Exceptions:

Coprocessor unusable exception

HIBERNATE Hibernate HIBERNATE

31 26	25	24 6	5 0
COP0 0 1 0 0 0 0	CO 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	HIBERNATE 1 0 0 0 1 1
6	1	19	6

Format:

HIBERNATE

Description:

In Hibernate mode, all internal clock, include Timer/Interrupt unit, and all system interface clocks are frozen at hi level.

To enter Hibernate mode from Fullspeed mode, first execute the HIBERNATE instruction. When the HIBERNATE instruction finishes the WB stage, the VR4100 wait by the SysAD bus is idle state, after then the internal clocks and the system interface clocks will shut down, thus freezing the pipeline.

Once the VR4100 is in Hibernate mode, the ColdRest sequence will cause the VR4100 to exit Hibernate mode and to enter Fullspeed mode.

Operation:

```
T:
T+1: hibernate operation ()
```

Exceptions:

Coprocessor unusable exception

J Jump J

31 26	25 0
J 0 0 0 0 1 0	target
6	26

Format:

J target

Description:

The 26-bit target address is shifted left two bits and combined with the high-order four bits of the address of the delay slot. The program unconditionally jumps to this calculated address with a delay of one instruction.

Operation:

```
T:    temp <- target
T+1:  PC <- $PC_{63...28}$ || temp || $0^2$
```

Exceptions:

None

JAL Jump And Link JAL

31 26	25 0
JAL 0 0 0 0 1 1	target
6	26

Format:

JAL target

Description:

The 26-bit target address is shifted left two bits and combined with the high-order four bits of the address of the delay slot. The program unconditionally jumps to this calculated address with a delay of one instruction. The address of the instruction after the delay slot is placed in the link register, *r31.*

Operation:

```
T:    temp <- target
      GPR [31] <- PC + 8
T+1:  PC <- $PC_{63...28}$ || temp || $0^2$
```

Exceptions:

None

JALR Jump And Link Register JALR

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	0 0 0 0 0 0	rd	0 0 0 0 0 0	JALR 0 0 1 0 0 1
6	5	5	5	5	6

Format:

JALR rs

JALR rd, rs

Description:

The program unconditionally jumps to the address contained in general register *rs*, with a delay of one instruction. The address of the instruction after the delay slot is placed in general register *rd*. The default value of *rd*, if omitted in the assembly language instruction, is 31.

Register specifiers *rs* and *rd* may not be equal, because such an instruction does not have the same effect when re-executed. However, an attempt to execute this instruction is *not* trapped, and the result of executing such an instruction is undefined.

Since instructions must be word-aligned, a **Jump and Link Register** instruction must specify a target register (*rs*) which contains an address whose two low-order bits are zero. If these low-order bits are not zero, an address error exception will occur when the jump target instruction is subsequently fetched.

Operation:

```
T:    temp <- GPR [rs]
      GPR [rd] <- PC + 8
T+1:  PC <- temp
```

Exceptions:

None

JR Jump Register JR

31 26	25 21	20 6	5 0
SPECIAL 0 0 0 0 0 0	rs	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	JR 0 0 1 0 0 0
6	5	15	6

Format:

JR rs

Description:

The program unconditionally jumps to the address contained in general register *rs*, with a delay of one instruction.

Since instructions must be word-aligned, a **Jump Register** instruction must specify a target register (*rs*) which contains an address whose two low-order bits are zero. If these low-order bits are not zero, an address error exception will occur when the jump target instruction is subsequently fetched.

Operation:

```
T:     temp <- GPR [rs]
T+1: PC <- temp
```

Exceptions:

None

LB Load Byte LB

31 — 26	25 — 21	20 — 16	15 — 0
LB 1 0 0 0 0 0	base	rt	offset
6	5	5	16

Format:

LB rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the byte at the memory location specified by the effective address are sign-extended and loaded into general register *rt*.

Operation:

```
T:    vAddr <- $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15...0})$ + GPR [base]
      (pAddr, uncached) <- AddressTranslation (vAddr, DATA)
      pAddr <- $\text{pAddr}_{\text{PSIZE - 1...3}} \,||\, (\text{pAddr}_{2...0}$ xor $\text{ReverseEndian}^{3})$
      mem <- LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
      byte <- $\text{vAddr}_{2...0}$ xor $\text{BigEndianCPU}^{3}$
      GPR [rt] <- $(\text{mem}_{7 + 8* \text{byte}})^{56} \,||\, \text{mem}_{7 + 8* \text{byte}...8* \text{byte}}$
```

Exceptions:

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

LBU Load Byte Unsigned LBU

31 26	25 21	20 16	15 0
LBU 1 0 0 1 0 0	base	rt	offset
6	5	5	16

Format:

LBU rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the byte at the memory location specified by the effective address are zero-extended and loaded into general register *rt*.

Operation:

```
T:     vAddr <- ((offset$_{15}$)$^{48}$ || offset$_{15...0}$) + GPR [base]
       (pAddr, uncached) <- AddressTranslation (vAddr, DATA)
       pAddr <- pAddr$_{PSIZE - 1...3}$ || (pAddr$_{2...0}$ xor ReverseEndian$^{3}$)
       mem <- LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
       byte <- vAddr$_{2...0}$ xor BigEndianCPU$^{3}$
       GPR [rt] <- 0$^{56}$ || mem$_{7 + 8* byte...8* byte}$
```

Exceptions:

TLB refill exception	Bus error exception
TLB invalid exception	Address error exception

LD Load Doubleword LD

31 … 26	25 … 21	20 … 16	15 … 0
LD 1 1 0 1 1 1	base	rt	offset
6	5	5	16

Format:

LD rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the 64-bit doubleword at the memory location specified by the effective address are loaded into general register *rt*.

If any of the three least-significant bits of the effective address are non-zero, an address error exception occurs.

This operation is only defined for the VR4100 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
T:   vAddr <- $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15...0})$ + GPR [base]
     (pAddr, uncached) <- AddressTranslation (vAddr, DATA)
     data <- LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
     GPR [rt] <- data
```

Exceptions:

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

Reserved instruction exception   (VR4100 in 32-bit user mode
VR4100 in 32-bit supervisor mode)

LDL   Load Doubleword Left   LDL

31 ... 26	25 ... 21	20 ... 16	15 ... 0
LDL 0 1 1 0 1 0	base	rt	offset
6	5	5	16

Format:

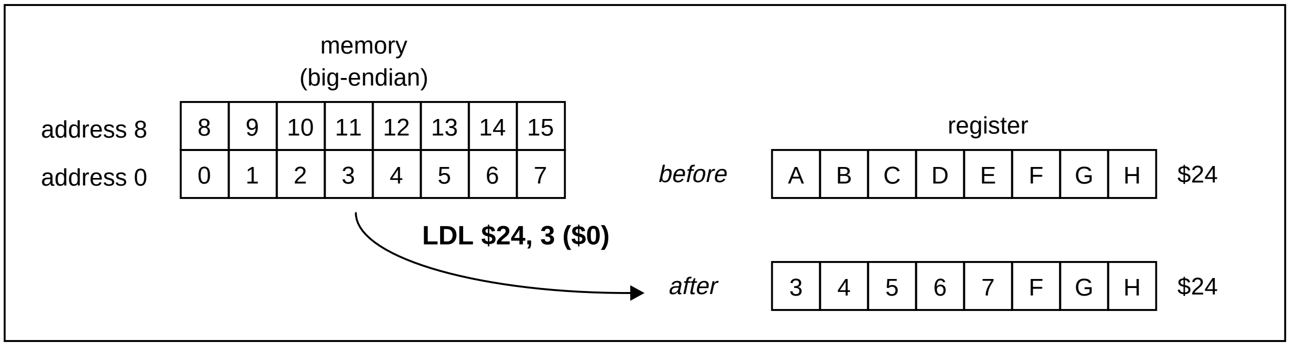
LDL rt, offset (base)

Description:

This instruction can be used in combination with the LDR instruction to load a register with eight consecutive bytes from memory, when the bytes cross a doubleword boundary. LDL loads the left portion of the register with the appropriate part of the high-order doubleword; LDR loads the right portion of the register with the appropriate part of the low-order doubleword.

The LDL instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which can specify an arbitrary byte. It reads bytes only from the doubleword in memory which contains the specified starting byte. From one to eight bytes will be loaded, depending on the starting byte specified.

Conceptually, it starts at the specified byte in memory and loads that byte into the high-order (left-most) byte of the register; then it loads bytes from memory into the register until it reaches the low-order byte of the doubleword in memory. The least-significant (right-most) byte(s) of the register will not be changed.



LDL Load Doubleword Left (Continued) LDL

The contents of general register *rt* are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register *rt* and a following LDL (or LDR) instruction which also specifies register *rt*.

No address error exceptions due to alignment are possible.

This operation is only defined for the VR4100 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
T:    vAddr <- $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15...0}) + \text{GPR [base]}$
      (pAddr, uncached) <- AddressTranslation (vAddr, DATA)
      pAddr <- $\text{pAddr}_{\text{PSIZE - 1...3}} \,||\, (\text{pAddr}_{2...0} \text{ xor ReverseEndian}^{3})$
      if BigEndianMem = 0 then
            pAddr <- $\text{pAddr}_{\text{PSIZE - 1...3}} \,||\, 0^{3}$
      endif
      byte <- $\text{vAddr}_{2...0} \text{ xor BigEndianCPU}^{3}$
      mem <- LoadMemory (uncached, byte, pAddr, vAddr, DATA)
      GPR [rt] <- $\text{mem}_{7 + 8^{*} \text{byte}...0} \,||\, \text{GPR [rt]}_{55 - 8^{*} \text{byte}...0}$
```

LDL Load Doubleword Left (Continued) LDL

Given a doubleword in a register and a doubleword in memory, the operation of LDL is as follows:

LDL

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

vAddr$_{2..0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	destination	type	offset		destination	type	offset	
			LEM	BEM			LEM	BEM
0	P B C D E F G H	0	0	7	I J K L M N O P	7	0	0
1	O P C D E F G H	1	0	6	J K L M N O P H	6	0	1
2	N O P D E F G H	2	0	5	K L M N O P G H	5	0	2
3	M N O P E F G H	3	0	4	L M N O P F G H	4	0	3
4	L M N O P F G H	4	0	3	M N O P E F G H	3	0	4
5	K L M N O P G H	5	0	2	N O P D E F G H	2	0	5
6	J K L M N O P H	6	0	1	O P C D E F G H	1	0	6
7	I J K L M N O P	7	0	0	P B C D E F G H	0	0	7

LEM Little-endian memory (BigEndianMem = 0)

BEM BigEndianMem = 1

Type AccessType (see Table 2-2) sent to memory

Offset pAddr$_{2...0}$ sent to memory

Exceptions:

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

Reserved instruction exception (VR4100 in 32-bit mode)

LDR Load Doubleword Right LDR

31 26	25 21	20 16	15 0
LDR 0 1 1 0 1 1	base	rt	offset
6	5	5	16

Format:

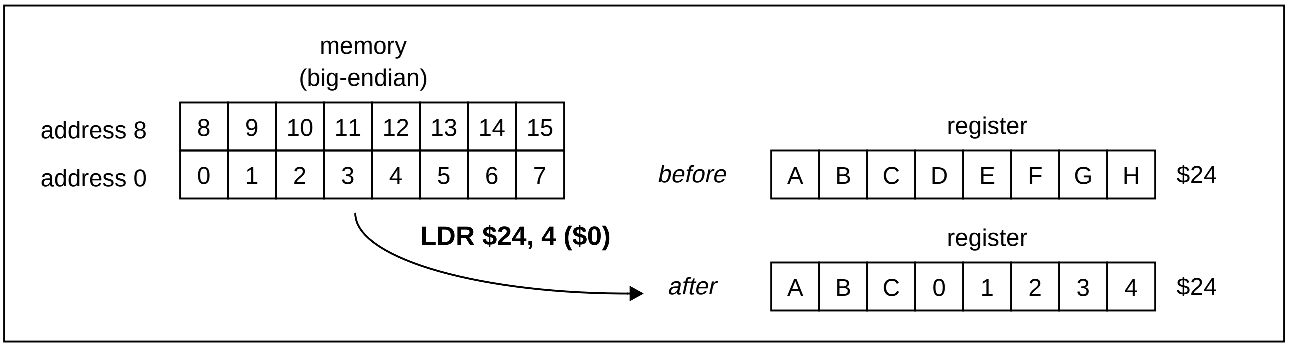
LDR rt, offset (base)

Description:

This instruction can be used in combination with the LDL instruction to load a register with eight consecutive bytes from memory, when the bytes cross a doubleword boundary. LDR loads the right portion of the register with the appropriate part of the low-order doubleword; LDL loads the left portion of the register with the appropriate part of the high-order doubleword.

The LDR instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which can specify an arbitrary byte. It reads bytes only from the doubleword in memory which contains the specified starting byte. From one to eight bytes will be loaded, depending on the starting byte specified.

Conceptually, it starts at the specified byte in memory and loads that byte into the low-order (right-most) byte of the register; then it loads bytes from memory into the register until it reaches the high-order byte of the doubleword in memory. The most significant (left-most) byte(s) of the register will not be changed.



LDR Load Doubleword Right (Continued) LDR

The contents of general register *rt* are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register *rt* and a following LDR (or LDL) instruction which also specifies register *rt*.

No address error exceptions due to alignment are possible.

This operation is only defined for the VR4100 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
T:   vAddr <- ($(\text{offset}_{15})^{48}$ || $\text{offset}_{15...0}$) + GPR [base]
     (pAddr, uncached) <- AddressTranslation (vAddr, DATA)
     pAddr <- $\text{pAddr}_{\text{PSIZE - 1...3}}$ || ($\text{pAddr}_{2...0}$ xor $\text{ReverseEndian}^3$)
     if BigEndianMem = 1 then
          pAddr <- $\text{pAddr}_{\text{PSIZE - 1...3}}$ || $0^3$
     endif
     byte <- $\text{vAddr}_{2...0}$ xor $\text{BigEndianCPU}^3$
     mem <- LoadMemory (uncached, DOUBLEWORD-byte, pAddr, vAddr, DATA)
     GPR [rt] <- GPR $[\text{rt}]_{63...64 - 8*\text{ byte}}$ || $\text{mem}_{63...8*\text{ byte}}$
```

LDR Load Doubleword Right (Continued) LDR

Given a doubleword in a register and a doubleword in memory, the operation of LDR is as follows:

LDR

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

vAddr$_{2..0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	destination	type	offset		destination	type	offset	
			LEM	BEM			LEM	BEM
0	I J K L M N O P	7	0	0	A B C D E F G I	0	7	0
1	A I J K L M N O	6	1	0	A B C D E F I J	1	6	0
2	A B I J K L M N	5	2	0	A B C D E I J K	2	5	0
3	A B C I J K L M	4	3	0	A B C D I J K L	3	4	0
4	A B C D I J K L	3	4	0	A B C I J K L M	4	3	0
5	A B C D E I J K	2	5	0	A B I J K L M N	5	2	0
6	A B C D E F I J	1	6	0	A I J K L M N O	6	1	0
7	A B C D E F G I	0	7	0	I J K L M N O P	7	0	0

LEM Little-endian memory (BigEndianMem = 0)

BEM BigEndianMem = 1

Type AccessType (see Table 2-2) sent to memory

Offset pAddr$_{2...0}$ sent to memory

Exceptions:

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

Reserved instruction exception (VR4100 in 32-bit mode)

LH Load Halfword LH

31 26	25 21	20 16	15 0
LH 1 0 0 0 0 1	base	rt	offset
6	5	5	16

Format:

LH rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the halfword at the memory location specified by the effective address are sign-extended and loaded into general register *rt*.

If the least-significant bit of the effective address is non-zero, an address error exception occurs.

Operation:

```
T:    vAddr <- $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15...0}) + \text{GPR [base]}$
      (pAddr, uncached) <- AddressTranslation (vAddr, DATA)
      pAddr <- $\text{pAddr}_{\text{PSIZE - 1...3}} \,||\, (\text{pAddr}_{2...0} \text{ xor } (\text{ReverseEndian}^{2} \,||\, 0))$
      mem <- LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
      byte <- $\text{vAddr}_{2...0} \text{ xor } (\text{BigEndianCPU}^{2} \,||\, 0)$
      GPR [rt] <- $(\text{mem}_{15 + 8^{*} \text{byte}})^{48} \,||\, \text{mem}_{15 + 8^{*} \text{byte}...8^{*} \text{byte}}$
```

Exceptions:

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

LHU Load Halfword Unsigned LHU

31 26	25 21	20 16	15 0
LHU 1 0 0 1 0 1	base	rt	offset
6	5	5	16

Format:

LHU rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the halfword at the memory location specified by the effective address are zero-extended and loaded into general register *rt*.

If the least-significant bit of the effective address is non-zero, an address error exception occurs.

Operation:

```
T:    vAddr <- $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15...0})$ + GPR [base]
      (pAddr, uncached) <- AddressTranslation (vAddr, DATA)
      pAddr <- $\text{pAddr}_{\text{PSIZE - 1...3}} \,||\, (\text{pAddr}_{2...0}$ xor $(\text{ReverseEndian}^{2} \,||\, 0))$
      mem <- LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
      byte <- $\text{vAddr}_{2...0}$ xor $(\text{BigEndianCPU}^{2} \,||\, 0)$
      GPR [rt] <- $0^{48} \,||\, \text{mem}_{15 + 8* \text{byte}...8* \text{byte}}$
```

Exceptions:

TLB refill exception Bus Error exception

TLB invalid exception Address error exception

LUI Load Upper Immediate LUI

31 26	25 21	20 16	15 0
LUI 0 0 1 1 1 1	0 0 0 0 0 0	rt	immediate
6	5	5	16

Format:

LUI rt, immediate

Description:

The 16-bit *immediate* is shifted left 16 bits and concatenated to 16 bits of zeros. The result is placed into general register *rt*. In 64-bit mode, the loaded word is sign-extended.

Operation:

```
T:    GPR [rt] <- $(\text{immediate}_{15})^{32}$ || immediate || $0^{16}$
```

Exceptions:

None

LW Load Word LW

31 26	25 21	20 16	15 0
LW 1 0 0 0 1 1	base	rt	offset
6	5	5	16

Format:

LW rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the word at the memory location specified by the effective address are loaded into general register *rt*. In 64-bit mode, the loaded word is sign-extended.

If either of the two least-significant bits of the effective address is non-zero, an address error exception occurs.

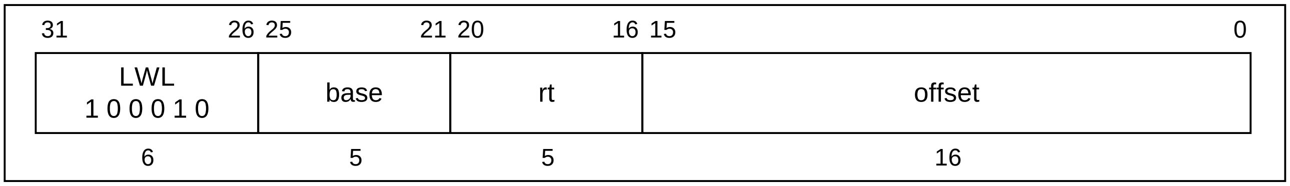
Operation:

```
T:    vAddr <- ((offset$_{15}$)$^{48}$ || offset$_{15...0}$) + GPR [base]
      (pAddr, uncached) <- AddressTranslation (vAddr, DATA)
      pAddr <- pAddr$_{PSIZE - 1...3}$ || (pAddr$_{2...0}$ xor (ReverseEndian || 0$^{2}$))
      mem <- LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
      byte <- vAddr$_{2...0}$ xor (BigEndianCPU || 0$^{2}$)
      GPR [rt] <- (mem$_{31 + 8* byte}$)$^{32}$ || mem$_{31 + 8* byte...8* byte}$
```

Exceptions:

TLB refill exception	Bus error exception
TLB invalid exception	Address error exception

LWL Load Word Left LWL



Format:

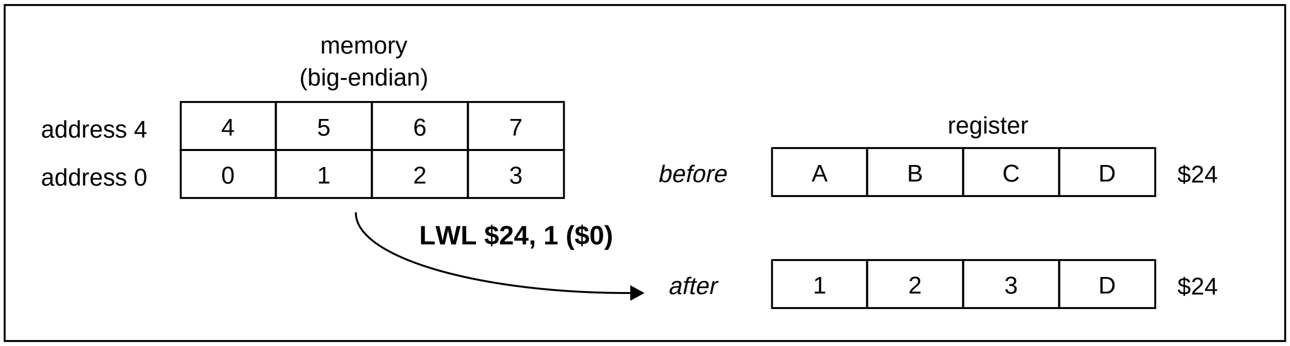
LWL rt, offset (base)

Description:

This instruction can be used in combination with the LWR instruction to load a register with four consecutive bytes from memory, when the bytes cross a word boundary. LWL loads the left portion of the register with the appropriate part of the high-order word; LWR loads the right portion of the register with the appropriate part of the low-order word.

The LWL instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which can specify an arbitrary byte. It reads bytes only from the word in memory which contains the specified starting byte. From one to four bytes will be loaded, depending on the starting byte specified. In 64-bit mode, the loaded word is sign-extended.

Conceptually, it starts at the specified byte in memory and loads that byte into the high-order (left-most) byte of the register; then it loads bytes from memory into the register until it reaches the low-order byte of the word in memory. The least-significant (right-most) byte(s) of the register will not be changed.



LWL Load Word Left (Continued) LWL

The contents of general register *rt* are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register *rt* and a following LWL (or LWR) instruction which also specifies register *rt*.

No address error exceptions due to alignment are possible.

Operation:

```
T:    vAddr <- ((offset$_{15}$)$^{48}$ || offset$_{15...0}$) + GPR [base]
      (pAddr, uncached) <- AddressTranslation (vAddr, DATA)
      pAddr <- pAddr$_{\text{PSIZE - 1...3}}$ || (pAddr$_{2...0}$ xor ReverseEndian$^{3}$)
      if BigEndianMem = 0 then
            pAddr <- pAddr$_{\text{PSIZE - 1...3}}$ || 0$^{3}$
      endif
      byte <- vAddr$_{1...0}$ xor BigEndianCPU$^{2}$
      word <- vAddr$_{2}$ xor BigEndianCPU
      mem <- LoadMemory (uncached, 0 || byte, pAddr, vAddr, DATA)
      temp <- mem$_{31 + 32* \text{word} - 8* \text{byte}...32* \text{word}}$ || GPR [rt]$_{23 - 8* \text{byte}...0}$
      GPR [rt] <- (temp$_{31}$)$^{32}$ || temp
```

LWL Load Word Left (Continued) LWL

Given a doubleword in a register and a doubleword in memory, the operation of LWL is as follows:

LWL

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

vAddr$_{2..0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	destination	type	offset		destination	type	offset	
			LEM	BEM			LEM	BEM
0	S S S S P F G H	0	0	7	S S S S I J K L	3	4	0
1	S S S S O P G H	1	0	6	S S S S J K L H	2	4	1
2	S S S S N O P H	2	0	5	S S S S K L G H	1	4	2
3	S S S S M N O P	3	0	4	S S S S L F G H	0	4	3
4	S S S S L F G H	0	4	3	S S S S M N O P	3	0	4
5	S S S S K L G H	1	4	2	S S S S N O P H	2	0	5
6	S S S S J K L H	2	4	1	S S S S O P G H	1	0	6
7	S S S S I J K L	3	4	0	S S S S P F G H	0	0	7

LEM Little-endian memory (BigEndianMem = 0)

BEM BigEndianMem = 1

Type AccessType (see Table 2-2) sent to memory

Offset pAddr$_{2...0}$ sent to memory

S sign-extend of destination$_{31}$

Exceptions:

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

LWR Load Word Right LWR

31 26	25 21	20 16	15 0
LWR 1 0 0 1 1 0	base	rt	offset
6	5	5	16

Format:

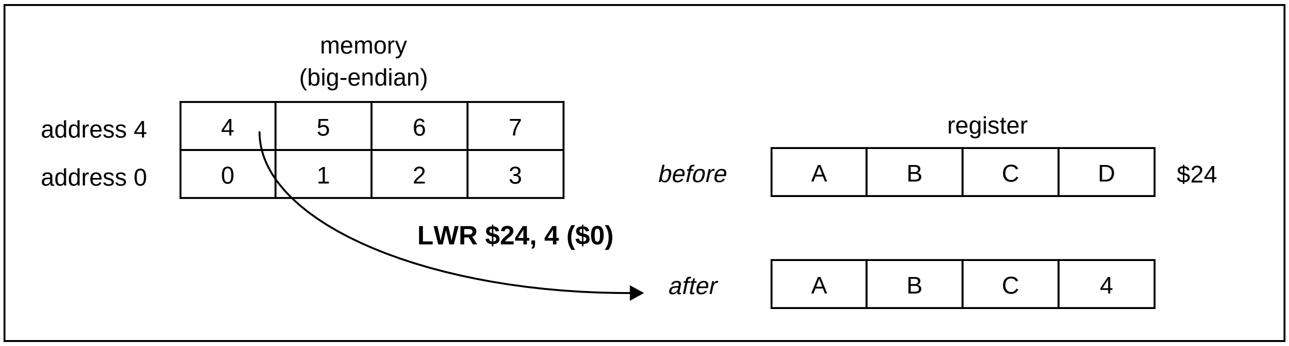
LWR rt, offset (base)

Description:

This instruction can be used in combination with the LWL instruction to load a register with four consecutive bytes from memory, when the bytes cross a word boundary. LWR loads the right portion of the register with the appropriate part of the low-order word; LWL loads the left portion of the register with the appropriate part of the high-order word.

The LWR instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which can specify an arbitrary byte. It reads bytes only from the word in memory which contains the specified starting byte. From one to four bytes will be loaded, depending on the starting byte specified. In 64-bit mode, the loaded word is sign-extended.

Conceptually, it starts at the specified byte in memory and loads that byte into the low-order (right-most) byte of the register; then it loads bytes from memory into the register until it reaches the high-order byte of the word in memory. The most significant (left-most) byte(s) of the register will not be changed.



LWR Load Word Right (Continued) LWR

The contents of general register *rt* are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register *rt* and a following LWR (or LWL) instruction which also specifies register *rt*.

No address error exceptions due to alignment are possible.

Operation:

```
T:   vAddr <- $((offset_{15})^{48}$ || $offset_{15...0})$ + GPR [base]
     (pAddr, uncached) <- AddressTranslation (vAddr, DATA)
     pAddr <- $pAddr_{PSIZE - 1...3}$ || $(pAddr_{2...0}$ xor $ReverseEndian^{3})$
     if BigEndianMem = 1 then
          pAddr <- $pAddr_{PSIZE - 1...3}$ || $0^{3}$
     endif
     byte <- $vAddr_{1...0}$ xor $BigEndianCPU^{2}$
     word <- $vAddr_{2}$ xor BigEndianCPU
     mem <- LoadMemory (uncached, 0 || byte, pAddr, vAddr, DATA)
     temp <- $GPR [rt]_{31...32 - 8* byte...0}$ || $mem_{32 + 32* word - 32* word + 8* byte}$
     GPR [rt] <- $(temp_{31})^{32}$ || temp
```

LWR Load Word Right (Continued) LWR

Given a word in a register and a word in memory, the operation of LWR is as follows:

LWR

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

vAddr$_{2..0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	destination	type	offset		destination	type	offset	
			LEM	BEM			LEM	BEM
0	S S S S M N O P	0	0	4	S S S S E F G I	0	7	0
1	S S S S E M N O	1	1	4	S S S S E F I J	1	6	0
2	S S S S E F M N	2	2	4	S S S S E I J K	2	5	0
3	S S S S E F G M	3	3	4	S S S S I J K L	3	4	0
4	S S S S I J K L	0	4	0	S S S S E F G M	0	3	4
5	S S S S E I J K	1	5	0	S S S S E F M N	1	2	4
6	S S S S E F I J	2	6	0	S S S S E M N O	2	1	4
7	S S S S E F G I	3	7	0	S S S S M N O P	3	0	4

LEM Little-endian memory (BigEndianMem = 0)

BEM BigEndianMem = 1

Type AccessType (see Table 2-2) sent to memory

Offset pAddr$_{2...0}$ sent to memory

S sign-extend of destination$_{31}$

Exceptions:

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

LWU Load Word Unsigned LWU

31 26	25 21	20 16	15 0
LWU 1 0 1 1 1 1	base	rt	offset
6	5	5	16

Format:

LWU rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of the word at the memory location specified by the effective address are loaded into general register *rt*. The loaded word is zero-extended.

If either of the two least-significant bits of the effective address is non-zero, an address error exception occurs.

This operation is only defined for the VR4100 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
T:  vAddr <- ((offset$_{15}$)$^{48}$ || offset$_{15...0}$) + GPR [base]
    (pAddr, uncached) <- AddressTranslation (vAddr, DATA)
    pAddr <- pAddr$_{PSIZE - 1...3}$ || (pAddr$_{2...0}$ xor (ReverseEndian || 0$^{2}$))
    mem <- LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
    byte <- vAddr$_{2...0}$ xor (BigEndianCPU || 0$^{2}$)
    GPR [rt] <- 0$^{32}$ || mem$_{31 + 8* byte...8* byte}$
```

Exceptions:

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

Reserved instruction exception (VR4100 in 32-bit mode)

MADD16 Multiply and Add 16-bit integer MADD16

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	0 0 0 0 0 0 0 0 0 0 0	DMADD16 1 0 1 0 0 0
6	5	5	10	6

Format:

MADD16 rs, rt

Description:

The contents of general registers *rs* and *rt* are multiplied, treating both operands as 16-bit 2's complement values. The operand[62:15] must be valid 15-bit, sign-extended values. If not, the results is unpredictable.

This multiplied result and the 64-bit data joined special register *HI* to *LO* are added to form the result.

No integer overflow exception occurs under any circumstances.

When the operation completes, the low-order word of the double result is loaded into special register *LO*, and the high-order word of the double result is loaded into special register *HI*.

The following Table are hazard cycles between MADD16 and other instructions.

MULT/MULTU -> MADD16	1 Cycle
DMULT/DMULTU -> MADD16	4 Cycles
DIV/DIVU -> MADD16	36 Cycles
DDIV/DDIVU -> MADD16	68 Cycles
MFHI/MFLO -> MADD16	2 Cycles
DMADD16 -> MADD16	0 Cycle
MADD16 -> MADD16	0 Cycle

MADD16 Multiply and Add 16-bit integer (Continued) MADD16

MADD16 -> MULT/MULTU	0 Cycle
MADD16 -> DMULT/DMULTU	0 Cycle
MADD16 -> DIV/DIVU	1 Cycle
MADD16 -> DDIV/DDIVU	1 Cycle
MADD16 -> MFHI/MFLO	0 Cycle
MADD16 -> DMADD16	0 Cycle
MADD16 -> MADD16	0 Cycle

Operation:

```
T-2:  LO     <- undefined
      HI     <- undefined
T-1:  LO     <- undefined
      HI     <- undefined
T:    temp1  <- GPR [rs] * GPR [rt]
      temp2  <- temp1 + ($HI_{31...0}$ || $LO_{31...0}$)
      LO     <- $(temp2_{31})^{32}$ || $temp2_{31...0}$
      HI     <- $(temp2_{63})^{32}$ || $temp2_{63...32}$
```

Exceptions:

None

MFC0 Move From System Control Coprocessor MFC0

31 26	25 21	20 16	15 11	10 0
COP0 0 1 0 0 0 0	MF 0 0 0 0 0	rt	rd	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format:

MFC0 rt, rd

Description:

The contents of coprocessor register *rd* of the CP0 are loaded into general register *rt.*

When using a register used by the MFC0 by means of instructions before and after it, refer to **Appendix A VR4100 Coprocessor 0 Hazards** and place the instructions in the appropriate location.

Operation:

```
T:    data <- CPR [0, rd]
T+1:  GPR [rt] <- $(data_{31})^{32}$ || $data_{31...0}$
```

Exceptions:

Coprocessor unusable exception (user and supervisor mode if CP0 not enabled)

MFHI Move From HI MFHI

31 26	25 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0	rd	0 0 0 0 0 0	MFHI 0 1 0 0 0 0
6	10	5	5	6

Format:

MFHI rd

Description:

The contents of special register *HI* are loaded into general register *rd*.

To ensure proper operation in the event of interruptions, the two instructions which follow a MFHI instruction may not be any of the instructions which modify the *HI* register: MULT, MULTU, DIV, DIVU, MTHI, DMULT, DMULTU, DDIV, DDIVU.

Operation:

```
T:     GPR [rd] <- HI
```

Exceptions:

None

MFLO   Move From LO   MFLO

31 26	25 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0	rd	0 0 0 0 0 0	MFLO 0 1 0 0 1 0
6	10	5	5	6

Format:

MFLO rd

Description:

The contents of special register *LO* are loaded into general register *rd*.

To ensure proper operation in the event of interruptions, the two instructions which follow a MFLO instruction may not be any of the instructions which modify the *LO* register: MULT, MULTU, DIV, DIVU, MTLO, DMULT, DMULTU, DDIV, DDIVU.

Operation:

```
T:     GPR [rd] <- LO
```

Exceptions:

None

MTC0 Move To Coprocessor0 MTC0

31 26	25 21	20 16	15 11	10 0
COP0 0 1 0 0 0 0	MT 0 0 1 0 0	rt	rd	0 0 0 0 0 0 0 0 0 0 0 0
6	5	5	5	11

Format:

MTC0 rt, rd

Description:

The contents of general register *rt* are loaded into coprocessor register *rd* of coprocessor 0.

Because the state of the virtual address translation system may be altered by this instruction, the operation of load instructions, store instructions, and TLB operations immediately prior to and after this instruction are undefined.

When using a register used by the MTC0 by means of instructions before and after it, refer to **Appendix A VR4100 Coprocessor 0 Hazards** and place the instructions in the appropriate location.

Operation:

```
T:     data <- GPR [rt]
T+1: CPR [0, rd] <- data
```

Exceptions:

Coprocessor unusable exception (user and supervisor mode if CP0 not enabled)

MTHI   Move To HI   MTHI

31 26	25 21	20 6	5 0
SPECIAL 0 0 0 0 0 0	rs	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	MTHI 0 1 0 0 0 1
6	5	15	6

Format:

MTHI rs

Description:

The contents of general register *rs* are loaded into special register *HI*.

If a MTHI operation is executed following a MULT, MULTU, DIV, or DIVU instruction, but before any MFLO, MFHI, MTLO, or MTHI instructions, the contents of special register *HI* are undefined.

Operation:

```
T-2:  HI <- undefined
T-1:  HI <- undefined
T:    HI <- GPR [rs]
```

Exceptions:

None

MTLO Move To LO MTLO

31 26	25 21	20 6	5 0
SPECIAL 0 0 0 0 0 0	rs	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	MTLO 0 1 0 0 1 1
6	5	15	6

Format:

MTLO rs

Description:

The contents of general register *rs* are loaded into special register *LO.*

If a MTLO operation is executed following a MULT, MULTU, DIV, or DIVU instruction, but before any MFLO, MFHI, MTLO, or MTHI instructions, the contents of special register *LO* are undefined.

Operation:

```
T-2:  LO <- undefined
T-1:  LO <- undefined
T:    LO <- GPR [rs]
```

Exceptions:

None

MULT Multiply MULT

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	0 0 0 0 0 0 0 0 0 0 0	MULT 0 1 1 0 0 0
6	5	5	10	6

Format:

MULT rs, rt

Description:

The contents of general registers *rs* and *rt* are multiplied, treating both operands as 32-bit 2's complement values. No integer overflow exception occurs under any circumstances. In 64-bit mode, the operands must be valid 32-bit, sign-extended values.

When the operation completes, the low-order word of the double result is loaded into special register *LO*, and the high-order word of the double result is loaded into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results of these instructions are undefined. Correct operation requires separating reads of *HI* or *LO* from writes by a minimum of two other instructions.

Operation:

```
T-2:  LO  <- undefined
      HI  <- undefined
T-1:  LO  <- undefined
      HI  <- undefined
T:    t   <- GPR [rs]$_{31...0}$ * GPR [rt]$_{31...0}$
      LO  <- $(t_{31})^{32}$ || $t_{31...0}$
      HI  <- $(t_{63})^{32}$ || $t_{63...32}$
```

Exceptions:

None

MULTU Multiply Unsigned MULTU

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 00 0000 0000	MULTU 011001
6	5	5	10	6

Format:

MULTU rs, rt

Description:

The contents of general register *rs* and the contents of general register *rt* are multiplied, treating both operands as unsigned values. No overflow exception occurs under any circumstances. In 64-bit mode, the operands must be valid 32-bit, sign-extended values.

When the operation completes, the low-order word of the double result is loaded into special register *LO*, and the high-order word of the double result is loaded into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results of these instructions are undefined. Correct operation requires separating reads of *HI* or *LO* from writes by a minimum of two instructions.

Operation:

```
T-2:  LO  <- undefined
      HI  <- undefined
T-1:  LO  <- undefined
      HI  <- undefined
T:    t   <- GPR [rs]$_{31...0}$ * GPR [rt]$_{31...0}$
      LO  <- $(t_{31})^{32}$ || $t_{31...0}$
      HI  <- $(t_{63})^{32}$ || $t_{63...32}$
```

Exceptions:

None

NOR Nor NOR

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	NOR 1 0 0 1 1 1
6	5	5	5	5	6

Format:

NOR rd, rs, rt

Description:

The contents of general register *rs* are combined with the contents of general register *rt* in a bit-wise logical NOR operation. The result is placed into general register *rd*.

Operation:

```
T:    GPR [rd] <- GPR [rs] nor GPR [rt]
```

Exceptions:

None

OR　　Or　　OR

31　　26	25　　21	20　　16	15　　11	10　　6	5　　0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	OR 1 0 0 1 0 1
6	5	5	5	5	6

Format:

OR rd, rs, rt

Description:

The contents of general register *rs* are combined with the contents of general register *rt* in a bit-wise logical OR operation. The result is placed into general register *rd*.

Operation:

```
T:    GPR [rd] <- GPR [rs] or GPR [rt]
```

Exceptions:

None

ORI Or Immediate ORI

31 26	25 21	20 16	15 0
ORI 0 0 1 1 0 1	rs	rt	immediate
6	5	5	16

Format:

ORI rt, rs, immediate

Description:

The 16-bit *immediate* is zero-extended and combined with the contents of general register *rs* in a bit-wise logical OR operation. The result is placed into general register *rt*.

Operation:

```
T:    GPR [rt] <- GPR [rs]$_{63...16}$ || (immediate or GPR [rs]$_{15...0}$)
```

Exceptions:

None

SB Store Byte SB

31 26	25 21	20 16	15 0
SB 1 0 1 0 0 0	base	rt	offset
6	5	5	16

Format:

SB rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The least-significant byte of register *rt* is stored at the effective address.

Operation:

```
T:   vAddr <- ((offset$_{15}$)$^{48}$ || offset$_{15...0}$) + GPR [base]
     (pAddr, uncached) <- AddressTranslation (vAddr, DATA)
     pAddr <- pAddr$_{\text{PSIZE - 1...3}}$ || (pAddr$_{2...0}$ xor (ReverseEndian$^{3}$))
     byte <- vAddr$_{2...0}$ xor BigEndianCPU$^{3}$
     data <- GPR [rt]$_{63 - 8^{*} \text{byte}...0}$ || 0$^{8^{*} \text{byte}}$
     StoreMemory (uncached, BYTE, data, pAddr, vAddr, DATA)
```

Exceptions:

TLB refill exception

TLB invalid exception

TLB modification exception

Bus error exception

Address error exception

SD Store Doubleword SD

31 26	25 21	20 16	15 0
SD 1 1 1 1 1 1	base	rt	offset
6	5	5	16

Format:

SD rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of general register *rt* are stored at the memory location specified by the effective address.

If either of the three least-significant bits of the effective address are non-zero, an address error exception occurs.

This operation is only defined for the VR4100 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
T:    vAddr <- ((offset$_{15}$)$^{48}$ || offset$_{15...0}$) + GPR [base]
      (pAddr, uncached) <- AddressTranslation (vAddr, DATA)
      data <- GPR [rt]
      StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
```

Exceptions:

TLB refill exception

TLB invalid exception

TLB modification exception

Bus error exception

Address error exception

Reserved instruction exception (VR4100 in 32-bit user mode
VR4100 in 32-bit supervisor mode)

SDL Store Doubleword Left SDL

31 26	25 21	20 16	15 0
SDL 1 0 1 1 0 0	base	rt	offset
6	5	5	16

Format:

SDL rt, offset (base)

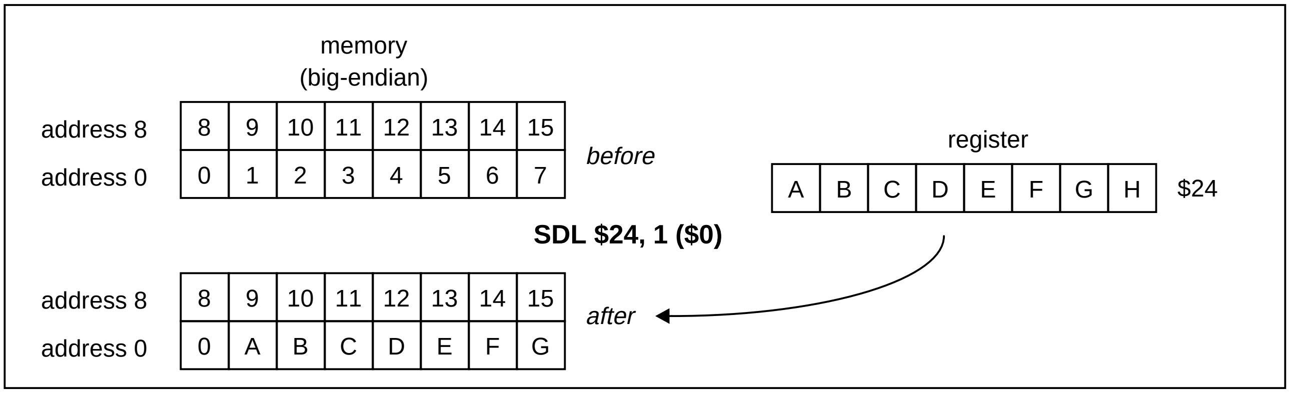
Description:

This instruction can be used with the SDR instruction to store the contents of a register into eight consecutive bytes of memory, when the bytes cross a doubleword boundary. SDL stores the left portion of the register into the appropriate part of the high-order doubleword of memory; SDR stores the right portion of the register into the appropriate part of the low-order doubleword.

The SDL instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which may specify an arbitrary byte. It alters only the word in memory which contains that byte. From one to four bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the most-significant byte of the register and copies it to the specified byte in memory; then it copies bytes from register to memory until it reaches the low-order byte of the word in memory.

No address error exceptions due to alignment are possible.



SDL Store Doubleword Left (Continued) SDL

This operation is only defined for the VR4100 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
T:   vAddr <- ((offset$_{15}$)$^{48}$ || offset$_{15...0}$) + GPR [base]
     (pAddr, uncached) <- AddressTranslation (vAddr, DATA)
     pAddr <- pAddr$_{\text{PSIZE - 1...3}}$ || (pAddr$_{2...0}$ xor ReverseEndian$^{3}$)
     if BigEndianMem = 0 then
          pAddr <- pAddr$_{31...3}$ || 0$^{3}$
     endif
     byte <- vAddr$_{2...0}$ xor BigEndianCPU$^{3}$
     data <- 0$^{56 - 8* \text{byte}}$ || GPR [rt]$_{63...56 - 8* \text{byte}}$
     Storememory (uncached, byte, data, pAddr, vAddr, DATA)
```

SDL Store Doubleword Left (Continued) SDL

Given a doubleword in a register and a doubleword in memory, the operation of SDL is as follows:

SDL

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

$vAddr_{2..0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	destination	type	offset		destination	type	offset	
			LEM	BEM			LEM	BEM
0	I J K L M N O A	0	0	7	A B C D E F G H	7	0	0
1	I J K L M N A B	1	0	6	I A B C D E F G	6	0	1
2	I J K L M A B C	2	0	5	I J A B C D E F	5	0	2
3	I J K L A B C D	3	0	4	I J K A B C D E	4	0	3
4	I J K A B C D E	4	0	3	I J K L A B C D	3	0	4
5	I J A B C D E F	5	0	2	I J K L M A B C	2	0	5
6	I A B C D E F G	6	0	1	I J K L M N A B	1	0	6
7	A B C D E F G H	7	0	0	I J K L M N O A	0	0	7

LEM	Little-endian memory (BigEndianMem = 0)
BEM	BigEndianMem = 1
Type	AccessType (see Table 2-2) sent to memory
Offset	$pAddr_{2...0}$ sent to memory

Exceptions:

TLB refill exception

TLB invalid exception

TLB modification exception

Bus error exception

Address error exception

Reserved instruction exception (VR4100 in 32-bit mode)

SDR Store Doubleword Right SDR

31 26	25 21	20 16	15 0
SDR 1 0 1 1 0 1	base	rt	offset
6	5	5	16

Format:

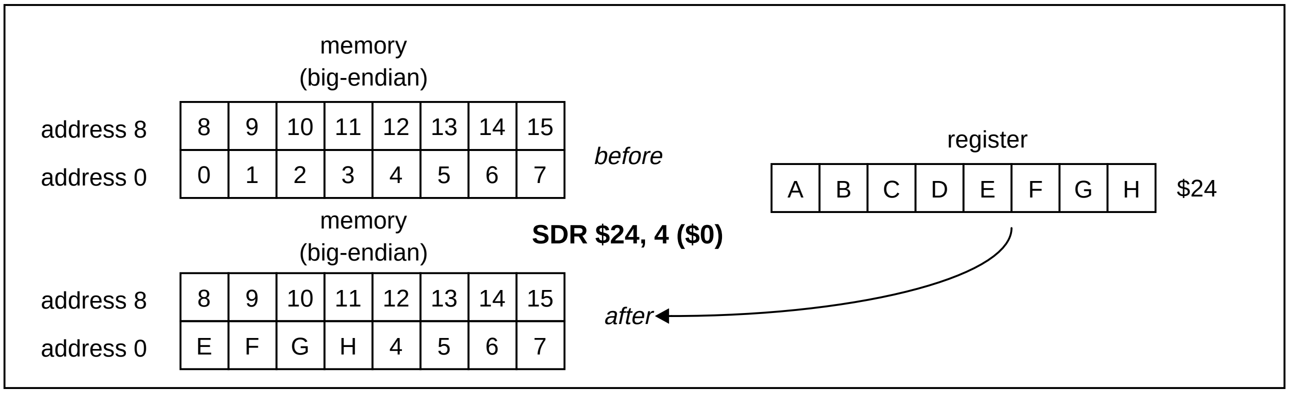
SDR rt, offset (base)

Description:

This instruction can be used with the SDL instruction to store the contents of a register into eight consecutive bytes of memory, when the bytes cross a boundary between two doublewords. SDR stores the right portion of the register into the appropriate part of the low-order doubleword; SDL stores the left portion of the register into the appropriate part of the low-order doubleword of memory.

The SDR instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which may specify an arbitrary byte. It alters only the word in memory which contains that byte. From one to eight bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the least-significant (rightmost) byte of the register and copies it to the specified byte in memory; then it copies bytes from register to memory until it reaches the high-order byte of the word in memory. No address error exceptions due to alignment are possible.



SDR   Store Doubleword Right (Continued)   SDR

This operation is only defined for the VR4100 operating in 64-bit mode. Execution of this instruction in 32-bit mode causes a reserved instruction exception.

Operation:

```
T:   vAddr <- ((offset$_{15}$)$^{48}$ || offset$_{15...0}$) + GPR [base]
     (pAddr, uncached) <- AddressTranslation (vAddr, DATA)
     pAddr <- pAddr$_{\text{PSIZE - 1...3}}$ || (pAddr$_{2...0}$ xor ReverseEndian$^{3}$)
     if BigEndianMem = 0 then
          pAddr <- pAddr$_{\text{PSIZE - 1...3}}$ || 0$^{3}$
     endif
     byte <- vAddr$_{2...0}$ xor BigEndianCPU$^{3}$
     data <- GPR [rt]$_{63 - 8^{*} \text{byte}}$ || 0$^{8^{*} \text{byte}}$
     StoreMemory (uncached, DOUBLEWORD-byte, data, pAddr, vAddr, DATA)
```

SDR Store Doubleword Right (Continued) SDR

Given a doubleword in a register and a doubleword in memory, the operation of SDR is as follows:

SDR

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

$vAddr_{2..0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	destination	type	offset		destination	type	offset	
			LEM	BEM			LEM	BEM
0	A B C D E F G H	7	0	0	H J K L M N O P	0	7	0
1	B C D E F G H P	6	1	0	G H K L M N O P	1	6	0
2	C D E F G H O P	5	2	0	F G H L M N O P	2	5	0
3	D E F G H N O P	4	3	0	E F G H M N O P	3	4	0
4	E F G H M N O P	3	4	0	D E F G H N O P	4	3	0
5	F G H L M N O P	2	5	0	C D E F G H O P	5	2	0
6	G H K L M N O P	1	6	0	B C D E F G H P	6	1	0
7	H J K L M N O P	0	7	0	A B C D E F G H	7	0	0

LEM Little-endian memory (BigEndianMem = 0)

BEM BigEndianMem = 1

Type AccessType (see Table 2-2) sent to memory

Offset $pAddr_{2...0}$ sent to memory

Exceptions:

TLB refill exception

TLB invalid exception

TLB modification exception

Bus error exception

Address error exception

Reserved instruction exception (VR4100 in 32-bit mode)

SH Store Halfword SH

31 26	25 21	20 16	15 0
SH 1 0 1 0 0 1	base	rt	offset
6	5	5	16

Format:

SH rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form an unsigned effective address. The least-significant halfword of register *rt* is stored at the effective address. If the least-significant bit of the effective address is non-zero, an address error exception occurs.

Operation:

```
T:   vAddr <- ($(\text{offset}_{15})^{48}$ || $\text{offset}_{15...0}$) + GPR [base]
     (pAddr, uncached) <- AddressTranslation (vAddr, DATA)
     pAddr <- $\text{pAddr}_{\text{PSIZE - 1...3}}$ || ($\text{pAddr}_{2...0}$ xor ($\text{ReverseEndian}^{2}$ || 0))
     byte <- $\text{vAddr}_{2...0}$ xor ($\text{BigEndianCPU}^{2}$ || 0)
     data <- GPR $[\text{rt}]_{63 - 8* \text{byte}...0}$ || $0^{8* \text{byte}}$
     StoreMemory (uncached, HALFWORD, data, pAddr, vAddr, DATA)
```

Exceptions:

TLB refill exception

TLB invalid exception

TLB modification exception

Bus error exception

Address error exception

SLL   Shift Left Logical   SLL

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	sa	SLL 0 0 0 0 0 0
6	5	5	5	5	6

Format:

SLL rd, rt, sa

Description:

The contents of general register *rt* are shifted left by *sa* bits, inserting zeros into the low-order bits.

The result is placed in register *rd.*

In 64-bit mode, the 32-bit result is sign-extended when placed in the destination register. It is sign extended for all shift amounts, including zero; SLL with zero shift amount truncates a 64-bit value to 32 bits and then sign extends this 32-bit value. SLL, unlike nearly all other word operations, does not require an operand to be a properly sign-extended word value to produce a valid sign-extended word result.

Note: SLL with a shift amount of zero may be treated as a NOP by some assemblers, at some optimization levels. If using SLL with a zero shift to truncate 64-bit values, check the assembler you are using.

Operation:

```
T:    s <- 0 || sa
      temp <- GPR [rt]$_{31 - s...0}$ || $0^{s}$
      GPR [rd] <- (temp$_{31}$)$^{32}$ || temp
```

Exceptions:

None

SLLV   Shift Left Logical Variable   SLLV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0	rt	rd	0 0 0 0 0 0	SLLV 0 0 0 1 0 0
6	5	5	5	5	6

Format:

SLLV rd, rt, rs

Description:

The contents of general register *rt* are shifted left the number of bits specified by the low-order five bits contained in general register *rs*, inserting zeros into the low-order bits.

The result is placed in register *rd*.

In 64-bit mode, the 32-bit result is sign-extended when placed in the destination register. It is sign extended for all shift amounts, including zero; SLLV with zero shift amount truncates a 64-bit value to 32 bits and then sign extends this 32-bit value. SLLV, unlike nearly all other word operations, does not require an operand to be a properly sign-extended word value to produce a valid sign-extended word result.

Note: SLLV with a shift amount of zero may be treated as a NOP by some assemblers, at some optimization levels. If using SLLV with a zero shift to truncate 64-bit values, check the assembler you are using.

Operation:

```
T:    s <- 0 || GP [rs]$_{4...0}$
      temp <- GPR [rt]$_{(31 - s)...0}$ || 0$^{s}$
      GPR [rd] <- (temp$_{31}$)$^{32}$ || temp
```

Exceptions:

None

SLT Set On Less Than SLT

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	SLT 1 0 1 0 1 0
6	5	5	5	5	6

Format:

SLT rd, rs, rt

Description:

The contents of general register *rt* are subtracted from the contents of general register *rs*. Considering both quantities as signed integers, if the contents of general register *rs* are less than the contents of general register *rt*, the result is set to one; otherwise the result is set to zero.

The result is placed into general register *rd*.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflows.

Operation:

```
T:    if GPR [rs] < GPR [rt] then
          GPR [rd] <- $0^{63}$ || 1
      else
          GPR [rd] <- $0^{64}$
      endif
```

Exceptions:

None

SLTI Set On Less Than Immediate SLTI

31 26	25 21	20 16	15 0
SLTI 0 0 1 0 1 0	rs	rt	immediate
6	5	5	16

Format:

SLTI rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and subtracted from the contents of general register *rs.* Considering both quantities as signed integers, if *rs* is less than the sign-extended immediate, the result is set to one; otherwise the result is set to zero.

The result is placed into general register *rt.*

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflows.

Operation:

```
T:   if GPR [rs] < $(\text{immediate}_{15})^{48}$ || $\text{immediate}_{15...0}$ then
          GPR [rd] <- $0^{63}$ || 1
     else
          GPR [rd] <- $0^{64}$
     endif
```

Exceptions:

None

SLTIU   Set On Less Than Immediate Unsigned   SLTIU

31 26	25 21	20 16	15 0
SLTIU 0 0 1 0 1 1	rs	rt	immediate
6	5	5	16

Format:

SLTIU rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and subtracted from the contents of general register *rs.* Considering both quantities as unsigned integers, if *rs* is less than the sign-extended immediate, the result is set to one; otherwise the result is set to zero.

The result is placed into general register *rt*.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflows.

Operation:

```
T:    if (0 || GPR [rs]) < (0 || $(\text{immediate}_{15})^{48}$ || $\text{immediate}_{15...0}$) then
              GPR [rd] <- $0^{63}$ || 1
      else
              GPR [rd] <- $0^{64}$
      endif
```

Exceptions:

None

SLTU Set On Less Than Unsigned SLTU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	SLTU 1 0 1 0 1 1
6	5	5	5	5	6

Format:

SLTU rd, rs, rt

Description:

The contents of general register *rt* are subtracted from the contents of general register *rs.* Considering both quantities as unsigned integers, if the contents of general register *rs* are less than the contents of general register *rt*, the result is set to one; otherwise the result is set to zero.

The result is placed into general register *rd*.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflows.

Operation:

```
T:    if (0 || GPR [rs]) < 0 || GPR [rt] then
              GPR [rd] <- $0^{63}$ || 1
      else
              GPR [rd] <- $0^{64}$
      endif
```

Exceptions:

None

SRA Shift Right Arithmetic SRA

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0	rt	rd	sa	SRA 0 0 0 0 1 1
6	5	5	5	5	6

Format:

SRA rd, rt, sa

Description:

The contents of general register *rt* are shifted right by *sa* bits, sign-extending the high-order bits.

The result is placed in register *rd*.

In 64-bit mode, the operand must be a valid sign-extended, 32-bit value.

Operation:

```
T:   $s \leftarrow 0 \| sa$
     $temp \leftarrow (GPR[rt]_{31})^{s} \| GPR[rt]_{31...s}$
     $GPR[rd] \leftarrow (temp_{31})^{32} \| temp$
```

Exceptions:

None

SRAV Shift Right Arithmetic Variable SRAV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	SRAV 0 0 0 1 1 1
6	5	5	5	5	6

Format:

SRAV rd, rt, rs

Description:

The contents of general register *rt* are shifted right by the number of bits specified by the low-order five bits of general register *rs*, sign-extending the high-order bits.

The result is placed in register *rd*.

In 64-bit mode, the operand must be a valid sign-extended, 32-bit value.

Operation:

```
T:   s <- GPR [rs]$_{4...0}$
     temp <- (GPR [rt]$_{31}$)$^{s}$ || GPR [rt]$_{31...s}$
     GPR [rd] <- (temp$_{31}$)$^{32}$ || temp
```

Exceptions:

None

SRL Shift Right Logical SRL

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0	rt	rd	sa	SRL 0 0 0 0 1 0
6	5	5	5	5	6

Format:

SRL rd, rt, sa

Description:

The contents of general register *rt* are shifted right by *sa* bits, inserting zeros into the high-order bits.

The result is placed in register *rd*.

In 64-bit mode, the operand must be a valid sign-extended, 32-bit value.

Operation:

```
T:    s <- 0 || sa
      temp <- $0^{s}$ || GPR [rt]$_{31\ldots s}$
      GPR [rd] <- $(\mathrm{temp}_{31})^{32}$ || temp
```

Exceptions:

None

SRLV Shift Right Logical Variable SRLV

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	SRLV 0 0 0 1 1 0
6	5	5	5	5	6

Format:

SRLV rd, rt, rs

Description:

The contents of general register *rt* are shifted right by the number of bits specified by the low-order five bits of general register *rs,* inserting zeros into the high-order bits.

The result is placed in register *rd*.

In 64-bit mode, the operand must be a valid sign-extended, 32-bit value.

Operation:

```
T:    s <- GPR [rs]$_{4...0}$
      temp <- $0^{s}$ || GPR [rt]$_{31...s}$
      GPR [rd] <- $(temp_{31})^{32}$ || temp
```

Exceptions:

None

STANDBY Standby STANDBY

31 26	25	24 6	5 0
COP0 0 1 0 0 0 0	CO 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	STANDBY 1 0 0 0 0 1
6	1	19	6

Format:

STANDBY

Description:

In Standby mode, all internal clocks, except Timer/Interrupt unit, are frozen at high level.

To enter Standby mode from Fullspeed mode, first execute the STANDBY instruction. When the STANDBY instruction finishes the WB stage, the VR4100 wait by the SysAD bus is idle state, after then the internal clocks will shut down, thus freezing the pipeline. The PLL, Timer/Interrupt clocks and the system interface clocks, TClock and MasterOut, will continue to run.

Once the VR4100 is in Standby mode, any interrupt, including the internally generated timer interrupt, NMI, SoftReset, and ColdReset will cause the VR4100 to exit Standby mode and to enter Fullspeed mode.

Operation:

```
T:
T+1: standby operation ()
```

Exceptions:

Coprocessor unusable exception

SUB Subtract SUB

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	SUB 1 0 0 0 1 0
6	5	5	5	5	6

Format:

SUB rd, rs, rt

Description:

The contents of general register *rt* are subtracted from the contents of general register *rs* to form a result. The result is placed into general register *rd.* In 64-bit mode, the operands must be valid sign-extended, 32-bit values.

The only difference between this instruction and the SUBU instruction is that SUBU never traps on overflow.

An integer overflow exception takes place if the carries out of bits 30 and 31 differ 2's complement overflow). The destination register *rd* is not modified when an integer overflow exception occurs.

Operation:

```
T:    temp <- GPR [rs] - GPR [rt]
      GPR [rd] <- $(temp_{31})^{32}$ || $temp_{31...0}$
```

Exceptions:

Integer overflow exception

SUBU Subtract Unsigned SUBU

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	SUBU 1 0 0 0 1 1
6	5	5	5	5	6

Format:

SUBU rd, rs, rt

Description:

The contents of general register *rt* are subtracted from the contents of general register *rs* to form a result.

The result is placed into general register *rd*.

In 64-bit mode, the operands must be valid sign-extended, 32-bit values.

The only difference between this instruction and the SUB instruction is that SUBU never traps on overflow. No integer overflow exception occurs under any circumstances.

Operation:

```
T:    temp <- GPR [rs] - GPR [rt]
      GPR [rd] <- $(\text{temp}_{31})^{32}$ || $\text{temp}_{31...0}$
```

Exceptions:

None

SUSPEND Suspend SUSPEND

31 26	25	24 6	5 0
COP0 0 1 0 0 0 0	CO 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SUSPEND 1 0 0 0 1 0
6	1	19	6

Format:

SUSPEND

Description:

In Suspend mode, all internal clocks, except Timer/Interrupt unit, and the TClock are frozen at high level.

To enter Suspend mode from Fullspeed mode, first execute the SUSPEND instruction. When the SUSPEND instruction finishes the WB stage, the VR4100 wait by the SysAD bus is idle state, after then the internal clocks and the TClock will shut down, thus freezing the pipeline. The PLL, Timer/Interrupt clocks and MasterOut, will continue to run.

Once the VR4100 is in Suspend mode, any interrupt, including the internally generated timer interrupt, NMI, SoftReset and ColdReset will cause the VR4100 to exit Suspend mode and to enter Fullspeed mode.

Operation:

```
T:
T+1: suspend operation ()
```

Exceptions:

Coprocessor unusable exception

SW Store Word SW

31 26	25 21	20 16	15 0
SW 1 0 1 0 1 1	base	rt	offset
6	5	5	16

Format:

SW rt, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general register *base* to form a virtual address. The contents of general register *rt* are stored at the memory location specified by the effective address.

If either of the two least-significant bits of the effective address are non-zero, an address error exception occurs.

Operation:

```
T:  vAddr <- $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15...0}) + \text{GPR [base]}$
    (pAddr, uncached) <- AddressTranslation (vAddr, DATA)
    pAddr <- $\text{pAddr}_{\text{PSIZE - 1...3}} \,||\, (\text{pAddr}_{2...0} \text{ xor } (\text{ReverseEndian} \,||\, 0^{2}))$
    byte <- $\text{vAddr}_{2...0} \text{ xor } (\text{BigEndianCPU} \,||\, 0^{2})$
    data <- $\text{GPR [rt]}_{63 - 8^{*}\text{ byte}} \,||\, 0^{8^{*}\text{ byte}}$
    StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
```

Exceptions:

TLB refill exception

TLB invalid exception

TLB modification exception

Bus error exception

Address error exception

SWL Store Word Left SWL

31 26	25 21	20 16	15 0
SWL 1 0 1 0 1 0	base	rt	offset
6	5	5	16

Format:

SWL rt, offset (base)

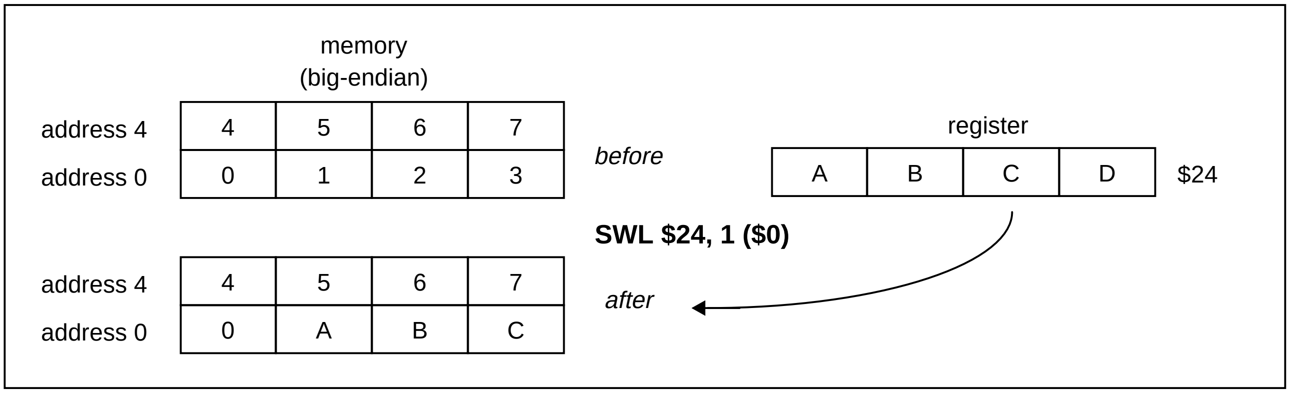
Description:

This instruction can be used with the SWR instruction to store the contents of a register into four consecutive bytes of memory, when the bytes cross a word boundary. SWL stores the left portion of the register into the appropriate part of the high-order word of memory; SWR stores the right portion of the register into the appropriate part of the low-order word.

The SWL instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which may specify an arbitrary byte. It alters only the word in memory which contains that byte. From one to four bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the most-significant byte of the register and copies it to the specified byte in memory; then it copies bytes from register to memory until it reaches the low-order byte of the word in memory.

No address error exceptions due to alignment are possible.



SWL Store Word Left (Continued) SWL

Operation:

```
T:  vAddr <- $((\text{offset}_{15})^{48} \| \text{offset}_{15...0}) + \text{GPR [base]}$
    (pAddr, uncached) <- AddressTranslation (vAddr, DATA)
    pAddr <- $\text{pAddr}_{\text{PSIZE - 1...3}} \| (\text{pAddr}_{2...0}$ xor $\text{ReverseEndian}^3)$
    if BigEndianMem = 0 then
        pAddr <- $\text{pAddr}_{31...2} \| 0^2$
    endif
    byte <- $\text{vAddr}_{1...0}$ xor $\text{BigEndianCPU}^2$
    if ($\text{vAddr}_2$ xor BigEndianCPU) = 0 then
        data <- $0^{32} \| 0^{24 - 8^* \text{byte}} \| \text{GPR [rt]}_{31...24 - 8^* \text{byte}}$
    else
        data <- $0^{24 - 8^* \text{byte}} \| \text{GPR [rt]}_{31...24 - 8^* \text{byte}} \| 0^{32}$
    endif
    StoreMemory (uncached, byte, data, pAddr, vAddr, DATA)
```

SWL Store Word Left (Continued) SWL

Given a doubleword in a register and a doubleword in memory, the operation of SWL is as follows:

SWL

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

vAddr$_{2..0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	destination	type	offset		destination	type	offset	
			LEM	BEM			LEM	BEM
0	I J K L M N O E	0	0	7	E F G H M N O P	3	4	0
1	I J K L M N E F	1	0	6	I E F G M N O P	2	4	1
2	I J K L M E F G	2	0	5	I J E F M N O P	1	4	2
3	I J K L E F G H	3	0	4	I J K E M N O P	0	4	3
4	I J K E M N O P	0	4	3	I J K L E F G H	3	0	4
5	I J E F M N O P	1	4	2	I J K L M E F G	2	0	5
6	I E F G M N O P	2	4	1	I J K L M N E F	1	0	6
7	E F G H M N O P	3	4	0	I J K L M N O E	0	0	7

LEM	Little-endian memory (BigEndianMem = 0)
BEM	BigEndianMem = 1
Type	AccessType (see Table 2-2) sent to memory
Offset	pAddr$_{2...0}$ sent to memory

Exceptions:

TLB refill exception

TLB invalid exception

TLB modification exception

Bus error exception

Address error exception

SWR Store Word Right SWR

31 26	25 21	20 16	15 0
SWR 1 0 1 1 1 0	base	rt	offset
6	5	5	16

Format:

SWR rt, offset (base)

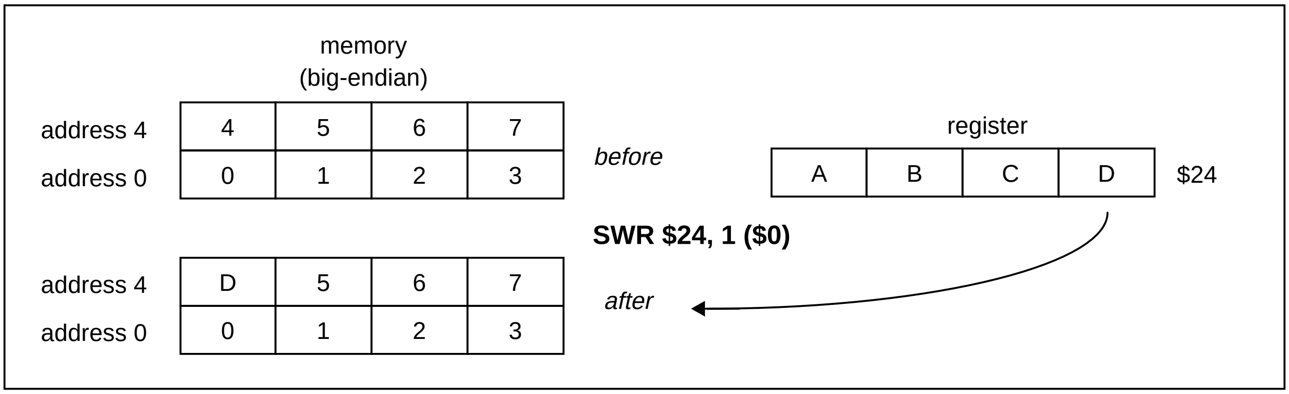
Description:

This instruction can be used with the SWL instruction to store the contents of a register into four consecutive bytes of memory, when the bytes cross a boundary between two words. SWR stores the right portion of the register into the appropriate part of the low-order word; SWL stores the left portion of the register into the appropriate part of the low-order word of memory.

The SWR instruction adds its sign-extended 16-bit *offset* to the contents of general register *base* to form a virtual address which may specify an arbitrary byte. It alters only the word in memory which contains that byte. From one to four bytes will be stored, depending on the starting byte specified.

Conceptually, it starts at the least-significant (rightmost) byte of the register and copies it to the specified byte in memory; then copies bytes from register to memory until it reaches the high-order byte of the word in memory.

No address error exceptions due to alignment are possible.



SWR Store Word Right (Continued) SWR

Operation:

```
T:  vAddr <- ((offset$_{15}$)$^{48}$ || offset$_{15...0}$) + GPR [base]
    (pAddr, uncached) <- AddressTranslation (vAddr, DATA)
    pAddr <- pAddr$_{PSIZE - 1...3}$ || (pAddr$_{2...0}$ xor ReverseEndian$^{3}$)
    if BigEndianMem = 0 then
        pAddr <- pAddr$_{PSIZE - 1...2}$ || 0$^{2}$
    endif
    byte <- vAddr$_{1...0}$ xor BigEndianCPU$^{2}$
    if (vAddr$_{2}$ xor BigEndianCPU) = 0 then
        data <- 0$^{32}$ || GPR [rt]$_{31 - 8* byte...0}$ || 0$^{8* byte}$
    else
        data <- GPR [rt]$_{31 - 8* byte}$ || 0$^{8* byte}$ || 0$^{32}$
    endif
    StoreMemory (uncached, WORD-byte, data, pAddr, vAddr, DATA)
```

SWR Store Word Right (Continued) SWR

Given a doubleword in a register and a doubleword in memory, the operation of SWR is as follows:

SWR

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

vAddr$_{2..0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	destination	type	offset		destination	type	offset	
			LEM	BEM			LEM	BEM
0	I J K L E F G H	3	0	4	H J K L M N O P	0	7	0
1	I J K L F G H P	2	1	4	G H K L M N O P	1	6	0
2	I J K L G H O P	1	2	4	F G H L M N O P	2	5	0
3	I J K L H N O P	0	3	4	E F G H M N O P	3	4	0
4	E F G H M N O P	3	4	0	I J K L H N O P	0	3	4
5	F G H L M N O P	2	5	0	I J K L G H O P	1	2	4
6	G H K L M N O P	1	6	0	I J K L F G H P	2	1	4
7	H J K L M N O P	0	7	0	I J K L E F G H	3	0	4

LEM	Little-endian memory (BigEndianMem = 0)
BEM	BigEndianMem = 1
Type	AccessType (see Table 2-2) sent to memory
Offset	pAddr$_{2...0}$ sent to memory

Exceptions:

TLB refill exception

TLB invalid exception

TLB modification exception

Bus error exception

Address error exception

SYNC Synchronize SYNC

31 26	25 6	5 0
SPECIAL 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	SYNC 0 0 1 1 1 1
6	20	6

Format:

SYNC

Description:

The SYNC instruction is executed as a NOP on the VR4100. This operation maintains compatibility with code compiled for the R4000.

Exceptions:

None

SYSCALL System Call SYSCALL

31 26	25 6	5 0
SPECIAL 0 0 0 0 0 0	Code	SYSCALL 0 0 1 1 0 0
6	20	6

Format:

SYSCALL

Description:

A system call exception occurs, immediately and unconditionally transferring control to the exception handler.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
T:      SystemCallException
```

Exceptions:

System Call exception

TEQ Trap If Equal TEQ

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	code	TEQ 1 1 0 1 0 0
6	5	5	10	6

Format:

TEQ rs, rt

Description:

The contents of general register *rt* are compared to general register *rs*. If the contents of general register *rs* are equal to the contents of general register *rt*, a trap exception occurs.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
T:    if GPR [rs] = GPR [rt] then
          TrapException
      endif
```

Exceptions:

Trap exception

TEQI Trap If Equal Immediate TEQI

31 26	25 21	20 16	15 0
REGIMM 0 0 0 0 0 1	rs	TEQI 0 1 1 0 0	immediate
6	5	5	16

Format:

TEQI rs, immediate

Description:

The 16-bit *immediate* is sign-extended and compared to the contents of general register *rs*. If the contents of general register *rs* are equal to the sign-extended *immediate*, a trap exception occurs.

Operation:

```
T:    if GPR [rs] = $(\text{immediate}_{15})^{48}$ || $\text{immediate}_{15...0}$ then
              TrapException
      endif
```

Exceptions:

Trap exception

TGE Trap If Greater Than Or Equal TGE

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	code	TGE 1 1 0 0 0 0
6	5	5	10	6

Format:

TGE rs, rt

Description:

The contents of general register *rt* are compared to the contents of general register *rs*. Considering both quantities as signed integers, if the contents of general register *rs* are greater than or equal to the contents of general register *rt*, a trap exception occurs.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
T:   if GPR [rs] $\geq$ GPR [rt] then
         TrapException
     endif
```

Exceptions:

Trap exception

TGEI Trap If Greater Than Or Equal Immediate TGEI

31 26	25 21	20 16	15 0
REGIMM 0 0 0 0 0 1	rs	TGEI 0 1 0 0 0	immediate
6	5	5	16

Format:

TGEI rs, immediate

Description:

The 16-bit *immediate* is sign-extended and compared to the contents of general register *rs*. Considering both quantities as signed integers, if the contents of general register *rs* are greater than or equal to the sign-extended *immediate*, a trap exception occurs.

Operation:

```
T:    if GPR [rs] $\geq$ $(\text{immediate}_{15})^{48}$ || $\text{immediate}_{15...0}$ then
              TrapException
      endif
```

Exceptions:

Trap exception

TGEIU Trap If Greater Than Or Equal Immediate Unsigned TGEIU

31 26	25 21	20 16	15 0
REGIMM 0 0 0 0 0 1	rs	TGEIU 0 1 0 0 1	immediate
6	5	5	16

Format:

TGEIU rs, immediate

Description:

The 16-bit *immediate* is sign-extended and compared to the contents of general register *rs*. Considering both quantities as unsigned integers, if the contents of general register *rs* are greater than or equal to the sign-extended *immediate*, a trap exception occurs.

Operation:

```
T:    if $(0 \| \text{GPR}\,[rs]) \geq (0 \| (\text{immediate}_{15})^{48} \| \text{immediate}_{15...0})$ then
              TrapException
      endif
```

Exceptions:

Trap exception

TGEU Trap If Greater Than Or Equal Unsigned TGEU

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	code	TGEU 1 1 0 0 0 1
6	5	5	10	6

Format:

TGEU rs, rt

Description:

The contents of general register *rt* are compared to the contents of general register *rs*. Considering both quantities as unsigned integers, if the contents of general register *rs* are greater than or equal to the contents of general register *rt*, a trap exception occurs.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
T:    if (0 || GPR [rs]) ≥ (0 || GPR [rt]) then
              TrapException
      endif
```

Exceptions:

Trap exception

TLBP Probe TLB For Matching Entry TLBP

31 26	25	24 6	5 0
COP0 0 1 0 0 0 0	CO 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	TLBP 0 0 1 0 0 0
6	1	19	6

Format:

TLBP

Description:

The *Index* register is loaded with the address of the TLB entry whose contents match the contents of the *EntryHi* register. If no TLB entry matches, the high-order bit of the *Index* register is set.

The architecture does not specify the operation of memory references associated with the instruction immediately after a TLBP instruction, nor is the operation specified if more than one TLB entry matches.

Operation:

```
T:    Index <- 1 || $0^{25}$ || $\text{Undefined}^{6}$
      for i in 0...TLBEntries - 1
            if (TLB $[i]_{167...141}$ and not ($0^{15}$ || TLB $[i]_{216...205}$))
            = ($\text{EntryHi}_{39...13}$) and not ($0^{15}$ || TLB $[i]_{216...205}$)) and
            (TLB $[i]_{140}$ *or* (TLB $[i]_{135...128}$ = $\text{EntryHi}_{7...0}$)) then
                        Index <- $0^{26}$ || $i_{5...0}$
            endif
      endfor
```

Exceptions:

Coprocessor unusable exception

TLBR Read Indexed TLB Entry TLBR

31 26	25	24 6	5 0
COP0 0 1 0 0 0 0	CO 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	TLBR 0 0 0 0 0 1
6	1	19	6

Format:

TLBR

Description:

The *G* bit (which controls ASID matching) read from the TLB is written into both of the *EntryLo0* and *EntryLo1* registers.

The *EntryHi* and *EntryLo* registers are loaded with the contents of the TLB entry pointed at by the contents of the TLB *Index* register. The operation is invalid (and the results are unspecified) if the contents of the TLB *Index* register are greater than the number of TLB entries in the processor.

Operation:

```
T:    PageMask <- TLB [$\text{Index}_{5...0}$]$_{255...192}$
      EntryHi <- TLB [$\text{Index}_{5...0}$]$_{191...128}$ and not TLB [$\text{Index}_{5...0}$]$_{255...192}$
      EntryLo1 <- TLB [$\text{Index}_{5...0}$]$_{127...65}$ || TLB [$\text{Index}_{5...0}$]$_{140}$
      EntryLo0 <- TLB [$\text{Index}_{5...0}$]$_{63...1}$ || TLB [$\text{Index}_{5...0}$]$_{140}$
```

Exceptions:

Coprocessor unusable exception

TLBWI Write Indexed TLB Entry TLBWI

31 26	25	24 6	5 0
COP0 0 1 0 0 0 0	CO 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	TLBWI 0 0 0 0 1 0
6	1	19	6

Format:

TLBWI

Description:

The TLB entry pointed at by the contents of the TLB *Index* register is loaded with the contents of the *EntryHi* and *EntryLo* registers.

The *G* bit of the TLB is written with the logical AND of the *G* bits in the *EntryLo0* and *EntryLo1* registers.

The operation is invalid (and the results are unspecified) if the contents of the TLB *Index* register are greater than the number of TLB entries in the processor.

Operation:

```
T:    TLB [$Index_{5...0}$] <-
              PageMask || (EntryHi and not PageMask) || EntryLo1 || EntryLo0
```

Exceptions:

Coprocessor unusable exception

TLBWR Write Random TLB Entry TLBWR

31 26	25	24 6	5 0
COP0 0 1 0 0 0 0	CO 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	TLBWR 0 0 0 1 1 0
6	1	19	6

Format:

TLBWR

Description:

The TLB entry pointed at by the contents of the TLB *Random* register is loaded with the contents of the *EntryHi* and *EntryLo* registers.

The *G* bit of the TLB is written with the logical AND of the *G* bits in the *EntryLo0* and *EntryLo1* registers.

Operation:

```
T:     TLB [$\text{Random}_{5...0}$] <-
               PageMask || (EntryHi and not PageMask) || EntryLo1 || EntryLo0
```

Exceptions:

Coprocessor unusable exception

TLT   Trap If Less Than   TLT

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	code	TLT 1 1 0 0 1 0
6	5	5	10	6

Format:

TLT rs, rt

Description:

The contents of general register *rt* are compared to general register *rs*. Considering both quantities as signed integers, if the contents of general register *rs* are less than the contents of general register *rt*, a trap exception occurs.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
T:    if GPR [rs] < GPR [rt] then
          TrapException
      endif
```

Exceptions:

Trap exception

TLTI Trap If Less Than Immediate TLTI

31 26	25 21	20 16	15 0
REGIMM 0 0 0 0 0 1	rs	TLTI 0 1 0 1 0	immediate
6	5	5	16

Format:

TLTI rs, immediate

Description:

The 16-bit *immediate* is sign-extended and compared to the contents of general register *rs*. Considering both quantities as signed integers, if the contents of general register *rs* are less than the sign-extended *immediate*, a trap exception occurs.

Operation:

```
T:    if GPR [rs] < $(\text{immediate}_{15})^{48}$ || $\text{immediate}_{15...0}$ then
            TrapException
      endif
```

Exceptions:

Trap exception

TLTIU Trap If Less Than Immediate Unsigned TLTIU

31 26	25 21	20 16	15 0
REGIMM 0 0 0 0 0 1	rs	TLTIU 0 1 0 1 1	immediate
6	5	5	16

Format:

TLTIU rs, immediate

Description:

The 16-bit *immediate* is sign-extended and compared to the contents of general register *rs*. Considering both quantities as unsigned integers, if the contents of general register *rs* are less than the sign-extended *immediate*, a trap exception occurs.

Operation:

```
T:    if $(0 \| \mathrm{GPR}\,[rs]) < (0 \| (\mathrm{immediate}_{15})^{48} \| \mathrm{immediate}_{15...0})$ then
          TrapException
      endif
```

Exceptions:

Trap exception

TLTU Trap If Less Than Unsigned TLTU

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	code	TLTU 1 1 0 0 1 1
6	5	5	10	6

Format:

TLTU rs, rt

Description:

The contents of general register *rt* are compared to general register *rs*. Considering both quantities as unsigned integers, if the contents of general register *rs* are less than the contents of general register *rt*, a trap exception occurs.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
T:    if (0 || GPR [rs]) < (0 || GPR [rt]) then
              TrapException
      endif
```

Exceptions:

Trap exception

TNE Trap If Not Equal TNE

31 26	25 21	20 16	15 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	code	TNE 1 1 0 1 1 0
6	5	5	10	6

Format:

TNE rs, rt

Description:

The contents of general register *rt* are compared to general register *rs*. If the contents of general register *rs* are not equal to the contents of general register *rt*, a trap exception occurs.

The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
T:    if GPR [rs] ≠ GPR [rt] then
              TrapException
      endif
```

Exceptions:

Trap exception

TNEI Trap If Not Equal Immediate TNEI

31 26	25 21	20 16	15 0
REGIMM 0 0 0 0 0 1	rs	TNEI 0 1 1 1 0	immediate
6	5	5	16

Format:

TNEI rs, immediate

Description:

The 16-bit *immediate* is sign-extended and compared to the contents of general register *rs*. If the contents of general register *rs* are not equal to the sign-extended *immediate*, a trap exception occurs.

Operation:

```
T:      if GPR [rs] $\neq$ $(\text{immediate}_{15})^{48}$ || $\text{immediate}_{15...0}$ then
                TrapException
        endif
```

Exceptions:

Trap exception

XOR Exclusive Or XOR

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 0 0 0 0 0 0	rs	rt	rd	0 0 0 0 0 0	XOR 1 0 0 1 1 0
6	5	5	5	5	6

Format:

XOR rd, rs, rt

Description:

The contents of general register *rs* are combined with the contents of general register *rt* in a bit-wise logical exclusive OR operation.

The result is placed into general register *rd.*

Operation:

```
T:     GPR [rd] <- GPR [rs] xor GPR [rt]
```

Exceptions:

None

XORI Exclusive OR Immediate XORI

31 26	25 21	20 16	15 0
XORI 0 0 1 1 1 0	rs	rt	immediate
6	5	5	16

Format:

XORI rt, rs, immediate

Description:

The 16-bit *immediate* is zero-extended and combined with the contents of general register *rs* in a bitwise logical exclusive OR operation.

The result is placed into general register *rt.*

Operation:

T: GPR [rt] <- GPR [rs] xor (0^{48} || immediate)

Exceptions:

None

14.7 CPU Instruction Opcode Bit Encoding

The remainder of this Appendix presents the opcode bit encoding for the CPU instruction set (ISA and extensions), as implemented by the VR4100. Figure 14-2 lists the VR4100 Opcode Bit Encoding.

28...26 **Opcode**

31...29	0	1	2	3	4	5	6	7
0	SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	COP0	π	π	*	BEQL	BNEL	BLEZL	BGTZL
3	DADDIε	DADDIUε	LDLε	LDRε	*	*	*	*
4	LB	LH	LWL	LW	LBU	LHU	LWR	LWUε
5	SB	SH	SWL	SW	SDLε	SDRε	SWR	CACHEδ
6	*	π	π	*	*	π	π	LDε
7	*	π	π	*	*	π	π	SDε

2...0 **SPECIAL function**

5...3	0	1	2	3	4	5	6	7
0	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
1	JR	JALR	*	*	SYSCALL	BREAK	*	SYNC
2	MFHI	MTHI	MFLO	MTLO	DSLLVε	*	DSRLVε	DSRAVε
3	MULT	MULTU	DIV	DIVU	DMULTε	DMULTUε	DDIVε	DDIVUε
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	MADD16	DMADD16	SLT	SLTU	DADDε	DADDUε	DSUBε	DSUBUε
6	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	DSLLε	*	DSRLε	DSRAε	DSLL32ε	*	DSRL32ε	DSRA32ε

18...16 **REGIMM rt**

20...19	0	1	2	3	4	5	6	7
0	BLTZ	BGEZ	BLTZL	BGEZL	*	*	*	*
1	TGEI	TGEIU	TLTI	TLTIU	TEQI	*	TNEI	*
2	BLTZAL	BGEZAL	BLTZALL	BGEZALL	*	*	*	*
3	*	*	*	*	*	*	*	*

23...21 **COP0 rs**

25, 24	0	1	2	3	4	5	6	7
0	MF	DMFε	γ	γ	MT	DMTε	γ	γ
1	BC	γ	γ	γ	γ	γ	γ	γ
2	CO MULTU							
3								

Figure 14-2 VR4100 Opcode Bit Encoding

18...16 **COP0 rt**

20...19	0	1	2	3	4	5	6	7
0	BCF	BCT	BCFL	BCTL	γ	γ	γ	γ
1	γ	γ	γ	γ	γ	γ	γ	γ
2	γ	γ	γ	γ	γ	γ	γ	γ
3	γ	γ	γ	γ	γ	γ	γ	γ

2...0 **CP0 Function**

5...3	0	1	2	3	4	5	6	7
0	φ	TLBR	TLBWI	φ	φ	φ	TLBWR	φ
1	TLBP	φ	φ	φ	φ	φ	φ	φ
2	ξ	φ	φ	φ	φ	φ	φ	φ
3	ERET χ	φ	φ	φ	φ	φ	φ	φ
4	φ	STANDBY	SUSPEND	HIBERNATE	φ	φ	φ	φ
5	φ	φ	φ	φ	φ	φ	φ	φ
6	φ	φ	φ	φ	φ	φ	φ	φ
7	φ	φ	φ	φ	φ	φ	φ	φ

Figure 14-2 (cont.) VR4100 Opcode Bit Encoding

Key:

* Operation codes marked with an asterisk cause reserved instruction exceptions in all current implementations and are reserved for future versions of the architecture.

γ Operation codes marked with a gamma cause a reserved instruction exception. They are reserved for future versions of the architecture.

δ Operation codes marked with a delta are valid only for VR4100 processors with CP0 enabled, and cause a reserved instruction exception on other processors.

φ Operation codes marked with a phi are invalid but do not cause reserved instruction exceptions in VR4100 implementations.

ξ Operation codes marked with a xi cause a reserved instruction exception on VR4100 processors.

χ Operation codes marked with a chi are valid on R4x00 and VR4100 processors, not R3000^{TM} or R6000^{TM}.

ε Operation codes marked with epsilon are valid when the processor operating as a 64-bit processor. These instructions will cause a reserved instruction exception if 64-bit operation is not enabled.

π Operation codes marked with a pi are invalid and cause coprocessor unusable exception.

VR4100 Coprocessor 0 Hazards

A

The VR4100 CP0 hazards are equally or less stringent than those of the R4000; Table A-1 lists the VR4100 Coprocessor 0 hazards. Code which complies with these hazards will run without modification on the R4000.

In this table, the number of instructions required between instruction A (which places a value in a CP0 register) and instruction B (any instruction which uses the same register as a source) is computed by the following formula :

(Destination Hazard number of A) -
[(Source Hazard number of B) + 1]

As an example, to compute the number of instructions required between an MTC0 and a subsequent MFC0 instruction, this is:

(5) - (3 + 1) = 1 instructions

Instruction or Event	CP0 Data Used, Stage Used		CP0 Data Written, Stage Available	
MTC0/DMTC0			CPR [0, rd]	5
MFC0/DMFC0	CPR [0, rd]	3		
TLBR	Index, TLB	2	PageMask, EntryHi, EntryLo0, EntryLo1	5
TLBWI TLBWR	Index or Random, PageMask, EntryHi, EntryLo0, EntryLo1	2	TLB	5
TLBP	PageMask, EntryHi	2	Index	6
ERET	EPC or ErrorEPC, TLB	2	Status [EXL, ERL]	4
	Status	2		
Index Load Tag			TagLO, TagHi, PErr	5
Index Store Tag	TagLo, TagHi, PErr	3		
CACHE ops	cache line (see note)	3	cache line (see note)	5
Load/Store	EntryHi [ASID], Status [KSU, EXL, ERL, RE], Config [K0], TLB	3		
	Config [AD, EP]	3		
	WatchHi, WatchLo	3		
Load/Store exception			EPC, Status, Cause, BadVAddr, Context, XContext	5
Instruction fetch exception			EPC, Status	4
			Cause, BadVAddr, Context, XContext	5
Instruction fetch	EntryHi [ASID], Status [KSU, EXL, ERL, RE], Config [K0]	2		
	TLB (mapped address)	2		
Coproc. usable test	Status [CU, KSU, EXL, ERL]	2		
Interrupt signals sampled	Cause [IP], Status [IM, IE, EXL, ERL]	2		
TLB shutdown			Status [TS]	2 (Inst.), 4 (Data)

Table A-1 VR4100 Coprocessor 0 Hazards

Difference between VR4100 and Other VR-Series Processors

B

Note: All tables and figures in this section are not guaranteed operation of the VR4200 and the VR4400.

Cache

Item	VR4200	VR4400	VR4100
Cache Sizes	I-Cache: 16KB D-Cache: 8KB	I-Cache: 16KB D-Cache: 16KB	I-Cache: 2KB D-Cache: 1KB
Cache Line Sizes	I-Cache: 32 Bytes D-Cache: 16 Bytes	Software selectable between 16B and 32B	I-Cache: 16 Bytes D-Cache: 16 Bytes
Cache Organization	Direct mapped	Direct mapped	Direct mapped
Cache Index	I: vAddr [13:0] D: vAddr [12:0]	vAddr [13:0]	I: vAddr [10:0] D: vAddr [9:0]
Cache Tag	pAddr [32:12]	pAddr [35:12]	pAddr [31:10]
Data Cache Write Policy	Write-allocate and write-back	Write-allocate and write-back	Write-allocate and write-back
Data order for block reads	Sub-block ordering	Same	Sub-block ordering
Data order for block writes	Sequential ordering	Same	Sub-block ordering
Instruction Cache miss restart	Restart after all data received and written to cache	Same	Restart at the same time last data received and written to cache
Data Cache miss restart	Early restart on first doubleword.	Restart after all data received and written to cache	Restart at the same time last data received and written to cache
Instruction cache parity	1 parity bit per 1 byte	1 parity bit per 1 byte	1 parity bit per 1 word
Data cache parity	1 parity bit per 1 byte	1 parity bit per 1 byte	1 parity bit per 1 byte

TLB

Item	VR4200	VR4400	VR4100
Instruction virtual address translation	2-entry ITLB	2-entry ITLB	4-entry ITLB
Data virtual address translation	JTLB	JTLB	4-entry DTLB
JTLB	32 entries of even/odd page pairs, fully associative	48 entries of even/odd page pairs, fully associative	32 entries of even/odd page pairs, fully associative
Page sizes	4KB, 16KB, 64KB, 256KB, 1MB, 4MB, 16MB	Same	1KB, 4KB, 16KB, 64KB, 256KB
Multiple entry match in JTLB	Sets TS in Status and disables TLB until Reset to prevent damage	Same	Same

Address Sizes	VSIZE = 40
	PSIZE = 33
		VSIZE = 40
		PSIZE = 36
			VSIZE = 40
			PSIZE = 32

Pipeline

Item	VR4200	VR4400	VR4100
CPU/FPU	Logically separate; datapath shared	Logically and physically separate	Unimplemented FPU
ALU Latency	1 cycle	1 cycle	1 cycle
Load Latency	2 cycles	3 cycles	2 cycles
Branch Latency	2 cycles	4 cycles	2 cycles
Store Buffer	2 doublewords	2 doublewords	2 doublewords
Uncached Store Buffer	2 doublewords (1 address) doubles as write buffer	1 doubleword	1 doubleword (1 address) doubles as write buffer
Integer Multiply	Done in adder/shifter, 12 cycles to issue	Integer multiply hardware, 1 cycle to issue	Integer multiply-divide hardware, 1 cycle to issue
Integer Divide	Done in common adder/shifter, 36 cycles to issue	Done in integer datapath adder, 69 cycles to issue	Integer multiply-divide hardware, 1 cycle to issue
Integer Multiply	HI and LO available at the same time	HI and LO available at the same time	HI and LO available at the same time
Integer Multiply and accumulator	None	None	0 latency
Integer Divide	HI and LO available at the same time	Same	Same
HI and LO Hazards	Yes, at least a one-cycle hazard. Assume two-cycle hazard (same as VR4000PC™) until further notice.	Yes, HI and LO written early in pipeline	Yes, at least a one-cycle hazard. Assume two-cycle hazard (same as VR4000PC) until further notice.
MFHI/MFLO Latency	1 cycle	1 cycle	2 cycles
SLLV, SRLV, SRAC	1 cycle	2 cycles	1 cycle
DSLL, DSRL, DSRA, DSLL32, DSRL32, DSRA32, DSLLV, DSRLCV, DSRAC	1 cycle	2 cycles	1 cycle

System Interface

Item	VR4200	VR4400	VR4100
I/O Level	LVCMOS	LVCMOS (3 V) TLL-compatible	LVTTL (LVCMOS)
Package	179-pin PGA/208-pin PQFP	179-pin PGA/447-pin PGA	100-pin TQFP
JTAG	Yes	Yes	No
Block transfer sizes	32 bytes (Instruction) 16 bytes (Data)	16 bytes or 32 bytes	16 bytes (Instruction) 16 bytes (Data)
Master Clock Input	40 MHz	75 MHz/100 MHz	DC - 8.25 MHz
SClock Divisor	2, 3, or 4	2, 3, 4, 6, or 8	1 or 2
Non-block write	Max throughput of 1 per 3 sclock cycles	Max throughput of 1 per 4 sclock cycles	Select the Max throughput of 1 per 2 or 1 per 4 sclock cycles
Serial Configuration	All configurations are specific by special pin	As described in R4000/4400 User's Guide	Set default value. Some configurations are specified by special pin. Ohters are set default value (programmable on software)
Upper address bits on reads and write	Bits 63...36 are Zero	Bits 63...33 are Zero	Bits 63...32 are not defined
Uncached and write-through stores	Uncached stores buffered in 1-entry write buffer	Uncached stores buffered in 1-entry dedicated uncached store buffer	Uncached stores buffered in 1-entry write buffer
SysAD bus	64-bit	64-bit	32-bit
SysCmd bus	9-bit	9-bit	5-bit
SysADC bus	8-bit parity only	8-bit parity only	4-bit parity only
SysADC for nondata cycles	Zero	Parity	Parity
SysCmdP for nondata cycles	Zero	Parity	Parity
Parity Error during writeback	Use Cache Error Exception	Use Cache Error Exception	Use Cache Error Exception
Error bit in data identifier of read responses	Bus error if error bit set for any doubleword	Same	Any word
Parity error on read data	Use Cache error Exception	Same	Same
System Interface Arbitration	Handshake by $\overline{\text{ExtRqst}}$, $\overline{\text{Release}}$	Handshake by $\overline{\text{ExtRqst}}$, $\overline{\text{Release}}$	Handshake by $\overline{\text{EReq}}$, $\overline{\text{PReq}}$, $\overline{\text{PMaster}}$
Block Write	0 cycle between address and data	1-2 null cycle between address and data	0 cycle between address and data
Release after Read Request	Variable latency	0 latency	0 latency
SysAD value for x cycles of writeback data pattern	Data bus maintains last D cycle value	Data bus undefined	Data bus maintains last D cycle value
SysAD bus used after last D cycle	Unused for trailing x cycles	Undefined	Unused for trailing x cycles

Item	VR4200	VR4400	VR4100
Interrupt	$\overline{\text{Int}}$[4:0], $\overline{\text{NMI}}$	$\overline{\text{Int}}$[5:0], $\overline{\text{NMI}}$	$\overline{\text{Int}}$[4:0], $\overline{\text{NMI}}$
Output slew rate	Simple CMOS output buffers	Dynamic feedback control	Simple CMOS output buffers
IOOut output	No external pins are assigned	Output slew rate control feedback loop output	No external pins are assigned
IOIn input	No external pins are assigned	Output slew rate control input	No external pins are assigned

Power Management

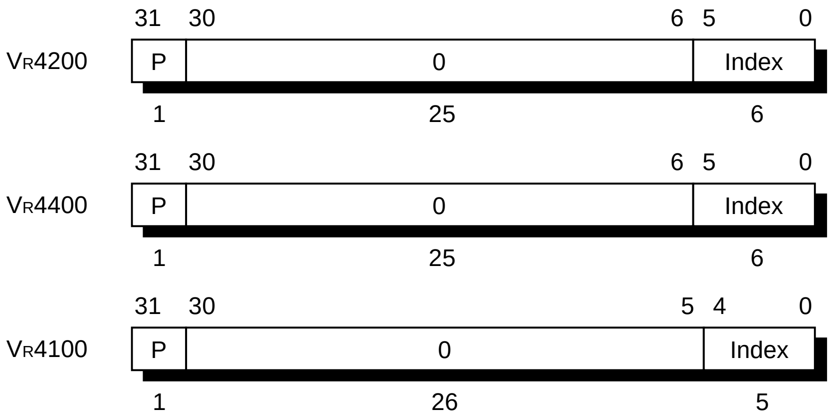
Item	VR4200	VR4400	VR4100
Low-power mode	1) Reduced-power mode (1/4 speed) 2) Instant-off mode	No	1) Suspend-mode (1/10 Power) 2) Hibernate mode (0 mW) 3) Standby mode

Exception Processing

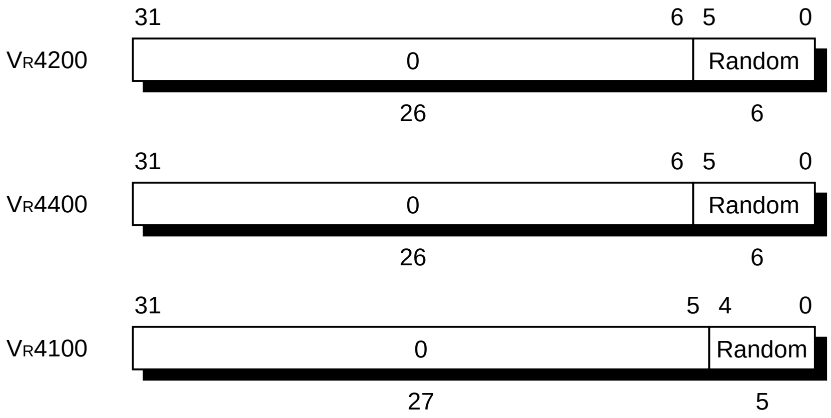
★

Item	VR4200	VR4400	VR4100
CP1, CP2 instruction	No exception	No exception	Coprocessor unusable exception
LL/LLD/SC/SCD instruction	No exception	No exception	Reserved instruction exception

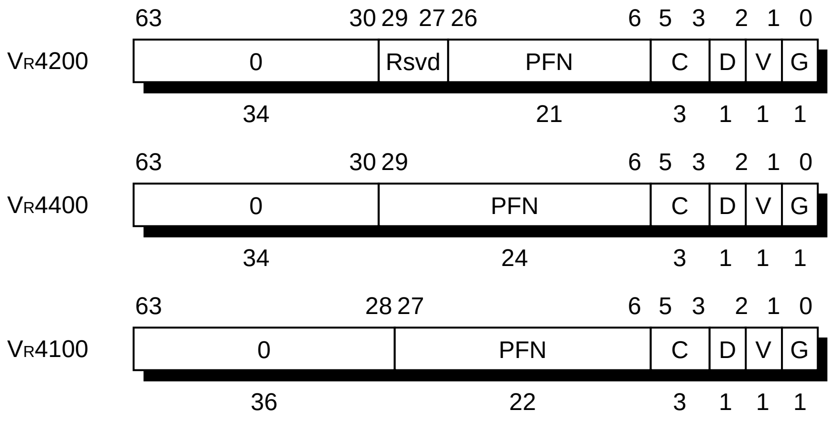
Index register (0)



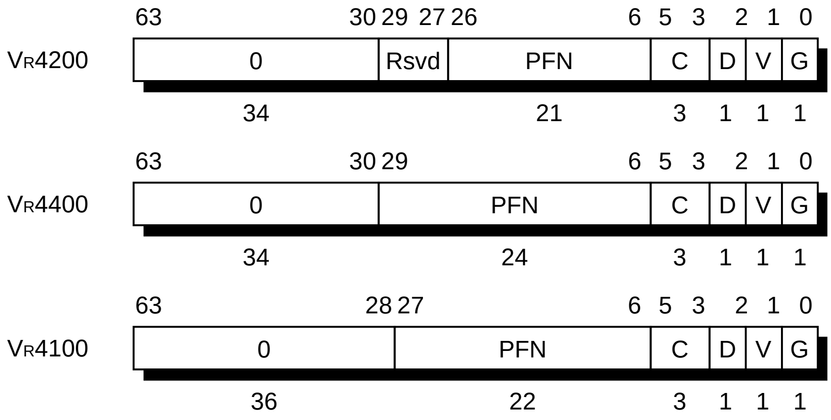
Random (1)



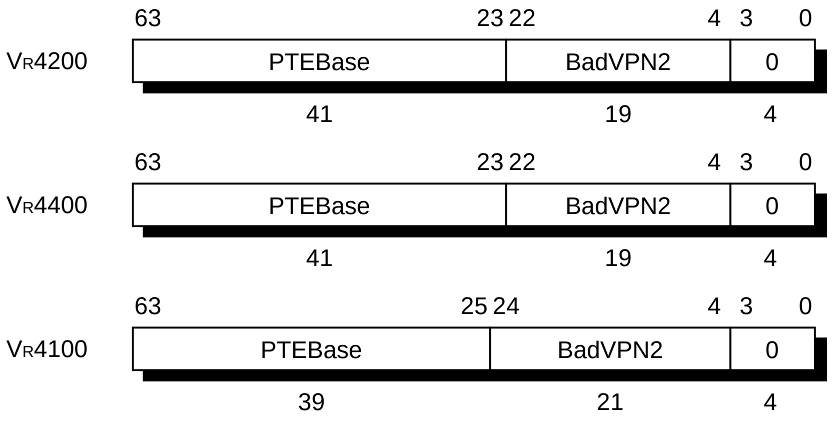
EntryLo0 (2)



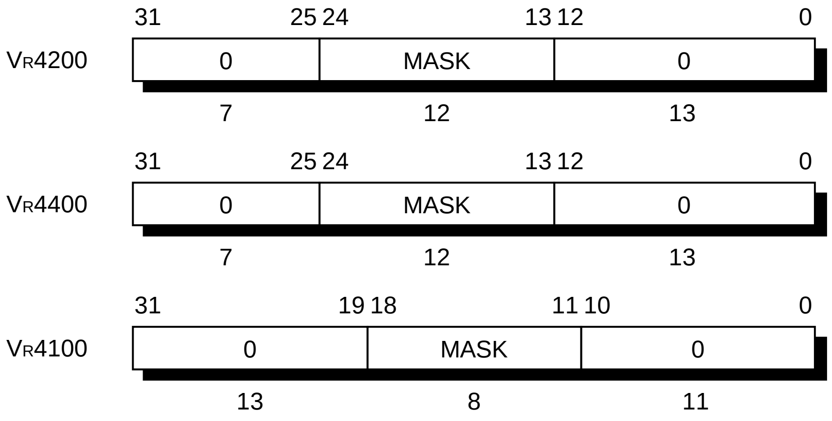
EntryLo1 (3)



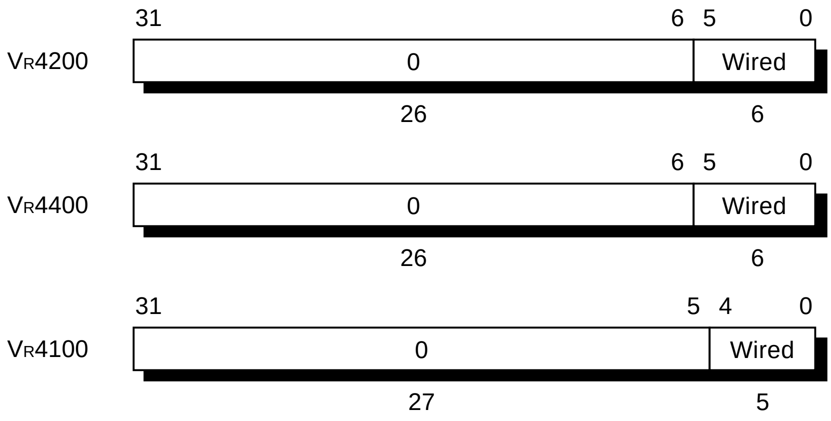
Context (4)



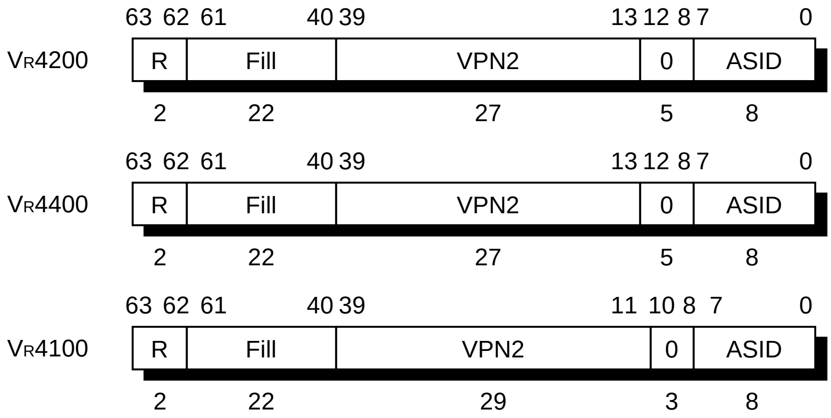
PageMask (5)



Wired (6)



EntryHi (10)



Status (12)

	31 28	27	26	25	24 16	15 8	7	6	5	4 3	2	1	0
VR4200	CPU (CU3 - CU0)	RP	FR	RE	DS	IM	KX	SX	UX	KSU	ERL	EXL	IE
	4	1	1	1	9	8	1	1	1	2	1	1	1

	31 28	27	26	25	24 16	15 8	7	6	5	4 3	2	1	0
VR4400	CPU (CU3 - CU0)	0	FR	RE	DS	IM	KX	SX	UX	KSU	ERL	EXL	IE
	4	1	1	1	9	8	1	1	1	2	1	1	1

	31 29	28	26	25	24 16	15 8	7	6	5	4 3	2	1	0
VR4100	0	CU0	0	RE	DS	IM	KX	SX	UX	KSU	ERL	EXL	IE
	3	1	2	1	9	8	1	1	1	2	1	1	1

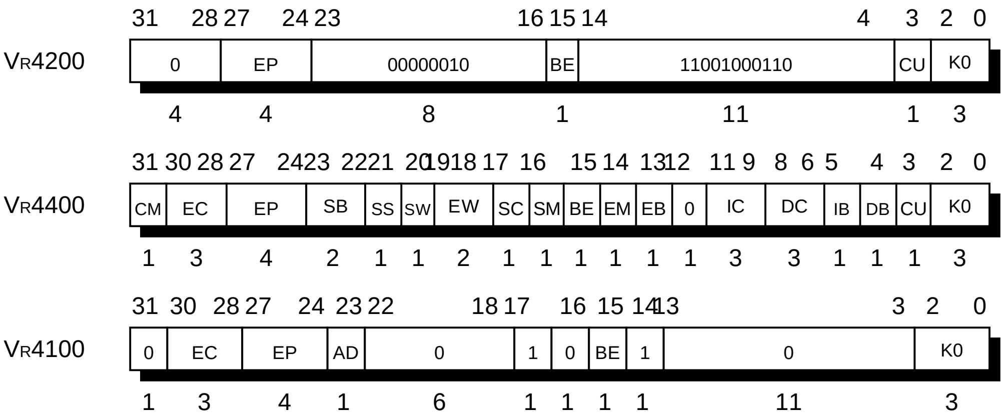
Status DS Field

	24	23	22	21	20	19	18	17	16
VR4200	ITS	0	BEV	TS	SR	0	CH	CE	DE
	1	1	1	1	1	1	1	1	1

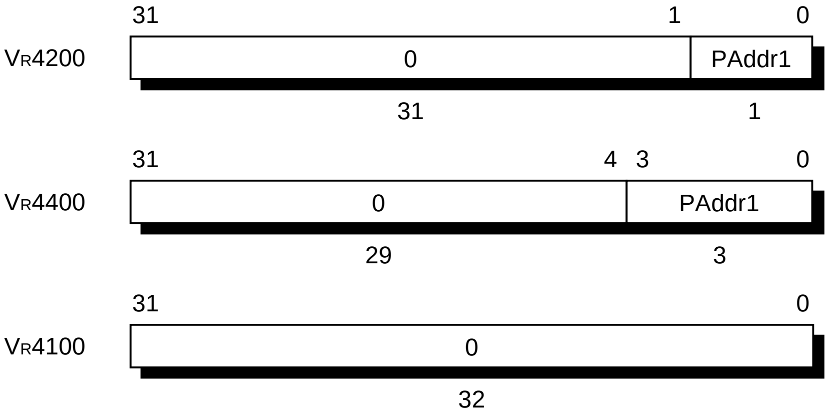
	24	23	22	21	20	19	18	17	16
VR4400	0		BEV	TS	SR	0	CH	CE	DE
	2		1	1	1	1	1	1	1

	24	23	22	21	20	19	18	17	16
VR4100	0		BEV	TS	SR	0	CH	CE	DE
	2		1	1	1	1	1	1	1

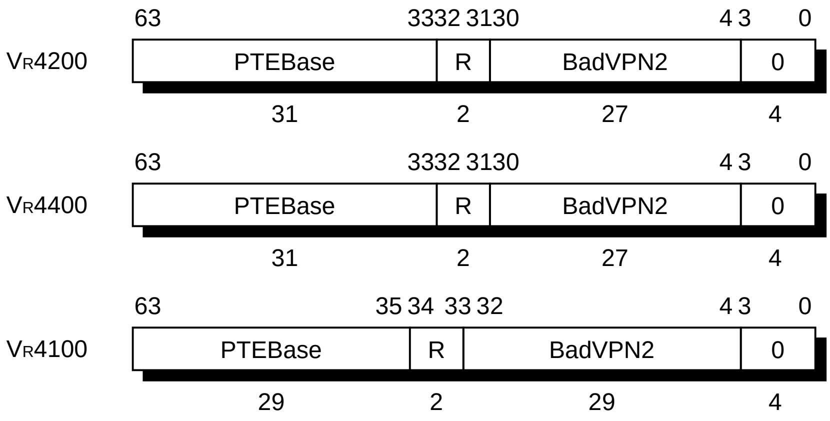
Config (16)



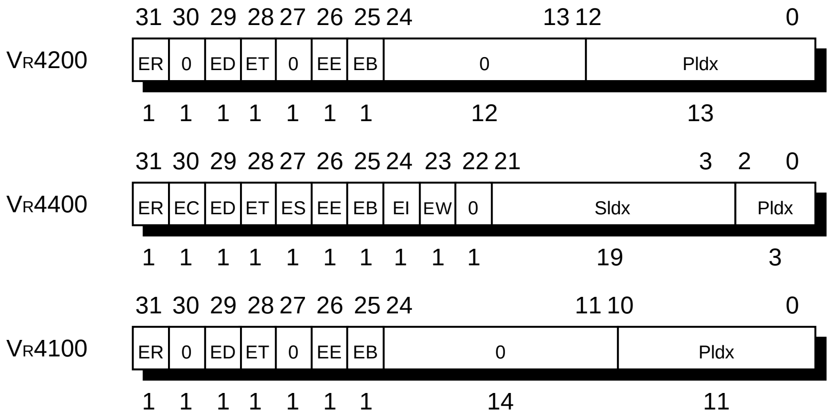
WatchHi (19)



XContext (20)



CacheErr (27)



TagLo (28)

VR4200

31 29	28 8	7	6 1	0
0	PTagLo	PState	0	P
3	21	2	5	1

VR4400

31 8	7	6 1	0
PTagLo	PState	0	P
24	2	5	1

VR4100

D-Cache

31 10	9	8	7	6 2	1	0
PTagLo	V	D	W	0	W’	P
22	1	1		6		1

I-Cache

31 10	9	8 1	0
PTagLo	V	0	P
22	1	8	1

Cycle Timing for Multiply and Divide Instructions

VR4200

Instruction	Pcycles Required
MULT	12
MULTU	13
DIV	39
DIVU	39
DMULT	23
DMULTU	24
DDIV	71
DDIVU	71
MADD16	-
DMADD16	-

VR4400

Instruction	Pcycles Required
MULT	10
MULTU	10
DIV	69
DIVU	69
DMULT	20
DMULTU	20
DDIV	133
DDIVU	133
MADD16	-
DMADD16	-

VR4100

Instruction	Pcycles Required
MULT	1
MULTU	1
DIV	35
DIVU	35
DMULT	4
DMULTU	4
DDIV	67
DDIVU	67
MADD16	1
DMADD16	1

[MEMO]