NEC

User's Manual

V_R4300^{TM}, V_R4305^{TM}, V_R4310^{TM}

64-Bit Microprocessor

μPD30200
μPD30210

Document No. U10504EJ7V0UMJ1 (7th edition)
Date Published August 2000 N CP(K)

Printed in Japan

[MEMO]

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 V_{DD} 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.

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 current as of October, 1999. The information is subject to change without notice. For actual design-in, refer to the latest publications of NEC's data sheets or data books, etc., for the most up-to-date specifications of NEC semiconductor products. Not all products and/or types are available in every country. Please check with an NEC sales representative for availability and additional information.**
- No part of this document may be copied or reproduced in any form or by any means without prior written consent of NEC. NEC assumes no responsibility for any errors that may appear in this document.
- NEC does not assume any liability for infringement of patents, copyrights or other intellectual property rights of third parties by or arising from the use of NEC semiconductor products listed in this document or any other liability arising from the use of such products. No license, express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC or others.
- Descriptions of circuits, software and other related information in this document are provided for illustrative purposes in semiconductor product operation and application examples. The incorporation of these circuits, software and information in the design of customer's equipment shall be done under the full responsibility of customer. NEC assumes no responsibility for any losses incurred by customers or third parties arising from the use of these circuits, software and information.
- While NEC endeavours to enhance the quality, reliability and safety of NEC semiconductor products, customers agree and acknowledge that the possibility of defects thereof cannot be eliminated entirely. To minimize risks of damage to property or injury (including death) to persons arising from defects in NEC semiconductor products, customers must incorporate sufficient safety measures in their design, such as redundancy, fire-containment, and anti-failure features.
- NEC semiconductor products are classified into the following three quality grades:
  "Standard", "Special" and "Specific". The "Specific" quality grade applies only to semiconductor products developed based on a customer-designated "quality assurance program" for a specific application. The recommended applications of a semiconductor product depend on its quality grade, as indicated below. Customers must check the quality grade of each semiconductor product 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": Aircraft, aerospace equipment, submersible repeaters, nuclear reactor control systems, life support systems and medical equipment for life support, etc.

  The quality grade of NEC semiconductor products is "Standard" unless otherwise expressly specified in NEC's data sheets or data books, etc. If customers wish to use NEC semiconductor products in applications not intended by NEC, they must contact an NEC sales representative in advance to determine NEC's willingness to support a given application.

  (Note)

  (1) "NEC" as used in this statement means NEC Corporation and also includes its majority-owned subsidiaries.

  (2) "NEC semiconductor products" means any semiconductor product developed or manufactured by or for NEC (as defined above).

M8E 00.4

Regional Information

Some information contained in this document may vary from country to country. Before using any NEC product in your application, please contact the NEC office in your country to obtain a list of authorized representatives and distributors. They will verify:

- Device availability
- Ordering information
- Product release schedule
- Availability of related technical literature
- Development environment specifications (for example, specifications for third-party tools and components, host computers, power plugs, AC supply voltages, and so forth)
- Network requirements

In addition, trademarks, registered trademarks, export restrictions, and other legal issues may also vary from country to country.

NEC Electronics Inc. (U.S.)
Santa Clara, California
Tel: 408-588-6000
800-366-9782
Fax: 408-588-6130
800-729-9288

NEC Electronics (Germany) GmbH
Duesseldorf, Germany
Tel: 0211-65 03 02
Fax: 0211-65 03 490

NEC Electronics (UK) Ltd.
Milton Keynes, UK
Tel: 01908-691-133
Fax: 01908-670-290

NEC Electronics Italiana s.r.l.
Milano, Italy
Tel: 02-66 75 41
Fax: 02-66 75 42 99

NEC Electronics (Germany) GmbH
Benelux Office
Eindhoven, The Netherlands
Tel: 040-2445845
Fax: 040-2444580

NEC Electronics (France) S.A.
Velizy-Villacoublay, France
Tel: 01-30-67 58 00
Fax: 01-30-67 58 99

NEC Electronics (France) S.A.
Madrid Office
Madrid, Spain
Tel: 91-504-2787
Fax: 91-504-2860

NEC Electronics (Germany) GmbH
Scandinavia Office
Taeby, Sweden
Tel: 08-63 80 820
Fax: 08-63 80 388

NEC Electronics Hong Kong Ltd.
Hong Kong
Tel: 2886-9318
Fax: 2886-9022/9044

NEC Electronics Hong Kong Ltd.
Seoul Branch
Seoul, Korea
Tel: 02-528-0303
Fax: 02-528-4411

NEC Electronics Singapore Pte. Ltd.
United Square, Singapore
Tel: 65-253-8311
Fax: 65-250-3583

NEC Electronics Taiwan Ltd.
Taipei, Taiwan
Tel: 02-2719-2377
Fax: 02-2719-5951

NEC do Brasil S.A.
Electron Devices Division
Guarulhos-SP Brasil
Tel: 55-11-6462-6810
Fax: 55-11-6462-6829

J00.7

Major Revisions in This Edition

Page	Description
p.33	**1.1 Characteristics** Correction of description
p.35	**1.4.1 Internal Block Configuration** Correction of description
p.166	**6.3.5 Status Register (12)** Correction of description
p.198	**6.4.17 Watch Exception** Correction and addition of description
p.244	**8.2.7 Unimplemented Operation Exception (E)** Addition of description
p.254	**9.3.1 Power Modes** Correction of description
pp.259, 260	**10.2 Basic System Clocks** Correction of description
p.264	**10.4 Low Power Mode Operation** Correction of description
p.360	**15.1 Features** Correction of description
p.360	**15.1.2 Low Power Mode** Correction of description
p.568 p.570 p.574 p.576 p.578 p.580 p.587 p.589 p.600 p.602 p.610 p.612	**17.5 FPU Instructions** Addition of description to the following instructions CEIL.L.fmt CEIL.W.fmt CVT.D.fmt CVT.L.fmt CVT.S.fmt CVT.W.fmt FLOOR.L.fmt FLOOR.W.fmt ROUND.L.fmt ROUND.W.fmt TRUNC.L.fmt TRUNC.W.fmt
p.628	**Table A-1 Differences Between the V_R4300, V_R4305, and V_R4310** Correction of description
p.630	**B.1.3 Status Register** Correction of description
p.632	**Table B-1 Differences in Software** Correction of description
p.634	**B.2.2 System Interface** Correction of description
p.635	**Table B-2 Differences in System Design** Correction of description
p.639	**Table B-3 Other Differences** Correction of description
p.644	**C.2.2 Clock** Correction of description
pp.647, 648	**Appendix D Restrictions of V_R4300** Addition

The mark ★ shows major revised points.

PREFACE

Readers

This manual targets users who intends to understand the functions of the V_R4300, V_R4305 (μPD30200, V_R4310 (μPD30210) and to design application systems using this microprocessor.

Purpose

This manual introduces the architecture functions of the V_R4300, V_R4305, and V_R4310 to users, following the organization described below.

Organization

This manual consists of the following contents:

- Introduction
- Pipeline operation
- Memory management system and cache
- Exception processing
- Floating-point operation
- Hardware
- Instruction set details

How to read this manual

It is assumed that the readers of this manual has a general knowledge of electric engineering, logic circuits, and microcomputers.

Unless otherwise specified, V_R4300 is described as a representative product in this manual. When using this manual as that for V_R4305 or V_R4310, read as follows.

V_R4300 → V_R4305
V_R4300 → V_R4310

The V_R4400™ in this manual represents the V_R4000™.
The V_R4000 series in this manual represents the V_R4100™, V_R4200™, V_R4300, V_R4305, V_R4310, and V_R4400.

To learn about detailed function of a specific instruction,

→ Refer to **Chapter 3 CPU Instruction Set Summary**, **Chapter 7 Floating-Point Operations**, and **Chapter 17 FPU Instruction Set Details**.

To learn about the overall functions of the V_R4300,
→ Read this manual in sequential order.

To learn about electrical specifications of the V_R4300,
→ Refer to the **data sheet** which is separately available.

Conventions

Data significance:	Higher digits on the left and lower digits on the right
Active low representation:	$\overline{\times\times\times}$ (overscore over pin or signal name)
*:	Footnote for item marked with * in the text
Caution:	Information requiring particular attention
Remark:	Supplementary information
Numerical representation:	binary or decimal ... ×××× hexadecimal .....0××××

Prefixes indicating power of 2 (address space, memory capacity):

K (kilo)	$2^{10} = 1024$
M (mega)	$2^{20} = 1024^2$
G (giga)	$2^{30} = 1024^3$
T (tera)	$2^{40} = 1024^4$
P (peta)	$2^{50} = 1024^5$
E (exa)	$2^{60} = 1024^6$

Related documents

See also the following documents.
The related documents indicated in this publication may include preliminary versions. However, preliminary versions are not marked as such.

Document Name	Document Number
V_R4300, V_R4305, V_R4310 User's Manual	This manual
μPD30200, 30210 Data Sheet	U10116E
V_R Series Application Note - Programming Guide	U10710E
V_R4000 Series Application Note - Simulation Guide	U11788J (Japanese only)

CONTENTS

LIST OF FIGURES (1/6)

LIST OF FIGURES (2/6)

LIST OF FIGURES (3/6)

LIST OF FIGURES (4/6)

LIST OF FIGURES (5/6)

LIST OF FIGURES (6/6)

LIST OF TABLES (1/4)

LIST OF TABLES (2/4)

LIST OF TABLES (3/4)

LIST OF TABLES (4/4)

[MEMO]

General

1

This chapter outlines the RISC 64-bit microprocessor V_R4300, V_R4305 (μPD30200), and V_R4310 (μPD30210).

1.1 Characteristics

The V_R4300, V_R4305, and V_R4310 are members of the NEC V_R Series™ RISC (Reduced Instruction Set Computer) microprocessors and is a high-performance 64-bit microprocessor employing the RISC architecture developed by MIPS™.

Its instructions are upward-compatible with the instructions of the V_R3000™ Series and are completely compatible with those of the V_R4400 and V_R4200. Therefore, existing applications can be used as is with the V_R4300, V_R4305, and V_R4310.

The V_R4300, V_R4305, and V_R4310 have the following features:

- Internal operating frequency:
  80 MHz max. (μPD30200-80),
  100 MHz max. (μPD30200-100),
  133 MHz max. (μPD30200-133, 30210-133),
  167 MHz max. (μPD30210-167)
- 64-bit architecture supporting 64-bit data processing
- Optimized, 5-stage pipeline processing
- High-speed translation lookaside buffer (TLB) supporting virtual addresses (of 32 double entries)
- Address space
  Physical: 32 bits
  Virtual: 40 bits (64-bit mode)
  31 bits (32-bit mode)
- Supports single-precision and double-precision floating-point operations
- On-chip cache memories
  Instruction: 16 KB
  Data: 8 KB
- Employs write back cache system → store operation via system bus decreased
- 32-bit external bus interface facilitating system development
- Multiplies external operating frequency (input clock and bus interface) to create internal operating frequency.
  Multiple is selected on power application
  (μPD30200-80: ×1, ×2, or ×3)
  (μPD30200-100: ×1.5, ×2, or ×3)
  (μPD30200-133: ×2, ×3, or ×4)
  (μPD30210-133: ×2, ×2.5, ×3, or ×4)
  (μPD30210-167: ×2, ×2.5, ×3, ×4, ×5, or ×6)

- Write buffer
- ★ Low power mode (μPD30200-80, 30200-100 only)
  Reduces internal and system bus clocks to 1/4 of normal level. Also reduces power consumption
- Software-compatible with V_R4400 and V_R4200 and upward-compatible with V_R3000 Series
- Supply voltage: 3.3 V ± 0.3 V (μPD30200-80, 30200-100), 3.0 to 3.5 V (μPD30200-133, 30210-×××)

1.2 Ordering Information

Part Number	Package	Maximum Operating Frequency (MHz)
μPD30200GD-80-LBB	120-pin plastic QFP (28 × 28 mm)	80
μPD30200GD-100-MBB	120-pin plastic QFP (28 × 28 mm)	100
μPD30200GD-133-MBB	120-pin plastic QFP (28 × 28 mm)	133
μPD30210GD-133-MBB	120-pin plastic QFP (28 × 28 mm)	133
μPD30210GD-167-MBB	120-pin plastic QFP (28 × 28 mm)	167

1.3 64-Bit Architecture

The V_R4300 is a 64-bit high-performance microprocessor. It can also execute 32-bit applications even when it operates as a 64-bit microprocessor.

1.4 V_R4300 Processor

Figure 1-1 shows the internal block diagram of the V_R4300.
The V_R4300 is equipped with a full-associative high-speed translation lookaside buffer (TLB) that has 32 entries with two pages corresponding to each entry; data cache and instruction cache; and FPU, in addition to a high-performance integer operation unit.

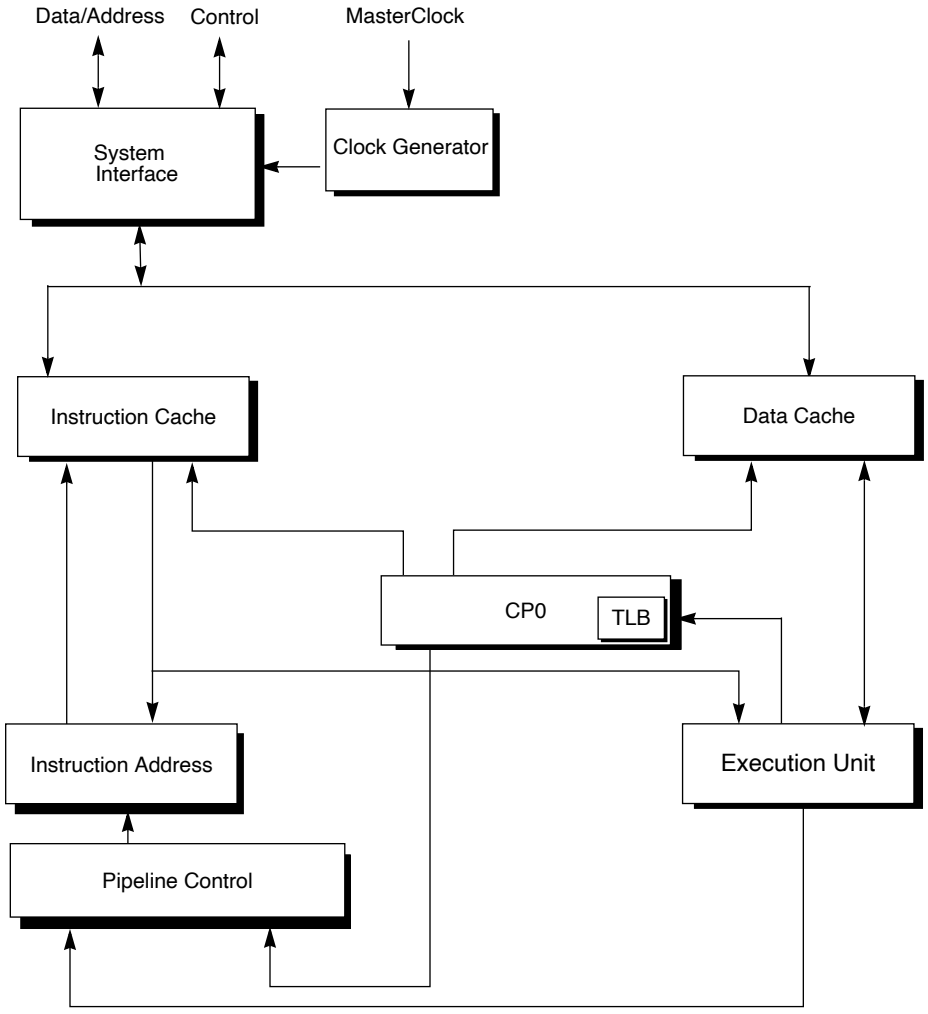


Figure 1-1 Internal Block Diagram

1.4.1 Internal Block Configuration

System Interface allows the processor to access external resources such as memories. It contains a 32-bit multiplexed address/data bus, with per-byte parity, clock signals, interrupt request signals, and various control signals. It is not compatible with the System interface bus used on the V_R4400 and V_R4200.

Clock Generator generates a pipeline clock (PClock) based on an externally input clock (MasterClock). The frequency of the **PClock** can be selected by setting the frequency ratio between the **MasterClock** and the **PClock**. This ratio is set using the **DivMode** pins on power application. (For setting of the **DivMode** pins, refer to **Table 2-2 Clock/Control Interface Signals**.) Table 1-1 indicates the selectable frequency ratio. System interface clock (**SClock**) usually has the same frequency as the **MasterClock**.

Table 1-1 Frequency Ratio Between PClock and MasterClock

Product Name	DivMode Pin	Selectable Frequency Ratio (MasterClock : PClock)
V_R4300	DivMode (1 : 0)	1 : 1.5*1, 1 : 2, 1 : 3, 1 : 4*2
V_R4305	DivMode (1 : 0)	1 : 1, 1 : 2, 1 : 3
V_R4310	DivMode (2 : 0)	1 : 2, 1 : 2.5*3, 1 : 3, 1 : 4, 1 : 5, 1 : 6

***1.** Selectable with the 100 MHz model only (With the 133 MHz model, this setting is reserved.)
2. Selectable with the 133 MHz model only (With the 100 MHz model, this setting is reserved.)
3. Selectable with the 167 MHz model only (With the 133 MHz model, this setting is reserved.)

If the RP bit of the *Status* register is set to 1 during operation, the frequencies of the PClock and SClock can be reduced to 1/4 of the normal frequency*. Because the PLL (Phase-Locked Loop) technique is employed, the skew (phase difference) between the external clock and internal operation clock can be minimized.

★ * 100 MHz model of the V_R4300 and the V_R4305 only

Instruction Cache is direct-mapped, virtually-indexed, and physically-tagged. The capacity is 16 KB.

Execution Unit has the hardware resources to execute integer and floating-point instructions. It has a 64-bit register file, 64-bit integer/mantissa datapath, and 12-bit exponent datapath. It is provided with a dedicated multiplexer in order to process multiply instruction at a high speed.

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.

Data Cache is a direct-mapped, virtually-indexed and physically-tagged write-back cache. The capacity is 8 KB.

Instruction Address calculates the effective address of the next instruction to be fetched. It contains the incrementer for the Program Counter (PC), the target address adder, and the conditional branch address selector.

Pipeline Control ensures the instruction pipeline operates properly (should one of the following conditions occur: pipeline stall or exception).

1.4.2 CPU Registers

The processor provides the following registers:

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

In addition, the processor provides the following special registers:

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

Two of the *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 JAL and JALR instructions. It can be used by other instructions. Make sure that other data used in calculations does not overlap with the register used by the JAL/JALR instruction.

Furthermore, the processor contains registers in the system control processor (CP0) which perform the exception processing and address management.

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

Figure 1-2 shows the *CPU* registers.

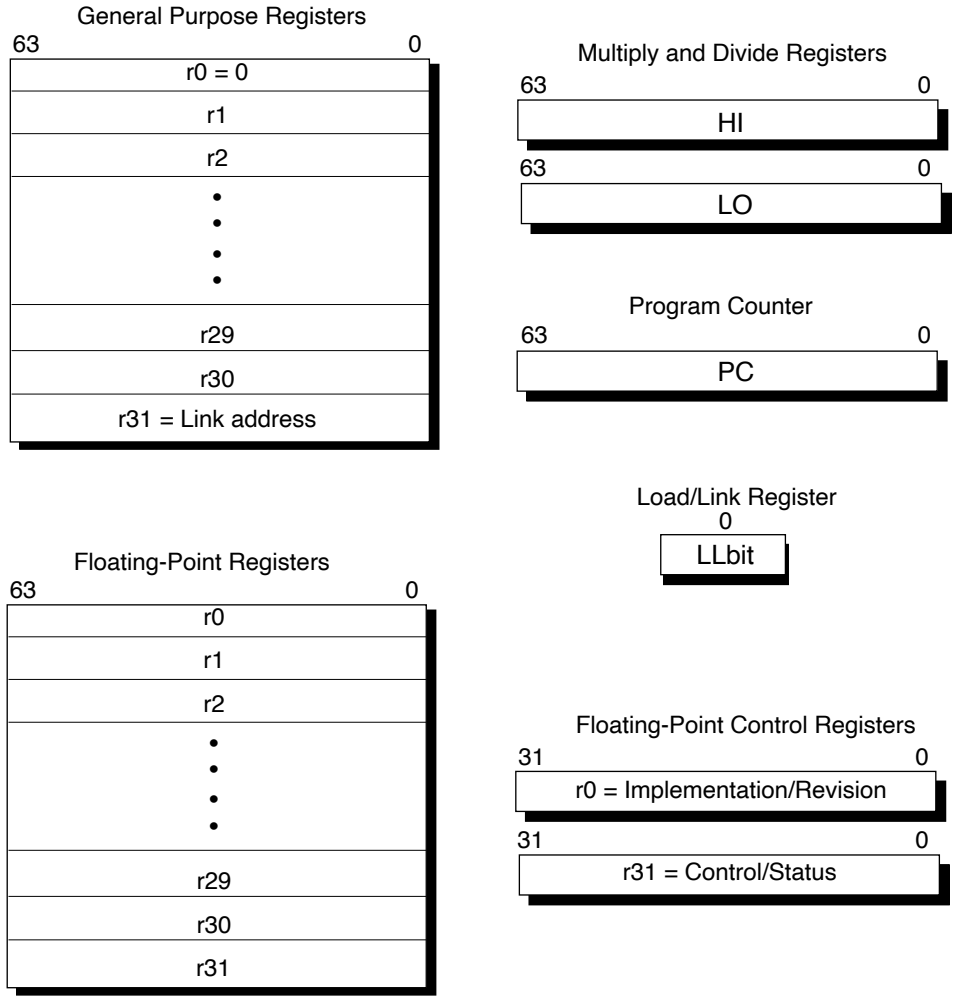


Figure 1-2 CPU Registers

The V_R4300 processor has no *Program Status Word* (PSW) register as such; this is covered by the *Status* and *Cause* registers incorporated within the System Control Coprocessor (CP0). For CP0 registers, refer to **1.4.5 System Control Coprocessor (CP0)**.

1.4.3 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)

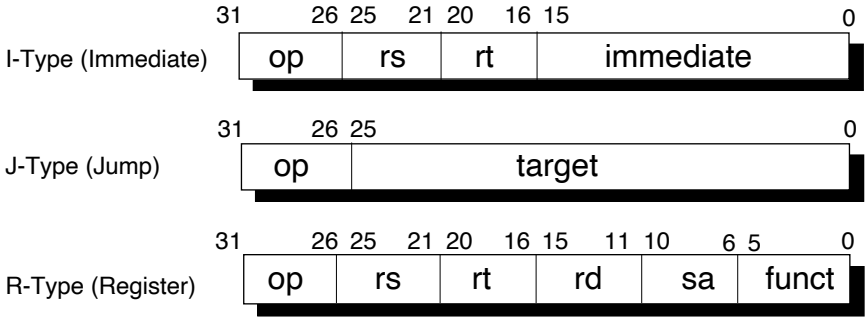


Figure 1-3 CPU Instruction Formats

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

- **Load and Store** instructions move data between memory and general purpose 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 signed immediate value) formats.
- **Jump and Branch** instructions change the control flow of a program. Jumps are always made to an 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). Branch instructions are performed to the 16-bit offset address relative to the program counter (I-type). Jump And Link instructions save their return address in register 31.

- **Coprocessor** instructions (CPz) perform operations in the coprocessors. Coprocessor load and store instructions are I-type. As opposed to CP0 instructions, CPz instructions are not specific to any coprocessor. (Refer to **Chapter 7 Floating-Point Operations**.)
- **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 call exception and breakpoint exception 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.

For each instruction, refer to **Chapter 3 CPU Instruction Set Summary** and **Chapter 16 CPU Instruction Set Details**.

1.4.4 Data Formats and Addressing

The V_R4300 processor uses four data formats: a 64-bit doubleword, a 32-bit word, a 16-bit halfword, and an 8-bit byte. Byte ordering within all of the larger data formats—halfword, word, doubleword—can be configured in either big-endian or little-endian. When the V_R4300 processor is configured as a big-endian system, byte 0 is the most-significant (leftmost) byte, thereby providing compatibility with MC 68000™ and IBM 370™ conventions. Figure 1-4 shows this configuration.

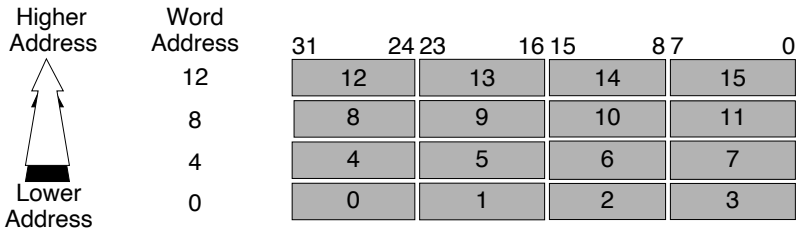


Figure 1-4 Big-Endian Byte Ordering

Remarks 1. The most-significant byte is the lowest address.

2. A word is addressed by the address of the most-significant byte.

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

Unless otherwise specified, the little endian is used throughout this manual.

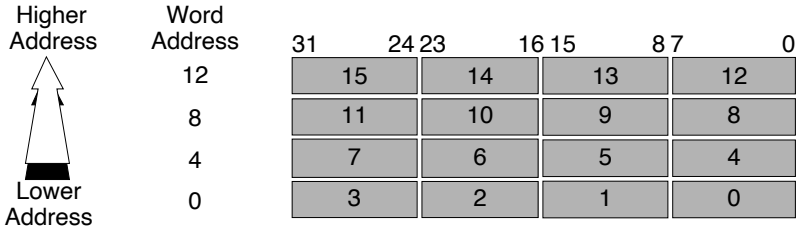


Figure 1-5 Little-Endian Byte Ordering

Remarks 1. The least-significant byte is the lowest address.

2. A word is addressed by the address of the least-significant byte.

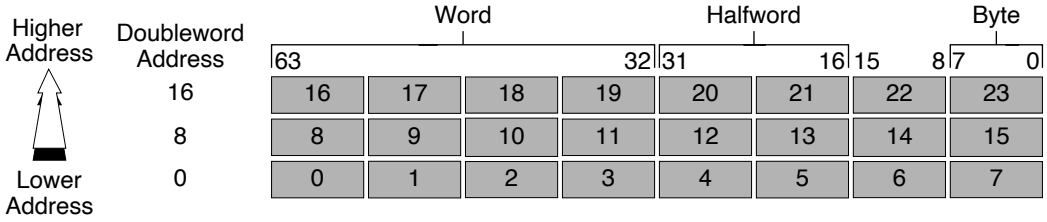


Figure 1-6 Big-Endian Data in a Doubleword

Remarks 1. The most-significant byte is the lowest address.

2. A word is addressed by the address of the most-significant byte.

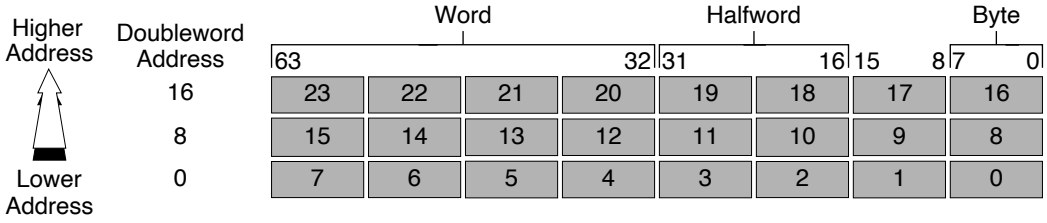


Figure 1-7 Little-Endian Data in a Doubleword

Remarks 1. The least-significant byte is the lowest address.

2. A word is addressed by the address of the least-significant byte.

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 always used in pairs to access data not aligned at an boundary. To access data not aligned at a boundary, additional 1P cycle is necessary as compared when accessing data aligned at a boundary.

Figure 1-8 illustrates how a word misaligned and having byte address 3 is accessed in big and little endian.

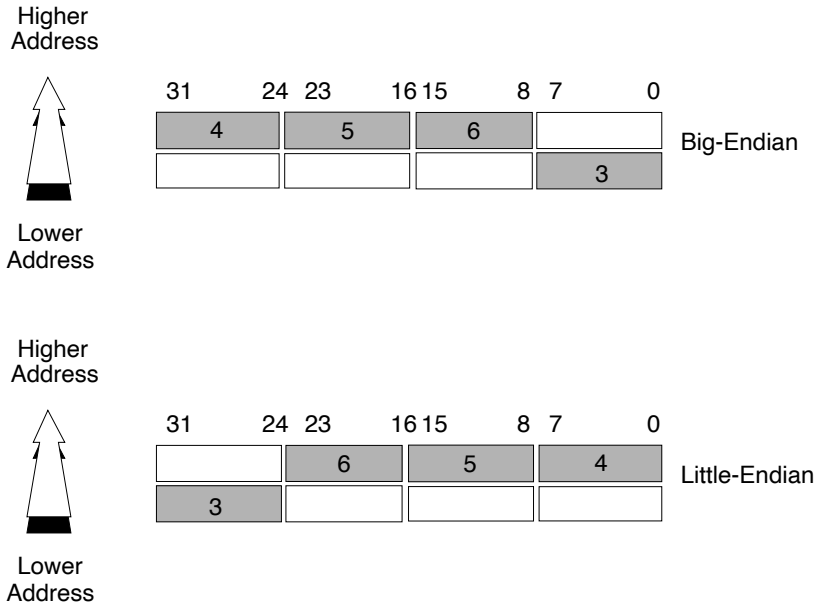


Figure 1-8 Misaligned Word Addressing

1.4.5 System Control Coprocessor (CP0)

ISA of MIPS defines four types of coprocessors (CP0 through CP3). CP0 is an internal system control coprocessor and supports a virtual memory system and exception processing. CP1 is an internal floating-point unit. CP2 is reserved for future definition. CP3 is also reserved for expansion. If the CP3 instruction is executed, a reserved instruction exception occurs.

CP0 converts virtual addresses into physical addresses, selects an operating mode (Kernel, supervisor, or user mode), and control exceptions. It also controls the cache subsystem to analyze causes and return execution from error processing. The CP0 register of the V_R4300 is the same as that of the V_R4200. Because the V_R4300 does not have a parity check function, however, its parity error register (26) and cache error register (27) do not practically operate. These registers are defined to maintain compatibility with the V_R4200.

Figure 1-9 shows the CP0 register. Table 1-2 briefly explains each register. For the details of the registers related to the virtual memory system, refer to **Chapter 5 Memory Management System**, and for the details of the registers used for exception processing, refer to **Chapter 6 Exception Processing.**

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*	*Parity Error*	*26*
Compare	*11*	*Cache Error*	*27*
Status	*12*	*TagLo*	*28*
Cause	*13*	*TagHi*	*29*
EPC	*14*	*ErrorEPC*	*30*
PRId	*15*		*31*

Memory Management *Exception Processing* *For Future Use*

Figure 1-9 CP0 Registers

Table 1-2 System Control Coprocessor (CP0) Register Definitions

Number	Register	Description
0	Index	Programmable pointer into TLB array
1	Random	Pseudorandom pointer into TLB array *(read only)*
2	EntryLo0	Low half of TLB entry for even virtual address (VPN)
3	EntryLo1	Low half of TLB entry for odd virtual address (VPN)
4	Context	Pointer to kernel virtual page table entry (PTE) in 32-bit mode
5	PageMask	Page size specification
6	Wired	Number of wired TLB entries
7	—	Reserved for future use
8	BadVAddr	Display of virtual address that occurred an error last
9	Count	Timer Count
10	EntryHi	High half of TLB entry (including ASID)
11	Compare	Timer Compare Value
12	Status	Operation status setting
13	Cause	Display of cause of last exception
14	EPC	Exception Program Counter
15	PRId	Processor Revision Identifier
16	Config	Memory system mode setting
17	LLAddr	Load Linked instruction address display
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 mode
21–25	—	Reserved for future use
26	Parity Error*	Cache parity bits
27	Cache Error*	Cache Error and Status register
28	TagLo	Cache Tag register low
29	TagHi	Cache Tag register high
30	ErrorEPC	Error Exception Program Counter
31	—	Reserved for future use

* These registers are defined to maintain compatibility with the V_R4200, and not used with the hardware of the V_R4300.

1.4.6 Floating-Point Unit (FPU), CP1

The floating-point unit (FPU) operates as a coprocessor for the CPU and performs arithmetic operations on floating-point values. The FPU, with associated system software, fully conforms to the requirements of ANSI/IEEE Standard 754–1985, *IEEE Standard for Binary Floating-Point Arithmetic*.

The FPU includes:

- **Full 64-bit Operation**. The FPU can contain either 16 64-bit registers to hold single-precision or double-precision values. Another sixteen floating-point registers can be used by setting the FR bit of the *Status* register to 1. Moreover, a 32-bit *Control*/*Status* register is provided, conforming to the IEEE exception processing standard.
- **Load and Store Instruction Set**. Like the CPU, the FPU uses a load- and store-based instruction set. Floating-point operations are started in a single cycle, however execution of floating-point ops are not allowed to overlap other operations.
- **Sharing Hardware**. There is no separate FPU on the V_R4300; floating-point operations are processed by the same hardware as is used for integer instructions.

1.4.7 Internal Cache

The V_R4300 has an instruction cache and a data cache to enhance the efficiency of pipelining. Each cache has a data width of 64 bits and can be accessed in 1 clock. The instruction cache and data cache can be accessed in parallel. The instruction cache has a capacity of 16K bytes, while the data cache has a capacity of 8K bytes.

For the details of the cache, refer to **Chapter 11 Cache Memory**.

1.5 Memory Management System (MMU)

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

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

A detailed description of these address spaces is given in **Chapter 5 Memory Management System**.

1.5.1 Translation Lookaside Buffer (TLB)

Virtual memory mapping is assisted by a translation lookaside buffer, which holds virtual-to-physical address translations. This fully-associative, on-chip TLB contains 32 entries, each of which maps a pair of variable-sized pages of either 4 KB or 16 MB.

Joint TLB (JTLB)

The TLB can hold both instruction and data addresses, and is thus also referred to as a joint TLB (JTLB).

An address translation value is tagged with the high-order bits of its virtual address (the number of these bits depends upon the size of the page) and a per-process identifier. If there is no matching entry in the TLB, an exception occurs and software writes the entry contents to the on-chip TLB from a page table in memory. The JTLB entry to be rewritten is selected by a value in either the *Random* or *Index* register.

Instruction Micro-TLB (ITLB)

The V_R4300 processor has a two-entry instruction micro-TLB (ITLB) which assists in instruction address translation. The ITLB can not be operated directly by the software. Instructions access this TLB while data accesses the Joint TLB; a miss in the micro-TLB stalls the pipeline until the micro-TLB is refilled from the joint TLB. The micro-TLB is fully associative, and uses the least-recently-used (LRU) replacement algorithm. Each micro-TLB entry maps 4 KB of virtual space to physical space. This ensures each ITLB entry is a subset of any single JTLB entry.

1.5.2 Operating Modes

The V_R4300 processor has three operating modes:

- User mode
- Supervisor mode
- Kernel mode

The manner in which memory addresses are translated or *mapped* depends on the operating mode of the CPU; this is described in **Chapter 5 Memory Management System**.

1.6 Instruction Pipeline

The V_R4300 has a 5-stage instruction pipeline. This pipeline is used for floating-point operations as well as for integer operations. In a normal environment, the pipeline executes one instruction in 1 cycle.

The pipeline of the V_R4300 operates at a frequency determined depending on the setting of the DivMode(1:0)* pins. For details, refer to **Chapter 4 Pipeline**.

** In V_R4300 and V_R4305. In V_R4310, DivMode(2:0).*

[MEMO]

Pin Functions

2

2.1 Pin Configuration (Top View)

- 120-pin plastic QFP (28 × 28 mm)

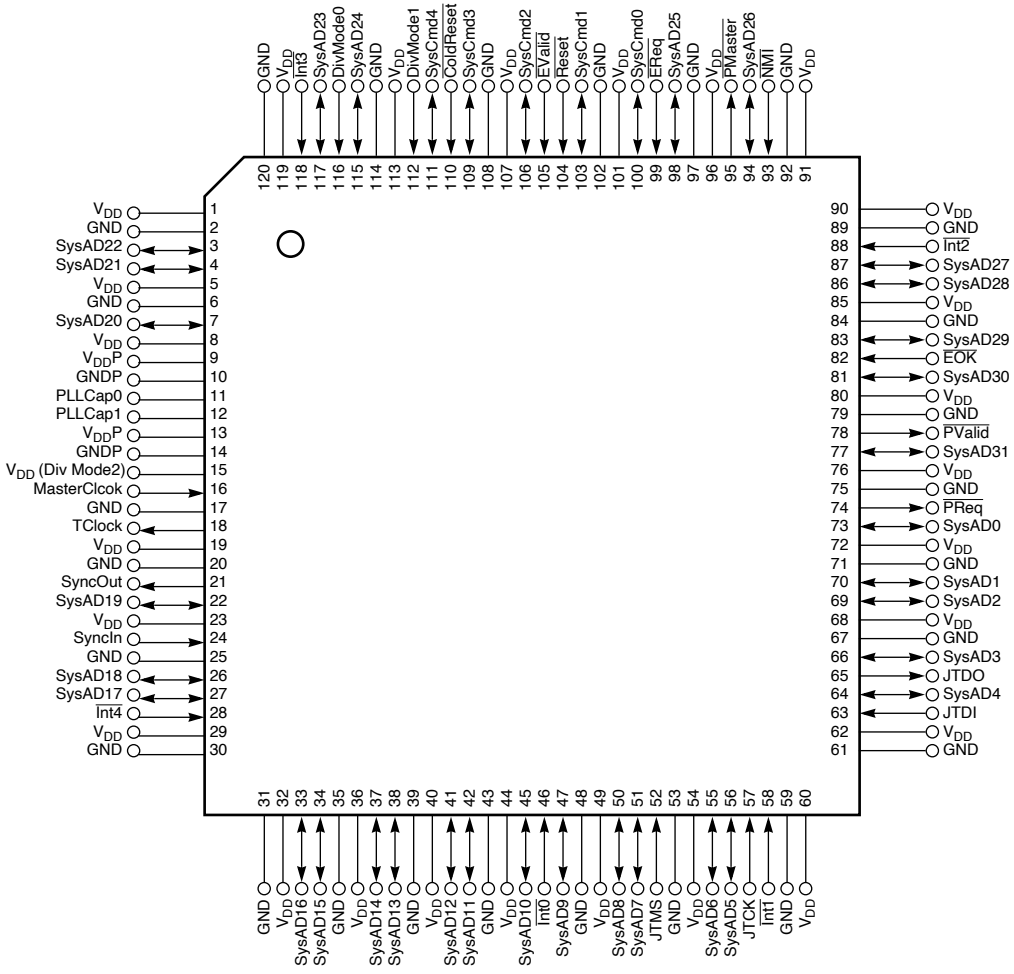
  μPD30200GD-80-LBB

  μPD30200GD-100-MBB

  μPD30200GD-133-MBB

  μPD30210GD-133-MBB

  μPD30210GD-167-MBB



Remark *( ): Pin name of the μPD30210-xxx*

PIN NAME

$\overline{\text{ColdReset}}$	:	Cold Reset
DivMode (1:0)*	:	Divide Mode
$\overline{\text{EOK}}$	:	External OK
$\overline{\text{EReq}}$	:	External Request
$\overline{\text{EValid}}$	:	External Valid
$\overline{\text{Int (4:0)}}$	:	Interrupt Request
JTCK	:	JTAG Clock Input
JTDI	:	JTAG Data In
JTDO	:	JTAG Data Out
JTMS	:	JTAG Command Signal
MasterClock	:	Master Clock
$\overline{\text{NMI}}$	:	Non-maskable Interrupt Request
PLLCap (1:0)	:	Phase Locked Loop Capacitance
$\overline{\text{PMaster}}$	:	Processor Master
$\overline{\text{PReq}}$	:	Processor Request
$\overline{\text{PValid}}$	:	Processor Valid
$\overline{\text{Reset}}$	:	Reset
Syncln	:	Synchronization Clock Input
SyncOut	:	Synchronization Clock Output
SysAD (31:0)	:	System Address/Data Bus
SysCmd (4:0)	:	System Command Data ID Bus
TClock	:	Transmit Clock
V_{DD}	:	Power Supply
GND	:	Ground
$V_{DD}P$	:	V_{DD} for PLL
GNDP	:	GND for PLL

** In the μPD30200-×××. DivMode (2:0) in the μPD30210-×××.*

2.2 Pin Functions

2.2.1 System Interface Signals

The system interface signals are used when the V_R4300 is connected with an external device in the system. Table 2-1 indicates the functions of these signals.

Table 2-1 System Interface Signals

Signal Name	Definition	I/O	Function
SysAD(31:0)	System address/data bus	I/O	32-bit address/data bus. Used to transmit or receive data or address between the processor and the external agent.
SysCmd(4:0)	System command/data ID bus	I/O	5-bit bus. Used to transfer commands or data identifiers between the processor and the external agent.
$\overline{\text{EReq}}$	External request	Input	Asserted active when the external agent requests the processor for the system interface.
$\overline{\text{PReq}}$	Processor request	Output	Asserted active when the processor requests the external agent for the system interface. If a protocol error is detected in the system interface, this signal is oscillated in synchronization with MasterClock in a cycle which is a multiple of SClock.
$\overline{\text{EValid}}$	External agent valid	Input	Asserted active when the external agent drives a valid address or valid data onto the SysAD bus, and a valid command/data identifier is on the SysCmd bus.
$\overline{\text{PValid}}$	Processor valid	Output	Asserted active when the processor drives a valid address or data onto the SysAD bus, and a valid command/data identifier is on the SysCmd bus.
$\overline{\text{PMaster}}$	Processor master	Output	Asserted active when the processor is the master of the system interface bus.
$\overline{\text{EOK}}$	External ready	Input	Asserted active when the external agent is ready to accept a processor request.

2.2.2 Clock/Control Interface Signals

These interface signals are used to supply or control clocks. Table 2-2 shows the functions of the signals.

Table 2-2 Clock/Control Interface Signals (1/3)

<table>
<tr><th>Signal Name</th><th>Definition</th><th>I/O</th><th>Function</th></tr>
<tr><td>MasterClock</td><td>Master clock</td><td>Input</td><td>Inputs the MasterClock from this pin. The internal operating speed is determined by the frequency of this signal and the contents of the DivMode signals.</td></tr>
<tr><td>TClock</td><td>Transmit/receive clock</td><td>Output</td><td>Outputs the transmit/receive clock at the same frequency as the MasterClock.</td></tr>
<tr><td>SyncOut</td><td>Synchronization clock output</td><td>Output</td><td>Outputs a synchronization clock. Connect this pin to SyncIn. Model the mutual connection between TClock and external agent.</td></tr>
<tr><td>SyncIn</td><td>Synchronization clock input</td><td>Input</td><td>Inputs a synchronization clock.</td></tr>
<tr><td>$V_{DD}P$</td><td>Static V_{DD} for PLL</td><td>–</td><td>This pins is static V_{DD} for the internal PLL circuit.</td></tr>
<tr><td>GNDP</td><td>Static GND for PLL</td><td>–</td><td>This pin is static GND for the internal PLL circuit.</td></tr>
<tr><td>PLLCap(1:0)</td><td>Adjusting PLL</td><td>–</td><td>This pin connects a capacitor for adjusting the internal PLL circuit of the processor.</td></tr>
<tr><td>DivMode</td><td>Internal operating frequency mode</td><td>Input</td><td>Indicates the ratio at which the internal PClock is generated from the MasterClock.
Normally, the frequency of the TClock is the same as that of the MasterClock.
Do not change the value of these pins after setting the value on power application.
Otherwise, the operation will not guaranteed.

The following indicates the relationship between the DivMode values and frequency ratio of each product.

Remark The maximum value of PClock is the same as the maximum internal operating frequencies of each product regardless of the frequency ratio. (Refer to 1.2 Ordering Information.)

• V_R4300
μPD30200-100
<table>
<tr><th rowspan="2">DivMode (1 : 0)</th><th colspan="2">MasterClock : PClock : TClock</th></tr>
<tr><th>Frequency ratio</th><th>Example [MHz]</th></tr>
<tr><td>00</td><td>RFU</td><td>–</td></tr>
<tr><td>01</td><td>2 : 3 : 2</td><td>66.7 : 100 : 66.7</td></tr>
<tr><td>10</td><td>1 : 2 : 1</td><td>50 : 100 : 50</td></tr>
<tr><td>11</td><td>1 : 3 : 1</td><td>33.3 : 100 : 33.3</td></tr>
</table>
</td></tr>
</table>

Table 2-2 Clock/Control Interface Signals (2/3)

<table>
<tr><th>Signal Name</th><th>Definition</th><th>I/O</th><th>Function</th></tr>
<tr><td>DivMode</td><td>Internal operating frequency mode</td><td>Input</td><td>

• V_R4300

µPD30200-133

<table>
<tr><th rowspan="2">DivMode (1 : 0)</th><th colspan="2">MasterClock : PClock : TClock</th></tr>
<tr><th>Frequency ratio</th><th>Example [MHz]</th></tr>
<tr><td>00</td><td>1 : 4 : 1</td><td>33.3 : 133 : 33.3</td></tr>
<tr><td>01</td><td>RFU</td><td>–</td></tr>
<tr><td>10</td><td>1 : 2 : 1</td><td>66.7 : 133 : 66.7</td></tr>
<tr><td>11</td><td>1 : 3 : 1</td><td>44.3 : 133 : 44.3</td></tr>
</table>

• V_R4305

µPD30200-80

<table>
<tr><th rowspan="2">DivMode (1 : 0)</th><th colspan="2">MasterClock : PClock : TClock</th></tr>
<tr><th>Frequency ratio</th><th>Example [MHz]</th></tr>
<tr><td>00</td><td>1 : 1 : 1</td><td>66.7 : 66.7 : 66.7</td></tr>
<tr><td>01</td><td>RFU</td><td>–</td></tr>
<tr><td>10</td><td>1 : 2 : 1</td><td>40 : 80 : 40</td></tr>
<tr><td>11</td><td>1 : 3 : 1</td><td>20 : 60 : 20</td></tr>
</table>

• V_R4310

µPD30210-133

<table>
<tr><th rowspan="2">DivMode (2 : 0)</th><th colspan="2">MasterClock : PClock : TClock</th></tr>
<tr><th>Frequency ratio</th><th>Example [MHz]</th></tr>
<tr><td>000</td><td>1 : 5 : 1</td><td>26.7 : 133 : 26.7</td></tr>
<tr><td>001</td><td>1 : 6 : 1</td><td>22.2 : 133 : 22.2</td></tr>
<tr><td>010</td><td>RFU</td><td>–</td></tr>
<tr><td>011</td><td>1 : 3 : 1</td><td>33.3 : 100 : 33.3</td></tr>
<tr><td>100</td><td>1 : 4 : 1</td><td>33.3 : 133 : 33.3</td></tr>
<tr><td>101</td><td>RFU</td><td>–</td></tr>
<tr><td>110</td><td>1 : 2 : 1</td><td>50 : 100 : 50</td></tr>
<tr><td>111</td><td>1 : 3 : 1</td><td>33.3 : 100 : 33.3</td></tr>
</table>

</td></tr>
</table>

Table 2-2 Clock/Control Interface Signals (3/3)

<table>
<tr><th>Signal Name</th><th>Definition</th><th>I/O</th><th>Function</th></tr>
<tr><td>DivMode</td><td>Internal operating frequency mode</td><td>Input</td><td>• V_R4310
µPD30210-167
<table>
<tr><th rowspan="2">DivMode (2 : 0)</th><th colspan="2">MasterClock : PClock : TClock</th></tr>
<tr><th>Frequency ratio</th><th>Example [MHz]</th></tr>
<tr><td>000</td><td>1 : 5 : 1</td><td>33.3 : 167 : 33.3</td></tr>
<tr><td>001</td><td>1 : 6 : 1</td><td>27.8 : 167 : 27.8</td></tr>
<tr><td>010</td><td>2 : 5 : 2</td><td>66.7 : 167 : 66.7</td></tr>
<tr><td>011</td><td>1 : 3 : 1</td><td>33.3 : 100 : 33.3</td></tr>
<tr><td>100</td><td>1 : 4 : 1</td><td>33.3 : 133 : 33.3</td></tr>
<tr><td>101</td><td>RFU</td><td>–</td></tr>
<tr><td>110</td><td>1 : 2 : 1</td><td>50 : 100 : 50</td></tr>
<tr><td>111</td><td>1 : 3 : 1</td><td>33.3 : 100 : 33.3</td></tr>
</table></td></tr>
</table>

2.2.3 Interrupt Interface Signals

These signals are used by the external device to issue interrupt requests to the V_R4300. Table 2-3 shows the functions of these signals.

Table 2-3 Interrupt Interface Signals

Signal Name	Definition	I/O	Function
$\overline{\text{Int}}$(4:0)	Interrupt request acknowledge	Input	General purpose interrupt request pins. These pins are ORed with the bits 4 through 0 of the internal interrupt register.
$\overline{\text{NMI}}$	Non-maskable interrupt	Input	This pin accepts the non-maskable interrupt signal. It is ORed with the bit 6 of the internal interrupt register.

2.2.4 Joint Test Action Group (JTAG) Interface Signals

These signals are for interfacing the boundary scan of JTAG. Table 2-4 shows the functions of these signals.

Table 2-4 JTAG Interface Signals

Signal Name	Definition	I/O	Function
JTDI	JTAG data input	Input	Inputs data to be scanned serially.
JTCK	JTAG clock input	Input	Inputs a serial clock. JTDI and JTMS are read simultaneously at the rising edge of this signal. Fix this signal to the low level when the JTAG interface is not used.
JTDO	JTAG data output	Output	Outputs serially scanned data.
JTMS	JTAG command	Input	Inputs a high level to this pin if the serial data to be input next is a command of the JTAG.

2.2.5 Initialization Interface Signals

These signals are used when the external device initializes the operation parameters of the processor. Table 2-5 shows the functions of these signals.

Table 2-5 Initialization Interface Signals

Signal Name	Definition	I/O	Function
$\overline{\text{ColdReset}}$	Cold reset	Input	Asserted active at cold reset. SClock and TClock start the cycle at the rising edge of this signal. This signal needs not be asserted active or deasserted inactive in synchronization with the MasterClock signal.
$\overline{\text{Reset}}$	Reset	Input	Make this pin active or inactive in synchronization with MasterClock, or keep it inactive at cold reset. Make this pin active or inactive in synchronization with MasterClock at soft reset.

CPU Instruction Set Summary

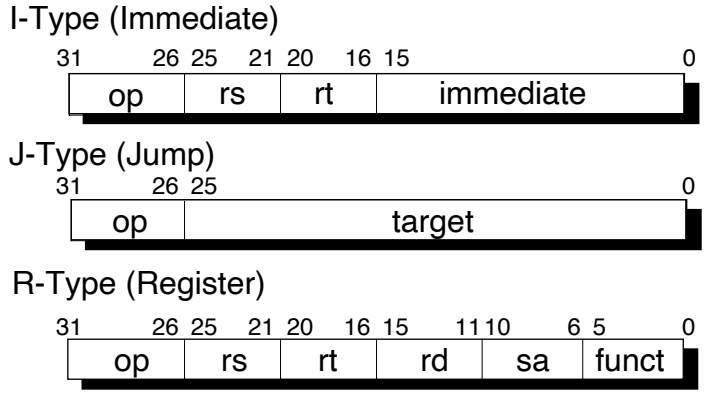
3

This chapter is an overview of the central processing unit (CPU) instruction set; refer to **Chapter 16 CPU Instruction Set Details** for detailed descriptions of individual CPU instructions.

Because the FPU instruction is dependent upon the structure of the coprocessor, refer to **Chapter 7 Floating-Point Operations** and **Chapter 17 FPU Instruction Set Details**.

3.1 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 3-1. By simplifying the instruction format in three ways, decoding instructions is simplified. Complicated and less frequently used operations and addressing modes are implemented by combining two or more instructions by using a compiler.



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

Figure 3-1 CPU Instruction Formats

Support of the MIPS ISA

Even though the V_R4300 processor does not support a multiprocessor operating environment, the synchronization support instructions defined in the MIPS II and MIPS III ISA—the Load Linked and Store Conditional instructions—are processed correctly, in order to maintain compatibility with V_R4400 and V_R4200. The load link bit (*LLbit*) is set by the LL instruction, cleared by an ERET, and tested by the SC instruction. The only operation to the *LLbit* that can be implemented is a reset due to cache invalidation.

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

3.2 Instruction Classes

The CPU instructions can be classified into six classes.

3.2.1 Load/Store Instructions

Load and store are immediate (I-type) instructions that move data between memory and the general purpose registers. Only a mode that adds a 16-bit signed immediate offset to the base register is available as the addressing mode of the load/store instructions.

Scheduling a Load Delay Slot

A load instruction whose loading result cannot be used by the instruction immediately following is called a delayed load instruction. The instruction slot immediately after a delayed load instruction is called a load delay slot. With the V_R4000 Series, an instruction including the load destination register can be described immediately after a load instruction. In this case, however, the interlock count is generated equal to the number of necessary cycles. Therefore, although any instruction can be described, it is recommended to schedule the load delay slot to improve the performances of the V_R4300 and to maintain its compatibility with the V_R3000 Series (for details, refer to **Chapter 4 Pipeline**).

Store Delay Slot

In the V_R4300 processor, a store instruction writing to the data cache keeps the data cache busy during both its DC and WB stages. If the instruction immediately following needs to access the data cache in its DC stage (e.g. a load instruction), the hardware interlocks. Consequently, scheduling store delay slots can be desirable for performance.

Table 3-1 Number of Cycles for Load and Store Instruction Delay Slot

Instruction	PCycles Required
Load	1
Store	1

Defining Access Types

Access type is the size of the data loaded/stored by the processor.

The op code of the load/store instruction determines the access type. Figure 3-2 shows the access type and the data to be loaded/stored. The address used for the load/store instruction is the least significant byte address (most significant byte in big endian and the address indicating the least significant byte in little endian), regardless of the access type and byte ordering (endianness).

The byte ordering in the doubleword of the data to be accessed is determined by the access type and the low-order 3 bits of the address, as shown in Figure 3-2. Combinations of an access type and the low-order bits of an address other than those shown in Figure 3-2 are prohibited. If a combination other than those shown in the figure is used, an address error exception occurs.

Table 3-2 lists the load/store instructions defined by ISA, and Table 3-3 lists the instructions of the extended ISA.

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

Figure 3-2 Byte Access within a Doubleword

Table 3-2 Load/Store Instructions (1/2)

Instruction	Format and Description	op	base	rt	offset
Load Byte	LB rt, offset (base) Generates an address by adding a sign-extended offset to the contents of register base. Sign-extends the contents of a byte specified by the address and loads the result to register rt.				
Load Byte Unsigned	LBU rt, offset (base) Generates an address by adding a sign-extended offset to the contents of register base. Zero-extends the contents of a byte specified by the address and loads the result to register rt.				
Load Halfword	LH rt, offset (base) Generates an address by adding a sign-extended offset to the contents of register base. Sign-extends the contents of a halfword specified by the address and loads the result to register rt.				
Load Halfword Unsigned	LHU rt, offset (base) Generates an address by adding a sign-extended offset to the contents of register base Zero-extends the contents of a halfword specified by the address and loads the result to register rt.				
Load Word	LW rt, offset (base) Generates an address by adding a sign-extended offset to the contents of register base. Sign-extends the contents of a word specified by the address (in the 64-bit mode) and loads the result to register rt.				
Load Word Left	LWL rt, offset (base) Generates an address by adding a sign-extended offset to the contents of register base. Shifts a word specified by the address to the left, so that a byte specified by the address is at the leftmost position of the word. Sign-extends (in the 64-bit mode), merges the result of the shift and the contents of register rt, and loads the result to register rt.				
Load Word Right	LWR rt, offset (base) Generates an address by adding a sign-extended offset to the contents of register base. Shifts a word specified by the address to the right, so that a byte specified by the address is at the rightmost position of the word. Sign-extends (in the 64-bit mode), merges the result of the shift and the contents of register rt, and loads the result to register rt.				

Table 3-2 Load/Store Instructions (2/2)

Instruction	Format and Description	op	base	rt	offset
Store Byte	SB rt, offset (base) Generates an address by adding a sign-extended offset to the contents of register base. Stores the contents of the low-order byte of register rt to the memory specified by the address.				
Store Halfword	SH rt, offset (base) Generates an address by adding a sign-extended offset to the contents of register base. Stores the contents of the low-order halfword of register rt to the memory specified by the address.				
Store Word	SW rt, offset (base) Generates an address by adding a sign-extended offset to the contents of register base. Stores the contents of the low-order word of register rt to the memory specified by the address.				
Store Word Left	SWL rt, offset (base) Generates an address by adding a sign-extended offset to the contents of register base. Shifts the contents of register rt to the right so that the leftmost byte of the word is at the position of the byte specified by the address. Stores the result of the shift to the lower portion of the word in memory.				
Store Word Right	SWR rt, offset (base) Generates an address by adding a sign-extended offset to the contents of register base. Shifts the contents of register rt to the left so that the rightmost byte of the word is at the position of the byte specified by the address. Stores the result of the shift to the higher portion of the word in memory.				

Table 3-3 Load/Store Instructions (Extended ISA) (1/2)

Instruction	Format and Description	op	base	rt	offset
Load Doubleword	LD rt, offset (base) Generates an address by adding a sign-extended offset to the contents of register base. Loads the contents of the doubleword specified by the address to register rt.				
Load Doubleword Left	LDL rt, offset (base) Generates an address by adding a sign-extended offset to the contents of register base. Shifts the doubleword specified by the address to the left so that the byte specified by the address is at the leftmost position of the doubleword. Merges the result of the shift and the contents of register rt, and loads the result to register rt.				
Load Doubleword Right	LDR rt, offset (base) Generates an address by adding a sign-extended offset to the contents of register base. Shifts the doubleword specified by the address to the right so that the byte specified by the address is at the rightmost position of the doubleword. Merges the result of the shift and the contents of register rt, and loads the result to register rt.				
Load Linked	LL rt, offset (base) Generates an address by adding a sign-extended offset to the contents of register base. Loads the contents of the word specified by the address to register rt nd sets the LL bit to 1.				
Load Linked Doubleword	LLD rt, offset (base) Generates an address by adding a sign-extended offset to the contents of register base. Loads the contents of the doubleword specified by the address to register rt and sets the LL bit to 1.				
Load Word Unsigned	LWU rt, offset (base) Generates an address by adding a sign-extended offset to the contents of register base. Zero-extends the contents of the word specified by the address, and loads the result to register rt.				

Table 3-3 Load/Store Instructions (Extended ISA) (2/2)

Instruction	Format and Description	op	base	rt	offset
Store Conditional	SC rt, offset (base) Generates an address by adding a sign-extended offset to the contents of register base. If the LL bit is 1, stores the contents of the low-order word of register rt to the memory specified by the address, and sets register rt to 1. If the LL bit is 0, does not store the contents of the word, and clears register rt to 0.				
Store Conditional Doubleword	SCD rt, offset (base) Generates an address by adding a sign-extended offset to the contents of register base. If the LL bit is 1, stores the contents of register rt to the memory specified by the address, and sets register rt to 1. If the LL bit is 0, does not store the contents of the register, and clears register rt to 0.				
Store Doubleword	SD rt, offset (base) Generates an address by adding a sign-extended offset to the contents of register base. Stores the contents of register rt to the memory specified by the address.				
Store Doubleword Left	SDL rt, offset (base) Generates an address by adding a sign-extended offset to the contents of register base. Shifts the contents of register rt to the right so that the leftmost byte of a doubleword is at the position of the byte specified by the address. Stores the result of the shift to the lower portion of the doubleword in memory.				
Store Doubleword Right	SDR rf, offset (base) Generates an address by adding a sign-extended offset to the contents of register base. Shifts the contents of register rt to the left so that the rightmost byte of a doubleword is at the position of the byte specified by the address. Stores the result of the shift to the higher portion of the doubleword in memory.				

3.2.2 Computational Instructions

Computational instructions executes arithmetic operations, multiply/divide, logical operations, and shift operations on the values of registers. These instructions are classified into two types: R-type and I-type. The R-type instructions uses registers as both the source, and the I-type instructions uses an immediate value as one of the sources. The operation instructions are divided into the following four types by classification of operation.

(1) ALU immediate instructions (Refer to **Tables 3-4** and **3-5**.)

(2) 3-operand type instructions (Refer to **Tables 3-6** and **3-7**.)

(3) Shift instructions (Refer to **Tables 3-8** and **3-9**.)

(4) Multiply/Divide instructions (Refer to **Tables 3-10** and **3-11**.)

If compatibility of data is necessary in the 64-bit and 32-bit modes, the 32-bit operands must be correctly sign-extended. Otherwise, the 32-bit value of the result of the operation will be meaningless.

Table 3-4 ALU Immediate Instructions

Instruction	Format and Description	op	rs	rt	immediate
Add Immediate	ADDI rt, rs, immediate Sign-extends the 16-bit immediate and adds it to register rs. Stores the 32-bit result to register rt (sign-extends the result in the 64-bit mode). Generates an exception if a 2's complement integer overflow occurs.				
Add Immediate Unsigned	ADDIU rt, rs, immediate Sign-extends the 16-bit immediate and adds it to register rs. Stores the 32-bit result to register rt (sign-extends the result in the 64-bit mode). Does not generate an exception even if an integer overflow occurs.				
Set On Less Than Immediate	SLTI rt, rs, immediate Sign-extends the 16-bit immediate and compares it with register rs as a signed integer. If rs is less than the immediate, stores 1 to register rt; otherwise, stores 0 to register rt.				
Set On Less Than Immediate Unsigned	SLTIU rt, rs, immediate Sign-extends the 16-bit immediate and compares it with register rs as an unsigned integer. If rs is less than the immediate, stores 1 to register rt; otherwise, stores 0 to register rt.				
And Immediate	ANDI rt, rs, immediate Zero-extends the 16-bit immediate, ANDs it with register rs, and stores the result to register rt.				
Or Immediate	ORI rt, rs, immediate Zero-extends the 16-bit immediate, ORs it with register rs, and stores the result to register rt.				
Exclusive Or Immediate	XORI rt, rs, immediate Zero-extends the 16-bit immediate, exclusive-ORs it with register rs, and stores the result to register rt.				
Load Upper Immediate	LUI rt, immediate Shifts the 16-bit immediate 16 bits to the left, and clears the low-order 16 bits of the word to 0. Stores the result to register rt (by sign-extending the result in the 64-bit mode).				

Table 3-5 ALU Immediate Instruction (Extended ISA)

Instruction	Format and Description	op	rs	rt	immediate
Doubleword Add Immediate	DADDI rt, rs, immediate Sign-extends the 16-bit immediate to 64 bits, and adds it to register rs. Stores the 64-bit result to register rt. Generates an exception if an integer overflow occurs.				
Doubleword Add Immediate Unsigned	DADDIU rt, rs immediate Sign-extends the 16-bit immediate to 64 bits, and adds it to register rs. Stores the 64-bit result to register rt. Does not generate an exception even if an integer overflow occurs.				

Table 3-6 Three-Operand Type Instruction

Instruction	Format and Description	op	rs	rt	rd	sa	funct
Add	ADD rd, rs, rt Adds the contents of register rs and rt, and stores (sign-extends in the 64-bit mode) the 32-bit result to register rd. Generates an exception if an integer overflow occurs.						
Add Unsigned	ADDU rd, rs, rt Adds the contents of register rs and rt, and stores (sign-extends in the 64-bit mode) the 32-bit result to register rd. Does not generate an exception even if an integer overflow occurs.						
Subtract	SUB rd, rs, rt Subtracts the contents of register rs from register rt, and stores (sign-extends in the 64-bit mode) the result to register rd. Generates an exception if an integer overflow occurs.						
Subtract Unsigned	SUBU rd, rs, rt Subtracts the contents of register rt from register rs, and stores (sign-extends in the 64-bit mode) the 32-bit result to register rd. Does not generate an exception even if an integer overflow occurs.						
Set On Less Than	SLT rd, rs, rt Compares the contents of registers rs and rt as signed integers. If the contents of register rs are less than those of rt, stores 1 to register rd; otherwise, stores 0 to rd.						
Set On Less Than Unsigned	SLTU rd, rs, rt Compares the contents of registers rs and rt as unsigned integers. If the contents of register rs are less than those of rt, stores 1 to register rd; otherwise, stores 0 to rd.						
And	AND rd, rs, rt ANDs the contents of registers rs and rt in bit units, and stores the result to register rd.						
Or	OR rd, rs, rt ORs the contents of registers rs and rt in bit units, and stores the result to register rd.						
Exclusive Or	XOR rd, rs, rt Exclusive-ORs the contents of registers rs and rt in bit units, and stores the result to register rd.						
Nor	NOR rd, rs, rt NORs the contents of registers rs and rt in bit units, and stores the result to register rd.						

Table 3-7 Three-Operand Type Instructions (Extended ISA)

Instruction	Format and Description	op	rs	rt	rd	sa	funct
Doubleword Add	DADD rd, rs, rt Adds the contents of registers rs and rt, and stores the 64-bit result to register rd. Generates an exception if an integer overflow occurs.						
Doubleword Add Unsigned	DADDU rd, rs, rt Adds the contents of registers rs and rt, and stores the 64-bit result to register rd. Does not generate an exception even if an integer overflow occurs.						
Doubleword Subtract	DSUB rd, rs, rt Subtracts the contents of register rt from register rs, and stores the 64-bit result to register rd. Generates an exception if an integer overflow occurs.						
Doubleword Subtract Unsigned	DSUBU rd, rs, rt Subtracts the contents of register rt from register rs, and stores the 64-bit result to register rd. Does not generate an exception even if an integer overflow occurs.						

Table 3-8 Shift Instructions

Instruction	Format and Description	op	rs	rt	rd	sa	funct
Shift Left Logical	SLL rd, rt, sa Shifts the contents of register rt sa bits to the left, and inserts 0 to the low-order bits. Sign-extends (in the 64-bit mode) the 32-bit result and stores it to register rd.						
Shift Right Logical	SRL rd, rt, sa Shifts the contents of register rt sa bits to the right, and inserts 0 to the high-order bits. Sign-extends (in the 64-bit mode) the 32-bit result and stores it to register rd.						
Shift Right Arithmetic	SRA rd, rt, sa Shifts the contents of register rt sa bits to the right, and sign-extends the high-order bits. Sign-extends (in the 64-bit mode) the 32-bit result and stores it to register rd.						
Shift Left Logical Variable	SLLV rd, rt, rs Shifts the contents of register rt to the left and inserts 0 to the low-order bits. The number of bits by which the register contents are to be shifted is specified by the low-order 5 bits of register rs. Sign-extends (in the 64-bit mode) the result and stores it to register rd.						
Shift Right Logical Variable	SRLV rd, rt, rs Shifts the contents of register rt to the right, and inserts 0 to the high-order bits. The number of bits by which the register contents are to be shifted is specified by the low-order 5 bits of register rs. Sign-extends (in the 64-bit mode) the 32-bit result and stores it to register rd.						
Shift Right Arithmetic Variable	SRAV rd, rt, rs Shifts the contents of register rt to the right and sign-extends the high-order bits. The number of bits by which the register contents are to be shifted is specified by the low-order 5 bits of register rs. Sign-extends (in the 64-bit mode) the 32-bit result and stores it to register rd.						

Table 3-9 Shift Instructions (Extended ISA) (1/2)

Instruction	Format and Description	op	rs	rt	rd	sa	funct
Doubleword Shift Left Logical	DSLL rd, rt, sa Shifts the contents of register rt sa bits to the left, and inserts 0 to the low-order bits. Stores the 64-bit result to register rd.						
Doubleword Shift Right Logical	DSRL rd, rt, sa Shifts the contents of register rt sa bits to the right, and inserts 0 to the high-order bits. Stores the 64-bit result to register rd.						
Doubleword Shift Right Arithmetic	DSRA rd, rt, sa Shifts the contents of register rt sa bits to the right, and sign-extends the high-order bits. Stores the 64-bit result to register rd.						
Doubleword Shift Left Logical Variable	DSLLV rd, rt, rs Shifts the contents of register rt to the left, and inserts 0 to the low-order bits. The number of bits by which the register contents are to be shifted is specified by the low-order 6 bits of register rs. Stores the 64-bit result and stores it to register rd.						
Doubleword Shift Right Logical Variable	DSRLV rd, rt, rs Shifts the contents of register rt to the right, and inserts 0 to the higher bits. The number of bits by which the register contents are to be shifted is specified by the low-order 6 bits of register rs. Sign-extends the 64-bit result and stores it to register rd.						
Doubleword Shift Right Arithmetic Variable	DSRAV rd, rt, rs Shifts the contents of register rt to the right, and sign-extends the high-order bits. The number of bits by which the register contents are to be shifted is specified by the low-order 6 bits of register rs. Sign-extends the 64-bit result and stores it to register rd.						
Doubleword Shift Left Logical + 32	DSLL32 rd, rt, sa Shifts the contents of register rt 32+sa bits to the left, and inserts 0 to the low-order bits. Stores the 64-bit result to register rd.						
Doubleword shift Right Logical + 32	DSRL32 rd, rt, sa Shifts the contents of register rt 32+sa bits to the right, and inserts 0 to the high-order bits. Stores the 64-bit result to register rd.						

Table 3-9 Shift Instructions (Extended ISA) (2/2)

Instruction	Format and Description	op	rs	rt	rd	sa	funct
Doubleword Shift Right Arithmetic + 32	DSRA32 rd, rt, sa Shifts the contents of register rt 32+sa bits to the right, and sign-extends the high-order bits. Stores the 64-bit result to register rd.						

Table 3-10 Multiply/Divide Instructions

Instruction	Format and Description	op	rs	rt	rd	sa	funct
Multiply	MULT rs, rt Multiplies the contents of register rs by the contents of register rt as a 32-bit signed integer. Sign-extends (in the 64-bit mode) and stores the 64-bit result to special registers HI and LO.						
Multiply Unsigned	MULTU rs, rt Multiplies the contents of register rs by the contents of register rt as a 32-bit unsigned integer. Sign-extends (in the 64-bit mode) and stores the 64-bit result to special registers HI and LO.						
Divide	DIV rs, rt Divides the contents of register rs by the contents of register rt. The operand is treated as a 32-bit signed integer. Sign-extends (in the 64-bit mode) and stores the 32-bit quotient to special register LO and the 32-bit remainder to special register HI.						
Divide Unsigned	DIVU rs, rt Divides the contents of register rs by the contents of register rt. The operand is treated as a 32-bit unsigned integer. Sign-extends (in the 64-bit mode) and stores the 32-bit quotient to special register LO and the 32-bit remainder to special register HI.						
Move From HI	MFHI rd Transfers the contents of special register HI to register rd.						
Move From LO	MFLO rd Transfers the contents of special register LO to register rd.						
Move To HI	MTHI rs Transfers the contents of register rs to special register HI.						
Move To LO	MTLO rs Transfers the contents of register rs to special register LO.						

Table 3-11 Multiply/Divide Instructions (Extended ISA)

Instruction	Format and Description	op	rs	rt	rd	sa	funct
Doubleword Multiply	DMULT rs, rt Multiplies the contents of register rs by the contents of register rt as a signed integer. Stores the 128-bit result to special registers HI and LO.						
Doubleword Multiply Unsigned	DMULTU rs, rt Multiplies the contents of register rs by the contents of register rt as an unsigned integer. Stores the 128-bit result to special registers HI and LO.						
Doubleword Divide	DDIV rs, rt Divides the contents of register rs by the contents of register rt. The operand is treated as a signed integer. Stores the 64-bit quotient to special register LO, and the 64-bit remainder to special register HI.						
Doubleword Divide Unsigned	DDIVU rs, rt Divides the contents of register rs by the contents of register rt. The operand is treated as an unsigned integer. Stores the 64-bit quotient to special register LO, and the 64-bit remainder to special register HI.						

When an integer multiply or divide instruction is executed, the V_R4300 stalls the entire pipeline. The number of processor cycles (PCycles) stalled at this time is shown below.

Table 3-12 Number of Cycles Stalled by Multiply/Divide Instruction

Instruction	MULT	MULTU	DIV	DIVU	DMULT	DMULTU	DDIV	DDIVU
Number of required cycles	5	5	37	37	8	8	69	69

3.2.3 Jump/Branch Instructions

The jump and branch instructions change the flow of the program. All the jump and branch instructions generate one delay slot. The instruction immediately following a jump or branch instruction (i.e., the instruction in the delay slot) is executed while the first instruction at the destination is fetched from the memory.

Instructions involving link, such as JAL and BLTZAL, store the return address to register r31.

Table 3-13 Number of Delay Slot Cycles of Jump/Branch Instruction

Instruction	Number of Required Cycles
Branch	1
Jump	1

Outline of Jump Instruction

Subroutine call described in a high-level language usually uses J or JAL instruction. The J and JAL instructions are J-type instructions. An instruction of this type shifts a 26-bit target address 2 bits to the left and combines it with the high-order 4 bits of the current program counter to generate a 32- or 64-bit absolute address.

To return, dispatch, or jump between pages, the JR or JALR instruction is usually used. Both of these instructions are of R-type and references the 32- or 64-bit byte address of a general purpose register.

For details, refer to **Chapter 16 CPU Instruction Set Details**.

Outline of Branch Instruction

The branch instruction has a signed 16-bit offset relative to the program counter. Instructions involving link, such as JAL and BLTZAL, store the return address to register r31.

Table 3-14 lists the jump instructions, and Table 3-15 shows the branch instructions. Table 3-16 lists the branch instructions of the extended ISA.

Table 3-14 Jump Instructions

Instruction	Format and Description	op	target
Jump	J target Shifts the 26-bit target address 2 bits to the left, and jumps to the address coupled with the high-order 4 bits of the PC, delayed by one instruction.		
Jump And Link	JAL target Shifts the 26-bit target address 2 bits to the left, and jumps to the address coupled with the high-order 4 bits of the PC, delayed by one instruction. Stores the address of the instruction following the delay slot to r31 (link register).		

Instruction	Format and Description	op	rs	rt	rd	sa	funct
Jump Register	JR rs Jumps to the address of register rs, delayed by one instruction.						
Jump And Link Register	JALR rs, rd Jumps to the address of register rs, delayed by one instruction. Stores the address of the instruction following the delay slot to register rd.						

The following common limits are applied to Tables 3-15 and 3-16.

Branch Address

The branch addresses of all the branch instructions are calculated by adding a 16-bit offset (signed 64 bits shifted 2 bits to the left) to the address of the instruction in the delay slot. All the branch instructions generate one delay slot.

Operation during No Branch (Table 3-16)

If the branch condition of the branch likely instruction is not satisfied, the instruction in the delay slot is invalidated. The instruction in the delay slot are unconditionally executed for all the other branch instructions.

Remark The instruction at the branch destination is fetched in the EX stage of the branch instruction. Comparison of branch and calculation of the target address are executed in phase 2 of the RF stage and phase 1 of the EX stage of the branch instruction. One cycle of the branch delay slot defined by the architecture is necessary. One cycle of the delay slot is also necessary for the jump instruction. If the branch condition of the branch likely instruction is not satisfied, the instruction in the branch slot are invalidated.

The following symbols in the instruction format in Table 3-15 through Table 3-21 are special.

REGIMM : op code
Sub : sub operation code
CO : sub operation identifier
BC : BC sub operation code
br : branch condition identifier
cofun : coprocessor function area
op : operation code

Table 3-15 Branch Instructions

Instruction	Format and Description	op	rs	rt	offset
Branch On Equal	BEQ rs, rt, offset Branches to the branch address if register rs equals to rt.				
Branch On Not Equal	BNE rs, rt, offset Branches to the branch address if register rs is not equal to rt.				
Branch On Less Than Or Equal To Zero	BLEZ rs, offset Branches to the branch address if register rs is less than 0.				
Branch On Greater Than Zero	BGTZ rs, offset Branches to the branch address if register rs is greater than 0.				

Instruction	Format and Description	REGIMM	rs	sub	offset
Branch On Less Than Zero	BLTZ rs, offset Branches to the branch address if register rs is less than 0.				
Branch On Greater Than Or Equal To Zero	BGEZ rs, offset Branches to the branch address if register rs is greater than 0.				
Branch On Less Than Zero And Link	BLTZAL rs, offset Stores the address of the instruction following the delay slot to register r31 (link register), and branches to the branch address if register rs is less than 0.				
Branch On Greater Than Or Equal To Zero And Link	BGEZAL rs, offset Stores the address of the instruction following the delay slot to register r31 (link register) and branches to the branch address if register rs is greater than 0.				

Table 3-16 Branch Instructions (Extended ISA)

Instruction	Format and Description	op	rs	rt	offset
Branch On Equal Likely	BEQL rs, rt, offset Branches to the branch address if registers rs and rt are equal. If the branch condition is not satisfied, the instruction in the branch delay slot is discarded.				
Branch On Not Equal Likely	BNEL rs, rt, offset Branches to the branch address if registers rs and rt are not equal. If the branch condition is not satisfied, the instruction in the branch delay slot is discarded.				
Branch On Less Than Or Equal To Zero Likely	BLEZL rs, offset Branches to the branch address if register rs is less than 0. If the branch condition is not satisfied, the instruction in the branch delay slot is discarded.				
Branch On Greater Than Zero Likely	BGTZL rs, offset Branches to the branch address if register rs is greater than 0. If the branch condition is not satisfied, the instruction in the branch delay slot is discarded.				

Instruction	Format and Description	REGIMM	rs	sub	offset
Branch On Less Than Zero Likely	BLTZL rs, offset Branches to the branch address if register rs is less than 0. If the branch condition is not satisfied, the instruction in the branch delay slot is discarded.				
Branch On Greater Than Or Equal To Zero Likely	BGEZL rs, offset Branches to the branch address if register rs is greater than 0. If the branch condition is not satisfied, the instruction in the branch delay slot is discarded.				
Branch On Less Than Zero And Link Likely	BLTZALL rs, offset Stores the address of the instruction following the delay slot to register r31 (link register). Branches to the branch address if register rs is less than 0. If the branch condition is not satisfied, the instruction in the branch delay slot is discarded.				
Branch On Greater Than Or Equal To Zero And Link Likely	BGEZALL rs, offset Stores the address of the instruction following the delay slot to register r31 (link register). Branches to the branch address if register rs is greater than 0. If the branch condition is not satisfied, the instruction in the branch delay slot is discarded.				

3.2.4 Special Instructions

The special instructions generate an exception by software. The instruction type is R-type (Syscall, Break). The trap instructions are invalid with the V_R3000 Series. All the other instructions are valid with all the V_R Series.

Table 3-17 Special Instructions

Instruction	Format and Description SPECIAL | rs | rt | rd | sa | funct
Synchronize	SYNC Completes the load/store instruction currently in the pipeline before the new load/store instruction is executed.
System Call	SYSCALL Generates a system call exception and transfers control to the exception processing program.
Breakpoint	BREAK Generates a breakpoint exception and transfers control to the exception processing program.

Table 3-18 Special Instructions (Extended ISA) (1/2)

Instruction	Format and Description SPECIAL | rs | rt | rd | sa | funct
Trap If Greater Than Or Equal	TGE rs, rt Compares registers rs and rt as signed integers. If register rs is greater than rt, generates an exception.
Trap If Greater Than Or Equal Unsigned	TGEU rs, rt Compares registers rs and rt as unsigned integers. If register rs is greater than rt, generates an exception.
Trap If Less Than	TLT rs, rt Compares registers rs and rt as signed integers. If register rs is less than rt, generates an exception.
Trap If Less Than Unsigned	TLTU rs, rt Compares registers rs and rt as unsigned integers. If register rs is less than rt, generates an exception.
Trap If Equal	TEQ rs, rt Generates an exception if registers rs and rt are equal.
Trap If Not Equal	TNE rs, rt Generates an exception if registers rs and rt are not equal.

Table 3-18 Special Instructions (Extended ISA) (2/2)

Instruction	Format and Description	REGIMM	rs	sub	immediate
Trap If Greater Than Or Equal Immediate	TGEI rs, immediate Compares the contents of register rs with 16-bit sign-extended immediate as signed integer. If rs contents are greater than the immediate, generates an exception.				
Trap If Greater Than Or Equal Immediate Unsigned	TGEIU rs, immediate Compares the contents of register rs with 16-bit zero-extended immediate as unsigned integer. If rs contents are greater than the immediate, generates an exception.				
Trap If Less Than Immediate	TLTI rs, immediate Compares the contents of register rs with 16-bit sign-extended immediate as signed integer. If rs contents are less than the immediate, generates an exception.				
Trap If Less Than Immediate Unsigned	TLTIU rs, immediate Compares the contents of register rs with 16-bit zero-extended immediate as unsigned integer. If rs contents are less than the immediate, generates an exception.				
Trap If Equal Immediate	TEQI rs, immediate Generates an exception if the contents of register rs are equal to immediate.				
Trap If Not Equal Immediate	TNEI rs, immediate Generates an exception if the contents of register rs are equal to immediate.				

3.2.5 Coprocessor Instructions

The coprocessor instructions are used to operate each coprocessor. The coprocessor load and store instructions are I-type. The format of the operation instruction of each coprocessor differs. Table 3-19 shows the coprocessor instructions valid for all the V_R Series. Table 3-20 lists the coprocessor instructions valid only with the V_R4000 which is defined as extended ISA.

Table 3-19 Coprocessor Instructions (1/2)

Instruction	Format and Description	op	base	rt	offset
Load Word To Coprocessor z	LWCz rt, offset (base) Sign-extends and adds offset to register base to generate an address. Loads the contents of the word specified by the address to the general purpose register rt of coprocessor z.				
Store Word From Coprocessor z	SWCz rt, offset (base) Sign-extends and adds offset to register base to generate an address. Stores the contents of the general purpose register rt of coprocessor z to the memory position specified by the address.				

Instruction	Format and Description	COPz	sub	rt	rd	0
Move To Coprocessor z	MTCz rt, rd Transfers the contents of CPU register rt to the general purpose register rd of coprocessor z.					
Move From Coprocessor z	MFCz rt, rd Transfers the contents of the general purpose register rd of coprocessor z to CPU register rt.					
Move Control To Coprocessor z	CTCz rt, rd Transfers the contents of CPU register rt to the coprocessor control register rd of coprocessor z.					
Move Control From Coprocessor z	CFCz rt, rd Transfers the contents of the coprocessor control register rd of coprocessor z to CPU register rt.					

Instruction	Format and Description	COPz	CO	cofun
Coprocessor z Operation	COPz cofun Coprocessor z executes an operation defined for each coprocessor. The status of the CPU is not changed by the operation of the coprocessor.			

Table 3-19 Coprocessor Instructions (2/2)

Instruction	Format and Description	COPz	BC	br	offset
Branch On Coprocessor z True	BCzT offset Shifts the 16-bit offset 2 bits to the left and sign-extends it to 32 bits. Adds the result to the address of the instruction in the delay slot to calculate the branch address. If the condition signal of coprocessor z is true, branches to the branch address, delayed by one instruction.				
Branch On Coprocessor z False	BCzF offset Shifts the 16-bit offset 2 bits to the left and sign-extends it to 32 bits. Adds the result to the address of the instruction in the delay slot to calculate the branch address. If the condition signal of coprocessor z is false, branches to the branch address, delayed by one instruction.				

Table 3-20 Coprocessor Instructions (Extended ISA) (1/2)

Instruction	Format and Description	COPz	sub	rt	rd	0
Doubleword Move To Coprocessor z	DMTCz rt, rd Transfers the contents of the general purpose register rt of the CPU to the general purpose register rd of coprocessor z.					
Doubleword Move From Coprocessor z	DMFCz rt, rd Transfers the contents of the general purpose register rd of coprocessor z to the general purpose register rt of the CPU.					

Instruction	Format and Description	op	base	rt	offset
Load Doubleword To Coprocessor z	LDCz rt, offset (base) Sign-extends and adds offset to register base to generate an address. Loads the contents of the doubleword specified by the address to the general purpose register (rt if FR = 1 and rt and rt+1 if FR = 0) of coprocessor z.				
Store Doubleword From Coprocessor z	SDCz rt, offset (base) Sign-extends and adds offset to register base to generate an address. Stores the contents of the doubleword of the general purpose register (rt if FR = 1 and rt and rt+1 if FR = 0) of coprocessor z to the memory position specified by the address.				

Table 3-20 Coprocessor Instructions (Extended ISA) (2/2)

Instruction	Format and Description	COPz	BC	br	offset
Branch On Coprocessor z True Likely	BCzTL offset Shifts the 16-bit offset 2 bits to the left and sign-extends it. Adds the result to the address of the instruction in the delay slot to calculate the branch address. If the condition signal of coprocessor z is true, branches to the branch address, delayed by one instruction. If the branch condition is not satisfied, the instruction in the branch delay slot is discarded.				
Branch On Coprocessor z False Likely	BCzFL offset Shifts the 16-bit offset 2 bits to the left and sign-extends it. Adds the result to the address of the instruction in the delay slot to calculate the branch address. If the condition signal of coprocessor z is false, branches to the branch address, delayed by one instruction. If the branch condition is not satisfied, the instruction in the branch delay slot is discarded.				

3.2.6 System Control Coprocessor (CP0) Instructions

The system control coprocessor (CP0) instructions execute operations to the CP0 register to control the memory of the processor and to perform exception processing.

Table 3-21 System Control Coprocessor (CP0) Instructions (1/2)

Instruction	Format and Description: COP0 | sub | rt | rd | 0
Move To System Control Coprocessor	MTC0 rt, rd Loads the contents of the word of the general purpose register rt of the CPU to the general purpose register rd of CP0.
Move From System Control Coprocessor	MFC0 rt, rd Loads the contents of the word of the general purpose register rd of CP0 to the general purpose register rt of the CPU.
Doubleword Move To System Control Coprocessor	DMTC0 rt, rd Loads the contents of the doubleword of the general purpose register rt of the CPU to the general purpose register rd of CP0.
Doubleword Move From System Control Coprocessor	DMFC0 rt, rd Loads the contents of the doubleword of the general purpose register rd of CP0 to the general purpose register rt of the CPU.

Instruction	Format and Description: COP0 | CO | funct
Read Indexed TLB Entry	TLBR Loads the TLB entry indicated by the index register to the entry Hi, entry Lo0, entry Lo1, and page mask registers.
Write Indexed TLB Entry	TLBWI Loads the contents of the entry Hi, entry Lo0, entry Lo1, and page mask registers to the TLB entry indicated by the index register.
Write Random TLB Entry	TLBWR Loads the contents of the entry Hi, entry Lo0, entry Lo1, and page mask registers to the TLB entry indicated by the random register.
Probe TLB For Matching Entry	TLBP Loads the address of the TLB entry coinciding with the contents of the entry Hi register to the index register.
Return From Exception	ERET Returns from an exception, interrupt, or error trap.

Table 3-21 System Control Coprocessor (CP0) Instructions (2/2)

Instruction	Format and Description	CACHE	base	op	offset
Cache Operation	Cache op, offset (base) Sign-extends the 16-bit offset to 32 bits and adds it to register base to generate a virtual address. The virtual address is converted into a physical address by using the TLB, and a cache operation indicated by a 5-bit sub op code is executed to that address.				

[MEMO]

Pipeline

4

This chapter describes the operation of the V_R4300 processor pipeline.

4.1 General

The V_R4300 uses a 5-stage pipeline. The pipeline is usually controlled by the pipeline clock that is determined by the value of the DivMode(1:0)* pins. This pipeline clock is called PClock and one cycle of it is called PCycle. Each stage of the pipeline is executed in 1 PCycle. The PCycle has two stages, Φ1 and Φ2, as shown in Figure 4-1. Therefore, at least 5 PCycles are required to execute an instruction. If the necessary data is not in the cache and must be fetched from the main memory, more cycles are necessary. When the pipeline flows smoothly, five instructions are executed simultaneously.

** In V_R4300 and V_R4305. In V_R4310, DivMode(2:0).*

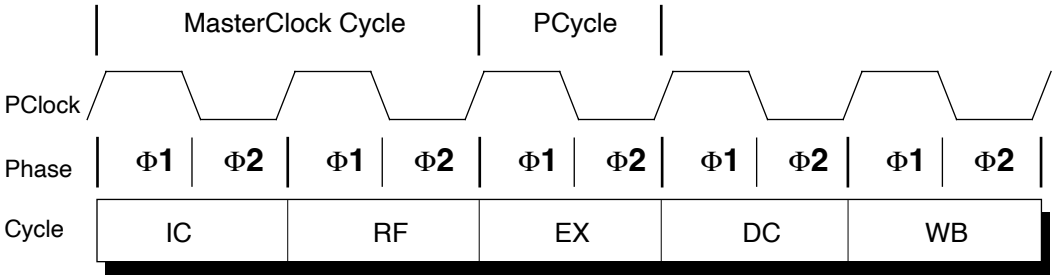


Figure 4-1 Pipeline Stages

The five pipeline stages are:

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

Figure 4-2 outlines the pipeline. The horizontal rows in this figure indicate the execution processes of instructions, and the vertical columns indicate the five processes executed at the same time.

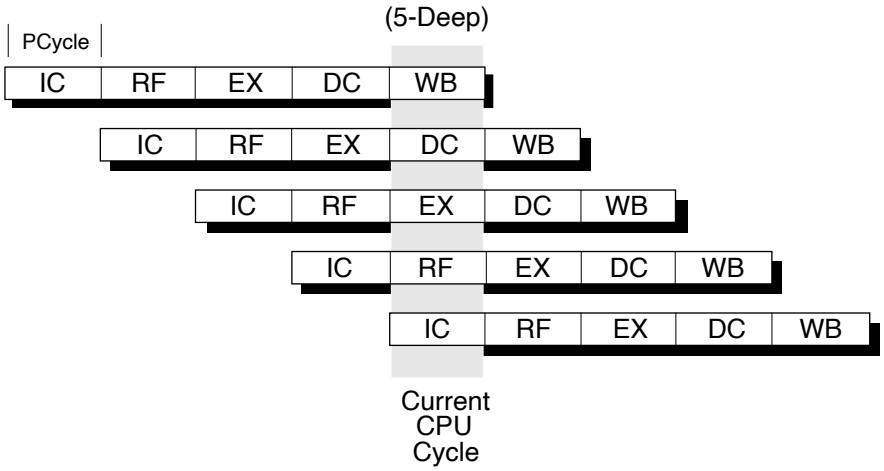


Figure 4-2 Instruction Execution in the Pipeline

4.1.1 Pipeline Operations

Figure 4-3 shows the operations that can occur during each pipeline stage; Table 4-1 describes these pipeline activities.

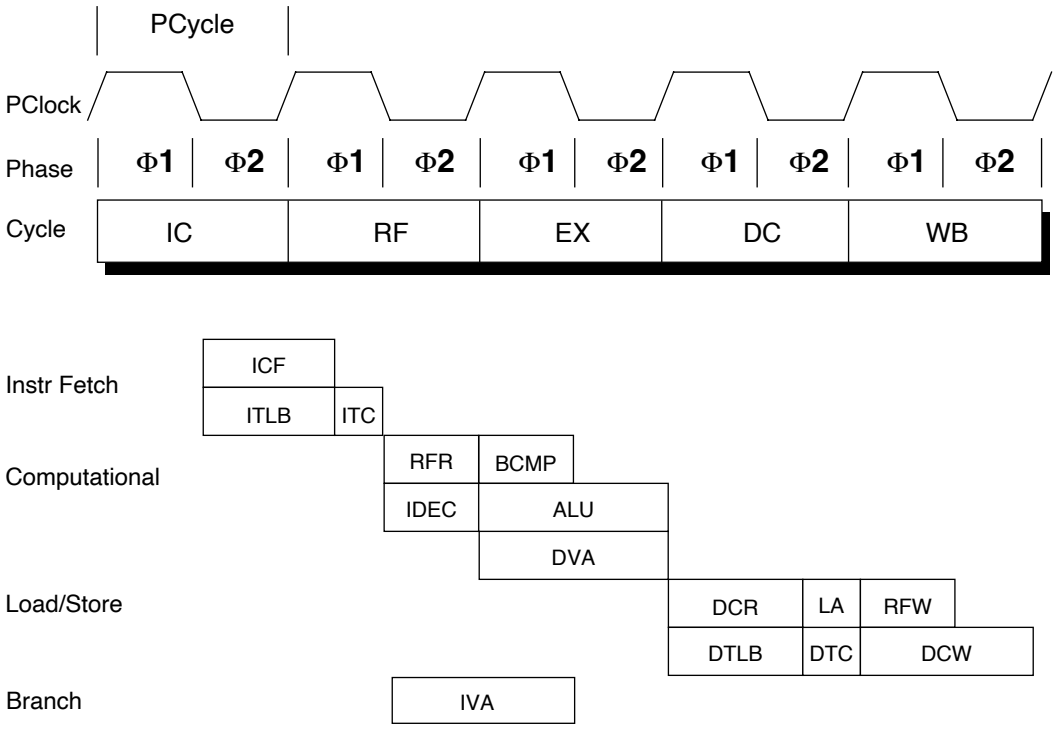


Figure 4-3 Pipeline Operations

Table 4-1 Description of Pipeline Showing Stage in Which Operations Commence

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

4.2 Branch Delay

The pipeline of the V_R4300 generates a branch delay of one cycle in the following cases:

- When a target address is calculated with a jump instruction
- When the branch condition of a branch instruction is satisfied and a target address is calculated

The instruction address generated in the EX stage of a jump/branch instruction cannot be used until the IC stage of the instruction to be executed after the next instruction.

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

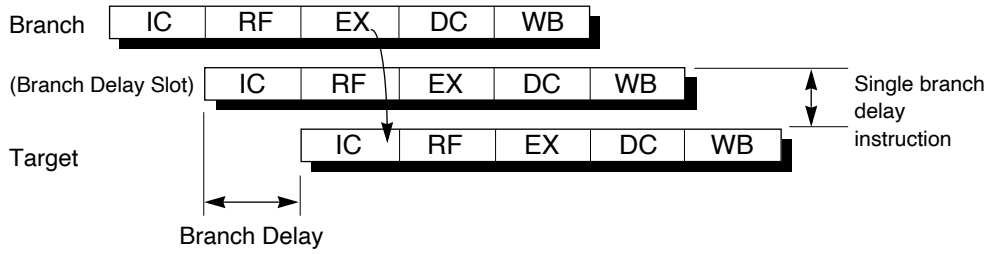


Figure 4-4 Branch Delay

4.3 Load Delay

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

In the V_R4300 processor, the instruction immediately following a load instruction can use the contents of the loaded register, however in such cases hardware interlocks insert additional delay cycles. Consequently, scheduling load delay slots can be desirable, both for performance and V_R-Series processor compatibility.

4.4 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.

Floating-point instructions are executed in the pipeline in the same manner as multicycle integer instructions.

Add Instruction

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

IC stage — In phase 2 of the IC stage, the fourteen low-order bits of the virtual address are used to address the instruction cache. The two high-order bits of this virtual address select one of four instruction cache banks, and the remaining bits address the selected bank. The ITLB selects the page.

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

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

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

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

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

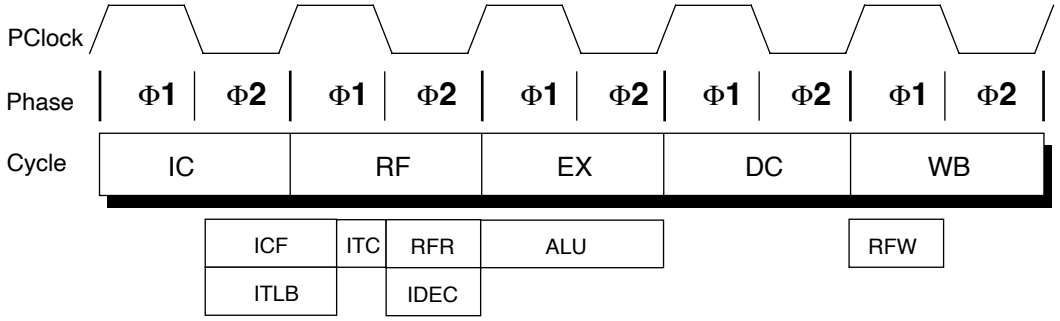


Figure 4-5 Add Instruction Pipeline Operations

Jump and Link Register Instruction

```
JALR rd,rs
```

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

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

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

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

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

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

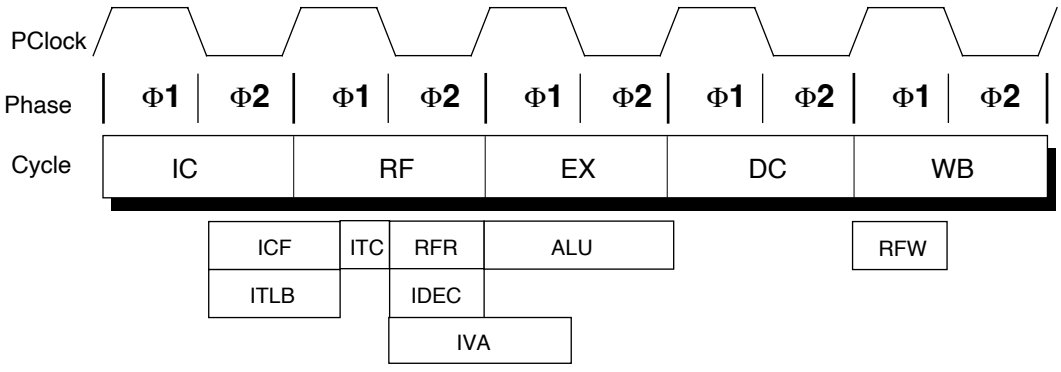


Figure 4-6 Jump and Link Register Instruction Pipeline Operations

Branch on Equal Instruction

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

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

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

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

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

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

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

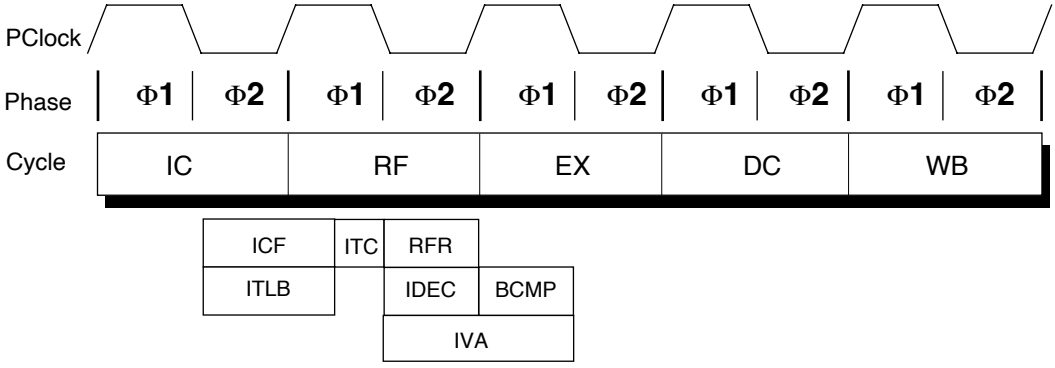


Figure 4-7 Branch on Equal Instruction Pipeline Operations

Trap if Less Than Instruction

```
TLT rs,rt
```

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

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

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

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

DC stage — The sign bits of operands and of the ALU output latch are checked to determine if a *less than* condition is true. If this condition is true, a Trap Exception occurs. This, as with all pipeline exceptions, implies a 2-cycle stall. The *PC* register is loaded with the value of the exception vector and instructions following in previous pipeline stages are killed.

WB stage — The exception code is set in the ExCode field in the *cause* register if the *less than* condition was met in the DC stage. The PC value of this instruction is stored in the *EPC* register and *BD* bit are updated appropriately according to the contents of 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.

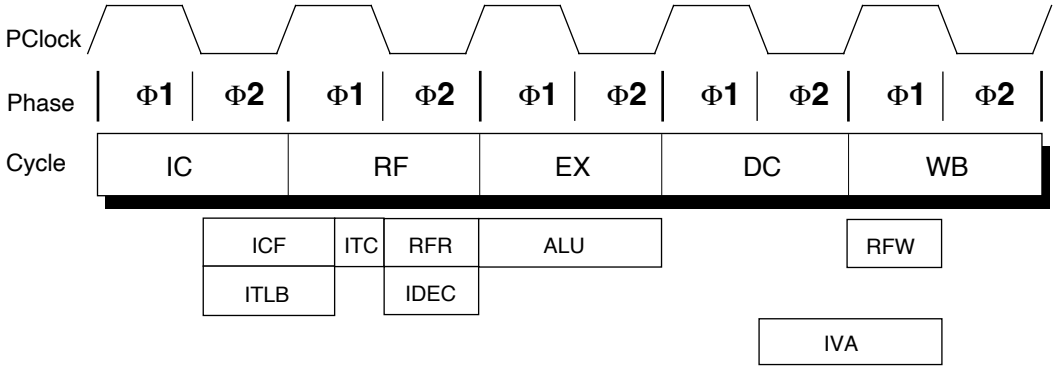


Figure 4-8 Trap if Less Than Instruction Pipeline Operations

Load Word Instruction

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

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

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

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

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

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

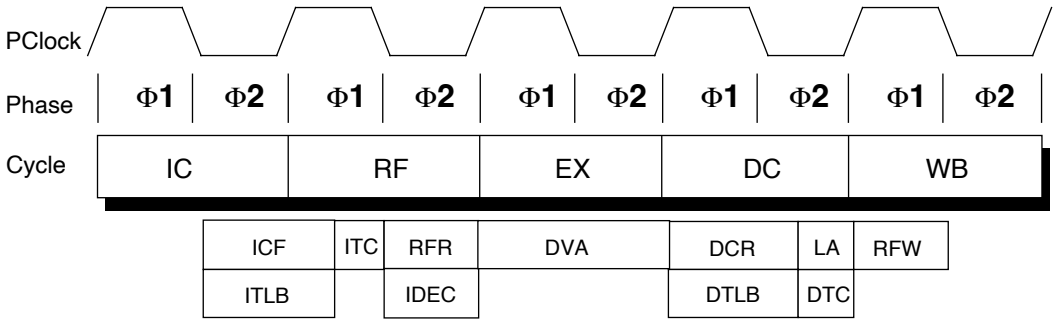


Figure 4-9 Load Word Instruction Pipeline Operations

Store Word Instruction

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

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

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

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

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

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

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

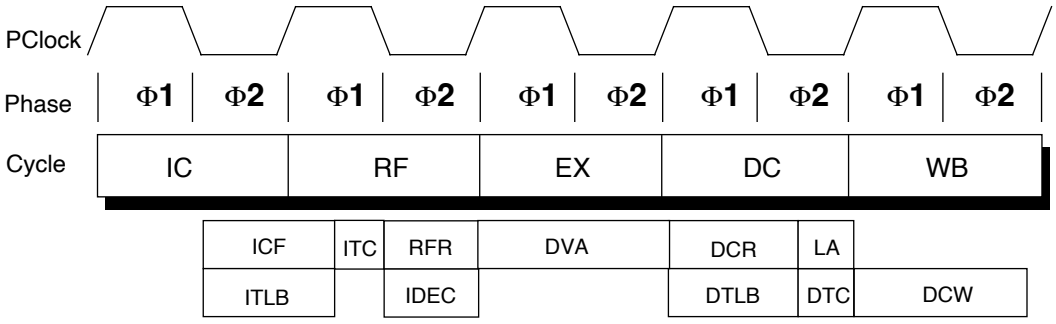


Figure 4-10 Store Word Instruction Pipeline Operations

4.5 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 4-11, all interlock and exception conditions are collectively referred to as faults.

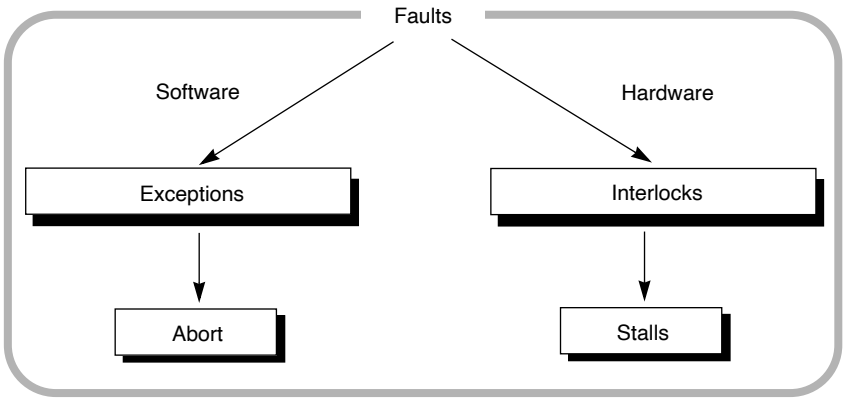
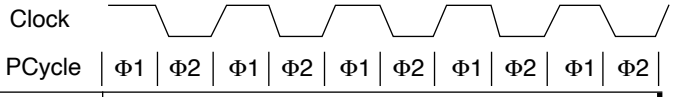


Figure 4-11 Interlocks, Exceptions, and Faults

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 4-12. For instance, an LDI Interlock is raised in the execution (EX) stage.

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



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

Remark The conditions of the exceptions are shown starting from the exception with the highest priority.

Figure 4-12 Correspondence of Pipeline Stage to Interlock and Exception Condition

Table 4-2 Description of Pipeline Exceptions

Exception	Description
IADE	Instruction Address Error Exception
ITLB	Instruction TLB Exception
IBE	Instruction Bus Error Exception
SYSC	SYSCALL Instruction Exception
BRPT	Breakpoint Instruction Exception
CPU	Coprocessor Unusable Exception
RSVD	Reserved Instruction Exception
RST	External Reset Exception
NMI	External NMI Exception
OVFL	Integer Overflow Exception
TRAP	TRAP Instruction Exception
FPE	Floating-point Exception
DADE	Data Address Error Exception
DTLB	Data TLB Exception
WAT	Reference to Watch Address Exception
INTR	Interrupt Exception
DBE	Data Bus Error Exception

Table 4-3 Description of Pipeline Interlocks

Interlock	Description
ITM	Instruction TLB Miss
ICB	Instruction Cache Busy
LDI	Load Interlock
MCI	Multi-cycle Interlock
DCM	Data Cache Miss
DCB	Data Cache Busy
COp	Cache Op
CP0I	CP0 Bypass Interlock

4.6 Pipeline Interlocks and Exceptions

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

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

Pipeline interlocks and pipeline exceptions are described in the following section. The exceptions themselves are described in **Chapter 6 Exception Processing**.

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

4.6.1 Pipeline Interlocks

When an interlock condition occurs, the pipeline stalls and remains stalled until the interlock is corrected. Should pipeline stall requests from different stages arise simultaneously, the Pipeline Control Unit prioritizes the stall requests. For instance, a stall request from the DC stage is always allowed to be resolved before a simultaneous RF-stage stall request, since both may require the same resource (TLB, memory) to be resolved. The EX stage is allowed to stall in order to complete a multicycle instruction as long as there is no load dependency between itself (the EX stage) and the DC stage. Interlock conditions for each pipeline stage are shown in Figure 4-12 and described in Table 4-3.

The remainder of this section describes in detail the following pipeline interlocks:

- Instruction TLB Miss (ITM)
- Instruction Cache Busy (ICB)
- Load Interlock (LDI)
- Multicycle Instruction Interlock (MCI)
- Data Cache Miss (DCM)
- Data Cache Busy (DCB)
- Cache Operation (COp)
- CP0 Bypass Interlock (CP0I)

4.6.2 Instruction TLB Miss (ITM)

A pipeline stall due to an Instruction TLB Miss occurs when the virtual address of the next instruction to be fetched is not found in the instruction micro-TLB (ITLB).

The pipeline stalls when the micro-TLB miss is detected in the RF stage, whereupon the pipeline controller notifies the micro-TLB to proceed in servicing the stall. The pipeline starts running again when the micro-TLB has been updated from the JTLB.

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

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

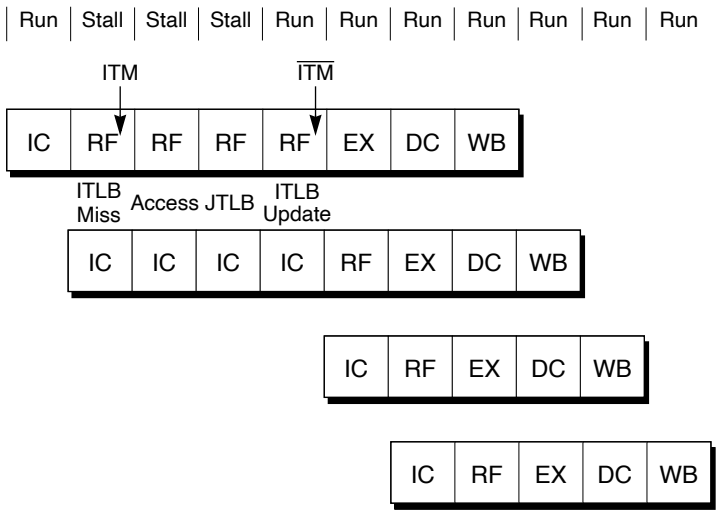


Figure 4-13 Instruction TLB Miss Interlock

4.6.3 Instruction Cache Busy (ICB)

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

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

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

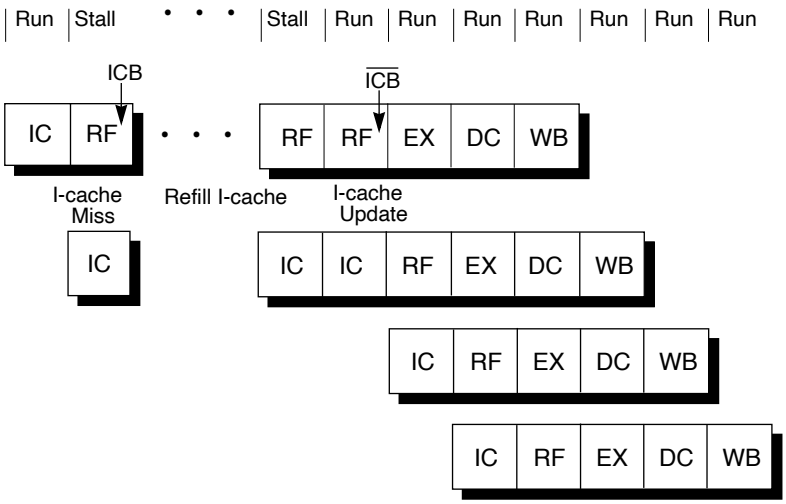


Figure 4-14 Example of an Instruction Cache Busy Interlock

4.6.4 Multicycle Instruction Interlock (MCI)

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

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

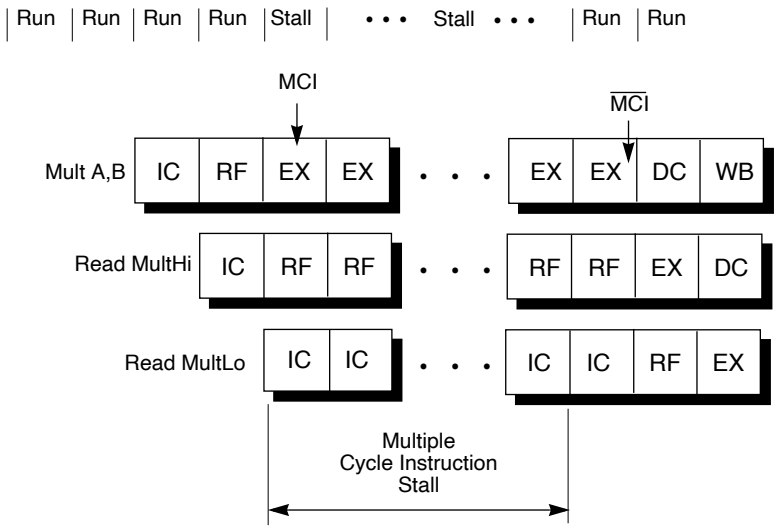


Figure 4-15 Example of a Multicycle Instruction Interlock

4.6.5 Load Interlock (LDI)

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

The pipeline begins running again when the clock after the target of the load is read from the data cache (in the DC stage of the "Load B" instruction in Figure 4-16).

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

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

This case is illustrated in Figure 4-17.

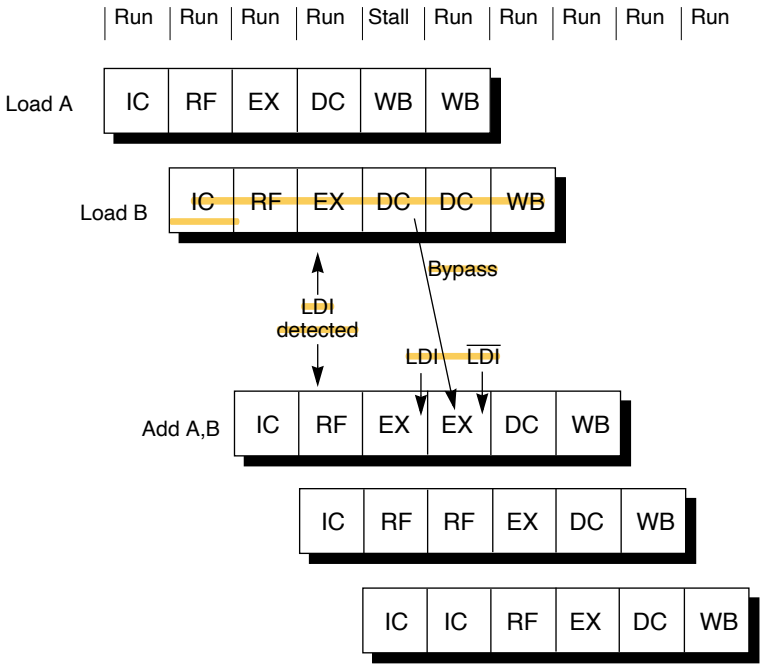


Figure 4-16 Example of a Load Interlock

4.6.6 Data Cache Miss (DCM)

If a data cache miss occurs in the DC stage, the pipeline stalls for 1 PCycle in which the miss is detected. The pipeline stalls regardless of whether the load or store instruction is executed. The data cache busy (explained next) continues stalling until a new cache line is read.

When a requested word data has been read from the cache, the pipeline begins running again.

Figure 4-17 illustrates DCM.

4.6.7 Data Cache Busy (DCB)

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

- If the instruction immediately after a store instruction requires use of the data cache then the pipeline is stalled in its DC stage while the store writes the data to the cache during its WB stage. ~~On a cache store hit the pipeline only stalls for one PClock while the data is written to the data cache.~~ On a cache store miss the pipeline stalls with the store in the DC stage until the cache line has been updated. Once the line has been updated, the pipeline restarts and moves the store instruction into the WB stage. If the instruction following the "store" (i.e. the instruction currently in the DC stage) also requires access to the data cache, the pipeline will then stall for one PCycle while the store data is being written to the cache.
- When a miss occurs on a load, the data cache signals it is busy while it fetches the missed data word from external memory. Refer to **Figure 4-17**.

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

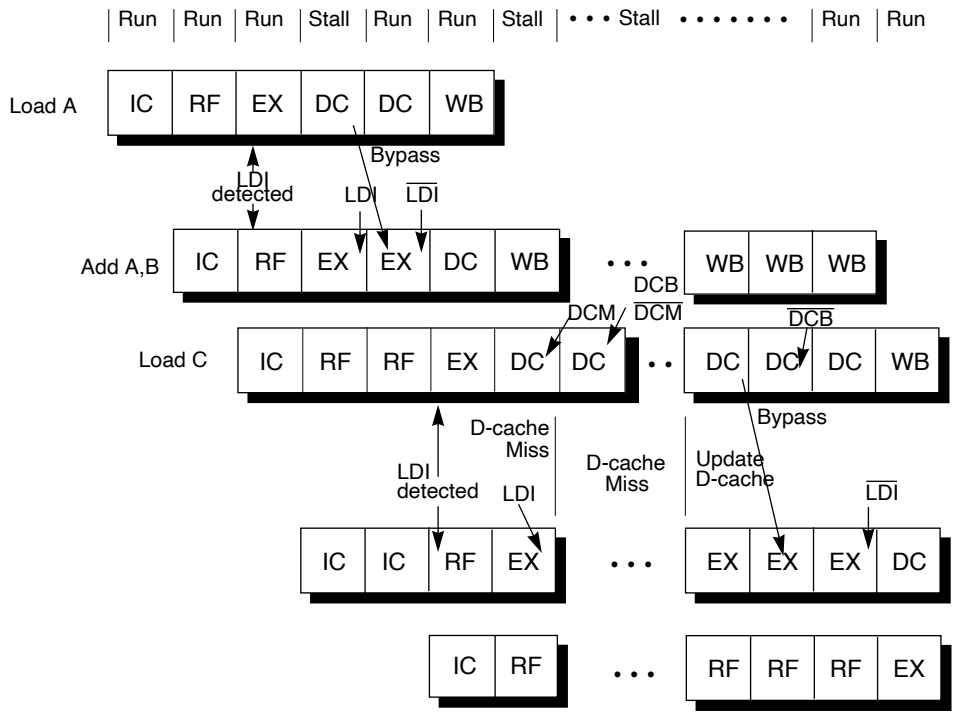


Figure 4-17 Example of a Data Cache Miss Followed by a Load Interlock

4.6.8 CACHE Operation (COp)

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

- When an instruction cache operation instruction enters the DC stage, the instruction cache operation continues to be serviced while the pipeline stalls. The pipeline begins running again when the instruction cache operation is complete, allowing the next instruction fetch to proceed.
- When the data cache operation instruction requiring an operation of 2 PCycles of the data cache has entered the DC stage.

4.6.9 Coprocessor 0 Bypass Interlock (CP0I)

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

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

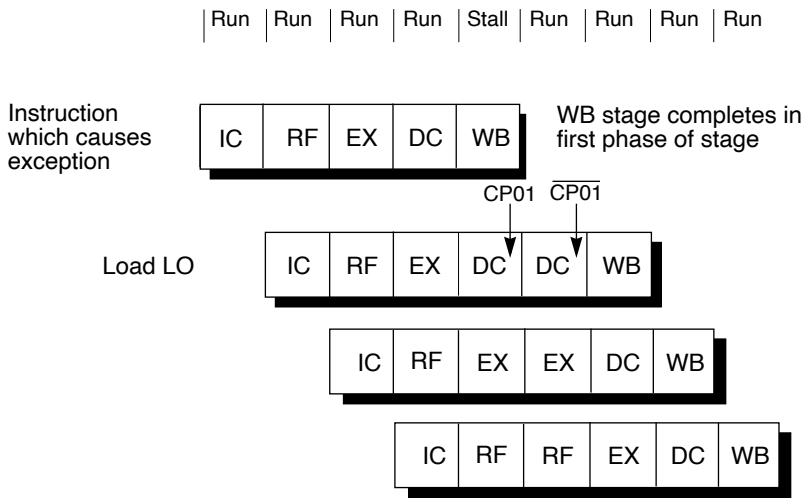


Figure 4-18 Example of a Coprocessor 0 Bypass Interlock (CP0I)

4.7 Pipeline Exceptions

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

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

Exception conditions for each pipeline stage are shown in Figure 4-12 and described in Table 4-2.

Exceptions can split into two groups:

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

Exceptions are logically precise.

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

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

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

4.7.2 Instruction-Dependent Exceptions

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

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

4.7.3 Interactions between Interlocks and Exceptions

With the V_R4300, the processing of the EX and RF stages can be continued while the pipeline stalls. The interaction between interlocking of the two stages and exceptions is relatively simple.

Interaction between EX and RF Stages

The EX exception occurs only when an instruction that causes the EX exception has entered a pipeline stage. Because the RF interlock solving processing has not yet been started at this time, the EX exception takes precedence because of the stall request from the RF stage. Interactions in various cases are described next.

- **When EX exception is stalled by DC interlock**
  The EX exception takes precedence over the RF stall request. This is because the RF interlock is not solved during the DC stall period.
- **If instruction cache busy and multi-cycle instruction interlock take place simultaneously**
  Both the RF and EX stages solve the respective interlocks.
  The cause that has generated a floating-point exception is detected before the instruction cache busy (ICB) stall ends, but the exception occurs after execution has entered the DC stage. Therefore, the exception condition is retained in the EX stage until the RF interlock is solved, and the related stage is deleted.
- **If exception from EX stage and RF interlock take place simultaneously**
  The EX exception takes precedence. This is because the instruction that has caused the RF interlock is canceled and no request is issued to the external memory.

Interaction between RF and DC Stages

If a stall request is made at the same time in the RF and DC stages, the pipeline controller gives the priority to the processing of the DC stage. In other words, the RF stall processing is started after the DC stall has been solved. This is because the same resources (such as the system interface and TLB) are necessary for solving the RF interlock and DC interlock.

4.7.4 Exception and Interlock Priorities

The priority for processing exceptions and interlocks within the same clock cycle is listed below. Exception and interlock requests from the WB stage always have priority over exception and interlock requests from the DC stage. Exception and interlock requests from the DC stage always have priority over exception and interlock requests from the EX stage. EX-stage exception and interlock requests in turn always have priority over any exception and interlock requests from the RF stage.

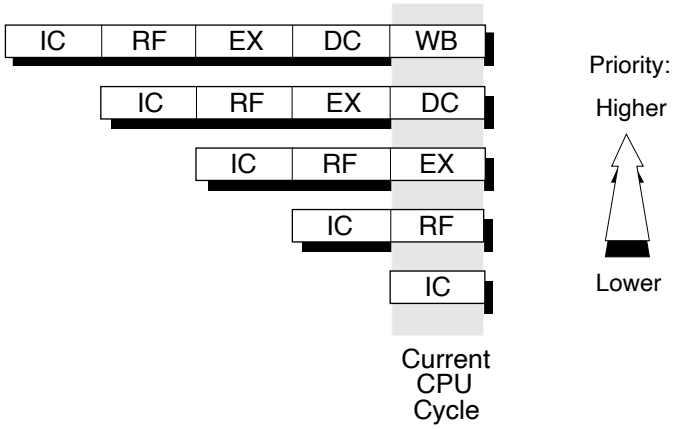


Figure 4-19 Execution and Interlock Priorities

In the case of multiple exception requests from the same pipeline stage, the highest-priority exception is processed first. The priority of the instruction-dependent exceptions and interlocks are shown in the following sections.

4.7.5 WB-Stage Interlock and Exception Priorities

Because there is only the following one exception or interlock in the WB stage, there is no priority.

- CP0 Bypass interlock

4.7.6 DC-Stage Interlock and Exception Priorities

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

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

4.7.7 EX-Stage Interlock and Exception Priorities

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

- System Call exception
- Breakpoint exception
- Coprocessor Unusable exception
- Reserved Instruction exception
- Load interlock
- Multicycle Instruction interlock

4.7.8 RF-Stage Interlock and Exception Priorities

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

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

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

Individual exceptions are described in detail in **Chapter 6 Exception Processing**.

4.7.9 Bypassing

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

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

The Bypass Control Unit also controls the source and destination register addresses supplied from the register file.

4.8 Code Compatibility

The V_R4300 can execute any programs which can be executed on the V_R3000 series and V_R4000 series*, but the reverse may not necessarily be true. Standard MIPS compilers produce code which will run on both. When hand-coding assembly code, it is strongly advised to maintain compatibility with the V_R Series. For more information, refer to the each product's user's manuals.

* The instruction set on the V_R4100 differs partially from the other products. (For example, FPU instructions are not supported.)

4.9 Write Buffer

The V_R4300 processor contains an on-chip write buffer, used as a temporary data storage for outgoing data. The write buffer stores one doubleword (8 bytes) of data for each PCycle, and can buffer a total of eight words (32 bytes) of data, equal to the data cache line size. When storing data, therefore, all the data lengths can be used.

The write buffer can store any data as long as it has a vacancy.

The format of the write buffer is shown below.

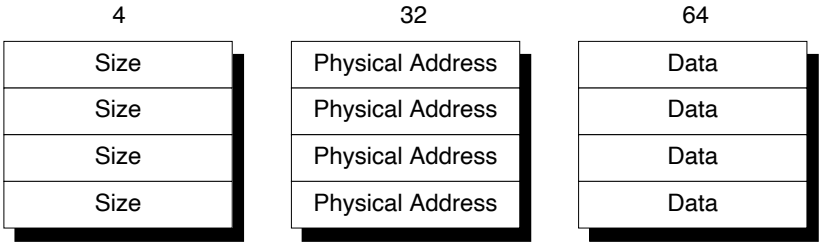


Figure 4-20 Write Buffer Format

The write buffer can store the following:

- Four 32-bit physical addresses
- 4-bit size area indicating four types of transfer data size
- Data up to 4 doublewords

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

During either a load miss or a store miss to a cache line in the dirty state (refer to **Chapter 11 Cache Memory** for a description of cache line states), dirty data is stored in this buffer until the requested data is returned from the external interface. The processor pipeline continues to run while the write buffer waits (for a response from the external interface) to empty its contents to the external interface/memory.

If the processor executes a load or store instruction requiring external resources when the write buffer is full, the pipeline is stalled until the write buffer has a space for the data to be stored.

Memory Management System

5

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

This chapter describes the operation of the TLB, those System Control Coprocessor (CP0) registers that provide the software interface to the TLB and the memory mapping method that translates the virtual address to the physical address.

5.1 Translation Lookaside Buffer (TLB)

A virtual address is converted into a physical address by using the internal TLB*. The internal TLB is a full-associative memory having 32 entries, and one entry is mapped with an odd and even numbers in pairs. The size of these pages can be 4K, 16K, 64K, 256K, 1M, 4M, or 16M, and can be specified for each entry. When a virtual address is given, each TLB entry checks the 32 entries whether the virtual address coincides with the virtual address appended with the ASID area stored to the Entry Hi register.

If the addresses coincide (if a hit occurs), a physical address is generated from the physical address in the TLB and an offset.

If the addresses do not coincide (if a miss occurs), an exception occurs, and the TLB entry is written by software from a page table on the memory. The software either writes the TLB entry over the entry selected by the index register, or writes it to a random entry indicated by the random register.

If there are two or more TLB entries that coincide, the TLB operation is not correctly executed. In this case, the TLB-Shutdown (TS) bit of the status register is set to 1, and then the TLB cannot be used.

5.2 Memory Management System Architecture

The memory management system expands the address space of the CPU by converting a large virtual memory space into physical addresses.

The physical address space of the V_R4300 is 4 GB with 32-bit addresses used. A virtual address is 32 bits wide in the case of the 32-bit mode, and the maximum user area is 2 GB (2^{31}). In the case of the 64-bit mode, the address is 64 bits wide, and the maximum user area is 1 TB (2^{40}). For the TLB entry format in each mode, refer to **5.3.1**.

The virtual address is expanded by the address space ID (ASID) (refer to **Figures 5-2** and **5-3**). ASID decreases the number of times of TLB flash when the context is switched. The ASID area is 8 bits wide and is in the entry Hi register of CP0. The global bit (G) is in the entry Lo0 and entry Lo1 registers.

* 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.

1. Virtual address (VA) represented by the virtual page number (VPN, high-order bit of the address) is compared with indicated area in TLB.
2. If there is a match, the page frame number (PFN) representing the high-order 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.

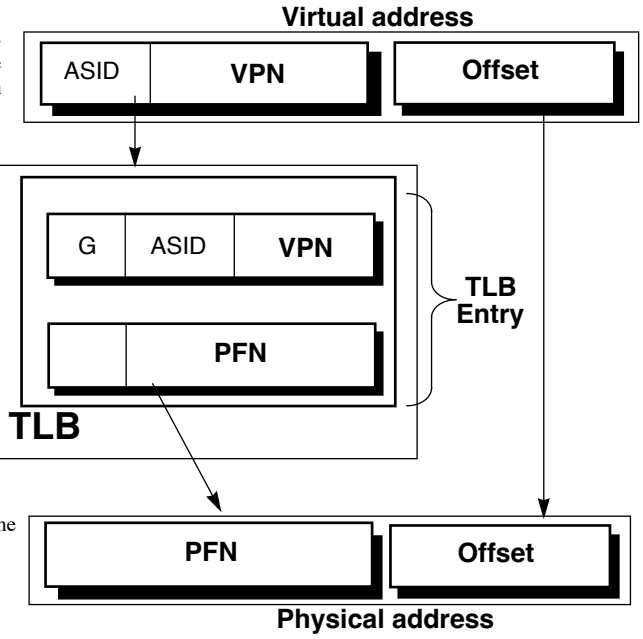


Figure 5-1 Overview of a Virtual-to-Physical Address Translation

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 reference a page table of virtual/physical addresses in memory and to write its contents to the TLB.

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. The lower bits of the virtual address are output as is.

For details, refer to **5.4.9 Virtual-to-Physical Address Translation Process**.

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

32-bit Mode Address Translation

Figure 5-2 shows the virtual-to-physical-address translation of a 32-bit mode address. This figure illustrates the two of seven possible page sizes: a 4 KB page (12 bits) and a 16 MB page (24 bits).

- The top portion of Figure 5-2 shows a virtual address with a 12-bit, or 4 KB, page size, labelled *Offset*. The remaining 20 bits of the address excluding ASID represent the VPN, and index the 1M-entry page table.
- The bottom portion of Figure 5-2 shows a virtual address with a 24-bit, or 16 MB, page size, labelled *Offset*. The remaining 8 bits of the address excluding ASID represent the VPN, and index the 256-entry page table.

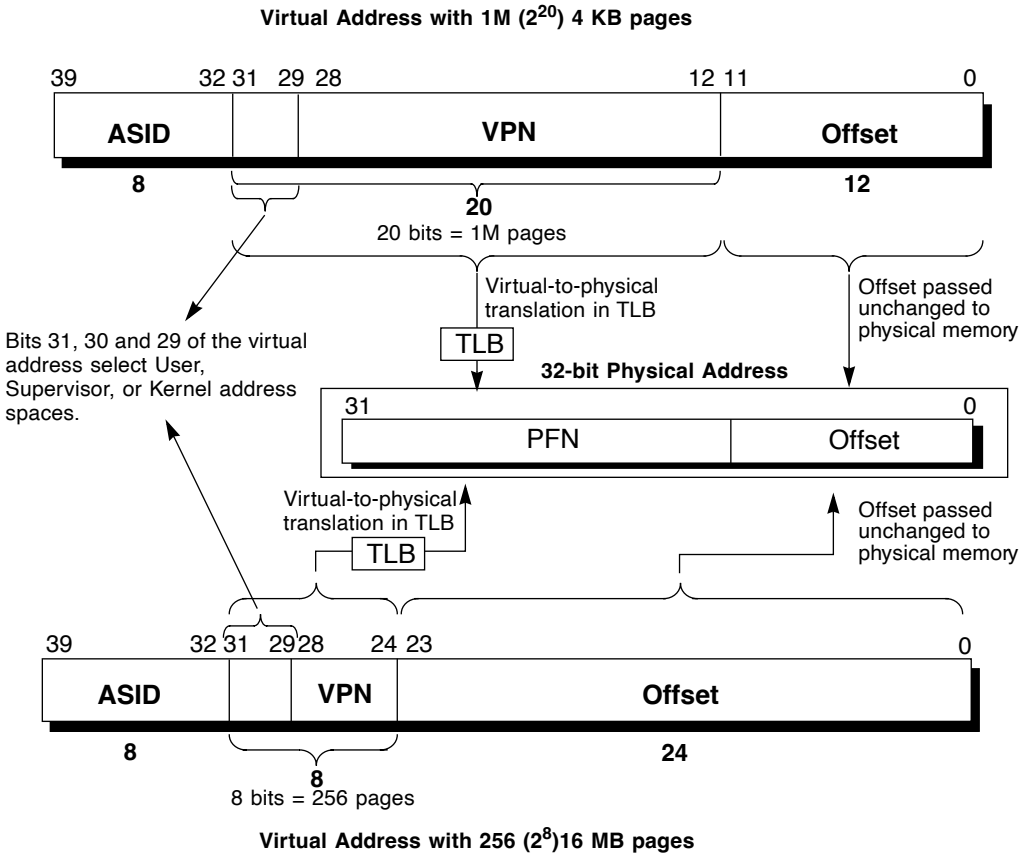


Figure 5-2 32-Bit Mode Virtual Address Translation

64-bit Mode Address Translation

Figure 5-3 shows the virtual-to-physical-address translation of a 64-bit mode address. This figure illustrates the two of seven possible page sizes: a 4 KB page (12 bits) and a 16 MB page (24 bits).

- The top portion of Figure 5-3 shows a virtual address with a 12-bit, or 4 KB, page size, labelled *Offset*. The remaining 28 bits of the address excluding ASID represent the VPN, and index the 256M-entry page table.
- The bottom portion of Figure 5-3 shows a virtual address with a 24-bit, or 16 MB, page size, labelled *Offset*. The remaining 16 bits of the address excluding ASID represent the VPN, and index the 64K-entry page table.

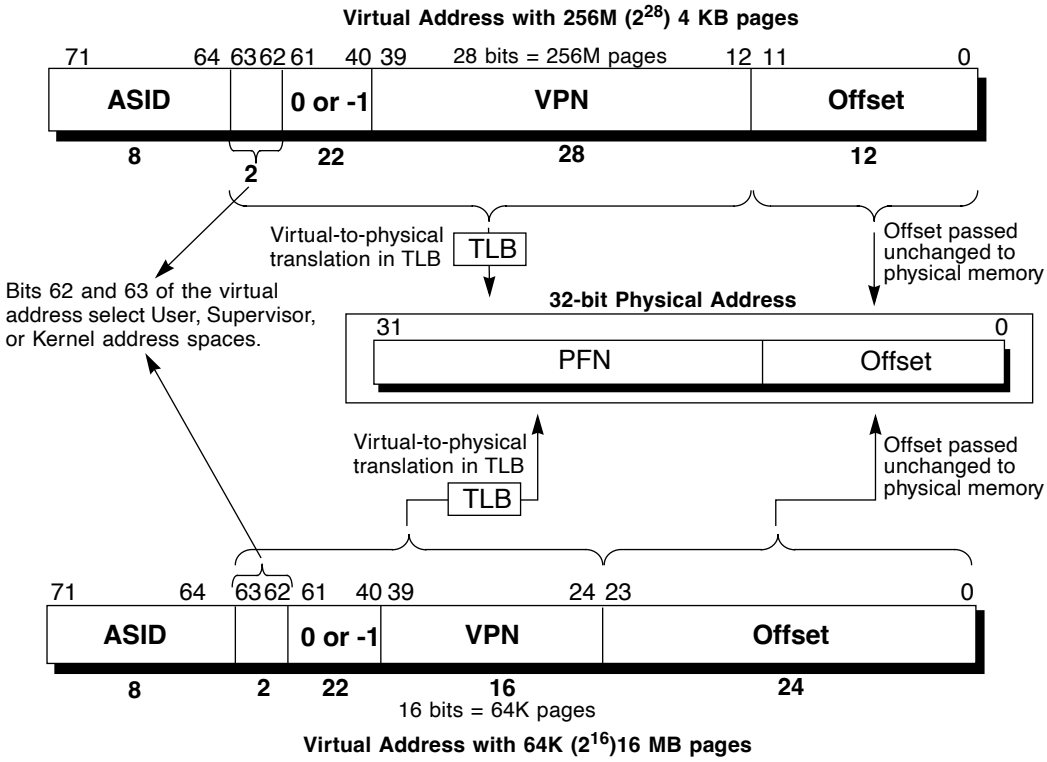


Figure 5-3 64-Bit Mode Virtual Address Translation

5.2.1 Operating Modes

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

- User mode
- Supervisor mode
- Kernel mode

The User mode and Kernel mode are common to all the V_R Series members. Generally, the operating system is executed in the Kernel mode, and the application program is executed in the user mode. The V_R4000 series is provided with a third mode. This mode, called the supervisor mode, is intermediate between the User and Kernel modes, and is used to organize a high security system.

If an exception occurs, the CPU enters the Kernel mode, and remains in this mode until an exception return instruction (ERET) is executed. The ERET instruction restores the mode in which the processor was operating before the occurrence of the exception.

5.2.2 Virtual Addressing in User Mode

In the single-user mode, a virtual address space (useg) of 2 GB (2^{31} bytes) can be used in the 32-bit mode, and a 1 TB (2^{40} bytes) virtual address space (xuseg) can be used in the 64-bit mode. As shown in Figures 5-2 and 5-3, each virtual address is expanded to a separate virtual address by an 8-bit address space ID (ASID) for up to 256 user processes. The system allocates each process with an ASID to retain the contents of the TLB even when it has switched the context. useg and xuseg are referenced via TLB. Whether the cache can be used or not is determined for each page by the TLB entry (the C bit of the TLB entry determines whether the cache can be used).

The user segment starts from address 0 and the currently valid user process resides in useg (in the 32-bit mode) or xuseg (in the 64-bit mode).

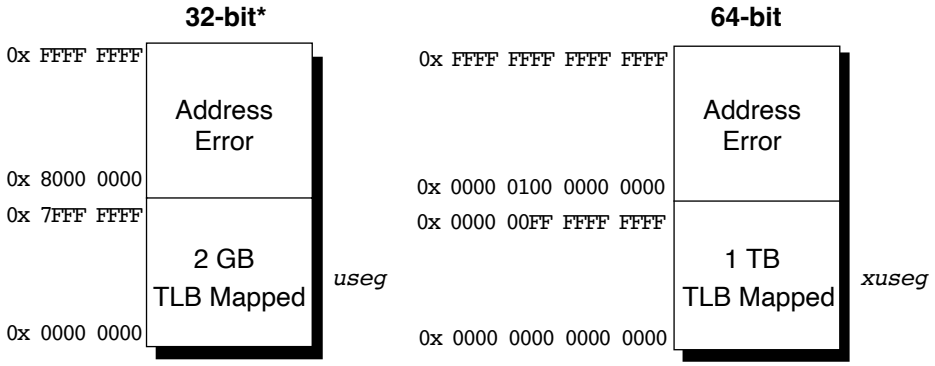
The V_R4300 operates in the user mode when the values of the bits in the *Status* register is as follows:

- *KSU* bits = 10
- *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:

- UX = 0: Selects 32-bit useg
  TLB miss is processed by a 32-bit TLB miss exception handler.
- UX = 1: Selects 64-bit xuseg
  TLB miss is processed by a 64-bit XTLB miss exception handler.

Table 5-1 lists the characteristics of the two user mode segments, *useg* and *xuseg*.



* The V_R4300 internally uses 64-bit addresses. In the Kernel mode, the processor saves and restores each register to initialize the register before switching the context. A 32-bit value is used as an address, with bit 31 sign-extended to bits 32 through 63, in the 32-bit mode.
Usually, the program in the 32-bit mode does not generate invalid addresses. If the context is switched and the processor enters the Kernel mode, a value other than the 32-bit address previously sign-extended may be stored to a 64-bit register. In this case, the program in the user mode may generate invalid addresses.

Figure 5-4 User Mode Virtual Address Space

Table 5-1 32-Bit and 64-Bit User Mode Segments

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

useg (32-bit mode)

When the *UX* bit of the *Status* register is 0 and the most significant bit of the virtual address is 0, this virtual address space is referred to as useg. If an attempt is made to reference an address whose most significant bit is 1, an address error exception occurs (refer to **Chapter 6 Exception Processing**).

xuseg (64-bit mode)

If the *UX* bit of the *Status* register is 1 and the bits (63:40) of the virtual address are all 0, the virtual address space is referred to as xuseg. A user address space of 1 TB (2^{40} bytes) can be used. If an attempt is made to reference an address that has 1 in bits (63:40), an address error exception occurs (refer to **Chapter 6 Exception Processing**).

5.2.3 Virtual Addressing in Supervisor Mode

The supervisor mode shown in Figure 5-5 is intended for hierarchical execution of the operating system. In the Kernel mode, the Kernel operating system in the highest hierarchy is executed, and the other operating systems are executed in the supervisor mode.

Referencing suseg, sseg, xsuseg, xsseg, and csseg (i.e., all spaces) is carried out via TLB. Whether the cache can be used or not is determined by the TLB entry of each page (the C bit of the TLB entry determines whether the cache can be used).

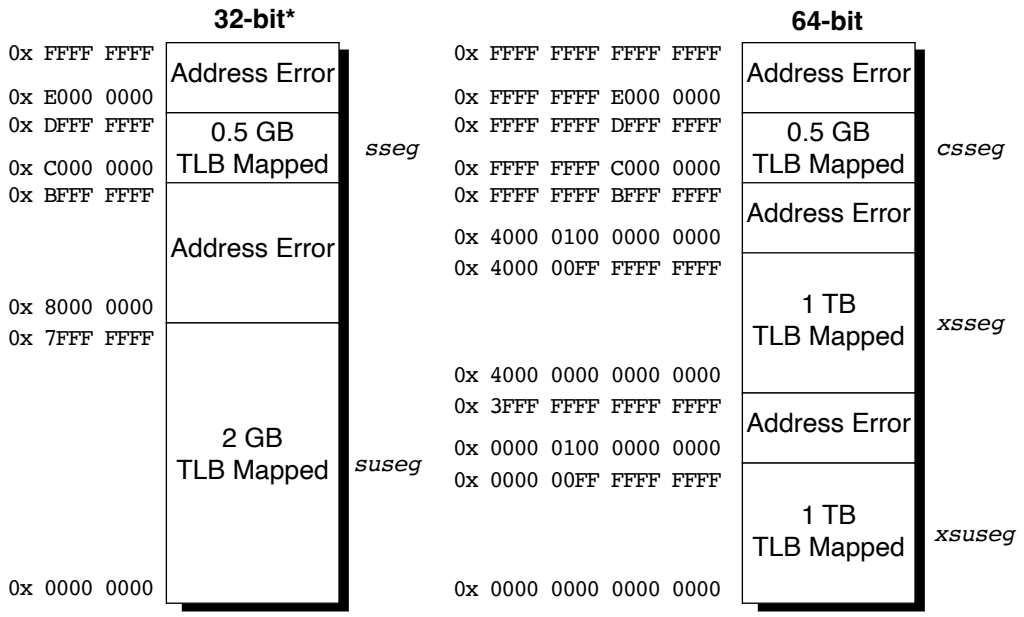
The processor operates in the supervisor mode if the bits of the *Status* register are in the following status:

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

In addition, the addressing mode in the supervisor mode is determined by the *SX* bit of the *Status* register.

- *SX* = 0: 32-bit supervisor space
  TLB miss is processed by a 32-bit TLB miss exception handler.
- *SX* = 1: 64-bit supervisor space
  TLB miss is processed by a 64-bit XTLB miss exception handler.

Table 5-2 shows the features of each segment in the supervisor mode.



* The V_R4300 internally uses 64-bit addresses. In the 32-bit mode, a 32-bit value with bits 32 through 63 sign-extended is used as an address. Normally, the program in the 32-bit mode does not generate an invalid address. However, there is a possibility that an integer overflow may occur as a result of an operation of *base* register + offset to calculate an address. The address calculated at this time is invalid, and the result is undefined. Two causes of the overflow are cited below.

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

Figure 5-5 Supervisor Mode Address Space

Table 5-2 32-Bit and 64-Bit Supervisor Mode Segments

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

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

In Supervisor mode, when *SX* = 0 in the *Status* register and the most-significant bit of the virtual address is set to 0, the *suseg* virtual address space is selected; it covers the full 2^{31} bytes (2 GB) 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 Supervisor Mode, Supervisor Space (*sseg*)

In Supervisor mode, when *SX* = 0 in the *Status* register and the three high-order bits of the virtual address are 110, the *sseg* virtual address space is selected; it covers 2^{29} bytes (512 MB) 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.

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, the *xsuseg* virtual address space is selected; it covers the full 2^{40} bytes (1 TB) 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.

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, the *xsseg* current supervisor virtual address space is selected; it covers the full 2^{40} bytes (1 TB) 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.

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, the *csseg* separate 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.

5.2.4 Virtual Addressing in Kernel Mode

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

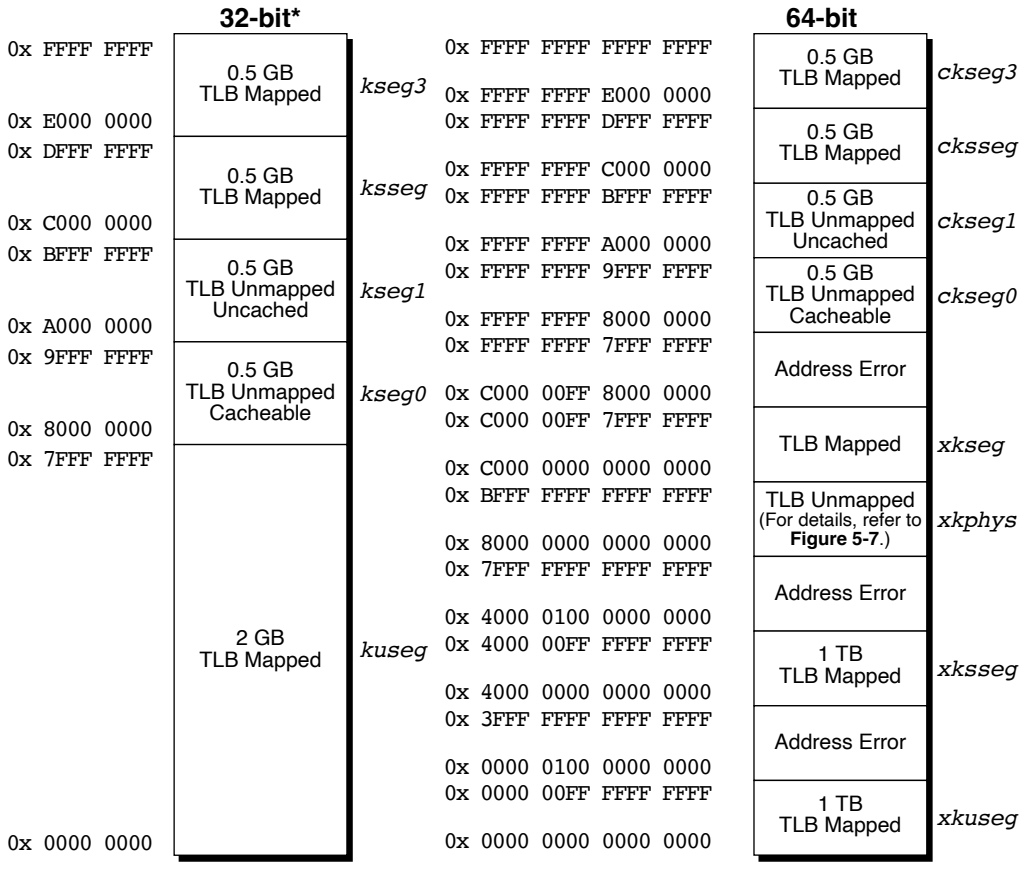
- *KSU* = 00
- *EXL* = 1
- *ERL* = 1

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

- when *KX* = 0, 32-bit kernel space is selected
  TLB miss is processed by a 32-bit TLB miss exception handler.
- when *KX* = 1, 64-bit kernel space is selected
  TLB miss is processed by a 64-bit XTLB miss exception handler.

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

Kernel mode virtual address space is divided into regions differentiated by the high-order bits of the virtual address, as shown in Figure 5-6. Table 5-3 lists the characteristics of the 32-bit kernel mode segments, and Table 5-4 lists the characteristics of the 64-bit kernel mode segments.



* The V_R4300 internally uses 64-bit addresses. In the 32-bit mode, a 32-bit value with bits 32 through 63 sign-extended is used as an address. Normally, the program in the 32-bit mode uses 64-bit instructions. However, there is a possibility that an integer overflow may occur as a result of an operation of base register + offset to calculate an address. The address calculated at this time is invalid, and the result is undefined. Two causes of the overflow are cited below.

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

Figure 5-6 Kernel Mode Address Space

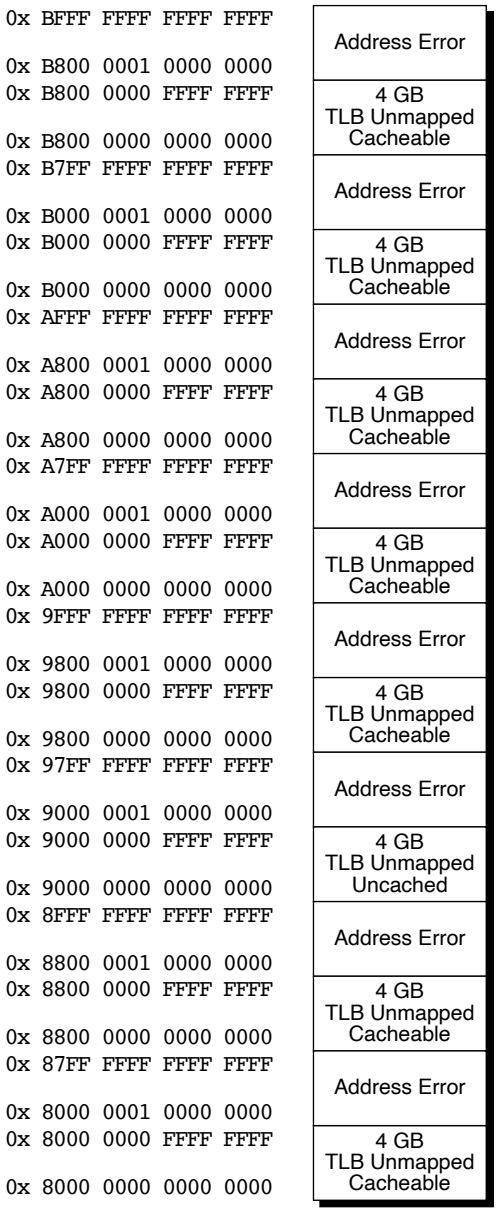


Figure 5-7 Details of xkphys Field

Table 5-3 32-Bit Kernel Mode Segments

Address Bit Values	Status Register Bit Value				Segment Name	Virtual Address	Physical Address	Segment Size
	KSU	EXL	ERL	KX				
A(31) = 0	KSU = 00 or EXL = 1 or ERL =1			0	kuseg	0x0000 0000 through 0x7FFF FFFF	TLB map	2 GB (2^{31} bytes)
A(31:29) = 100				0	kseg0	0x8000 0000 through 0x9FFF FFFF	0x0000 0000 through 0x1FFF FFFF	512 MB (2^{29} bytes)
A(31:29) = 101				0	kseg1	0xA000 0000 through 0xBFFF FFFF	0x0000 0000 through 0x1FFF FFFF	512 MB (2^{29} bytes)
A(31:29) = 110				0	ksseg	0xC000 0000 through 0xDFFF FFFF	TLB map	512 MB (2^{29} bytes)
A(31:29) = 111				0	kseg3	0xE000 0000 through 0xFFFF FFFF	TLB map	512 MB (2^{29} bytes)

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

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

This space is referenced via TLB. Whether the cache can be used or not is determined by the value of the C bit of the TLB entry of each page.

If the *ERL* bit of the *Status* register is 1, the user address area is a 2 GB area that cannot be cached without TLB mapping (i.e., the virtual addresses are used as physical addresses as is). However, this is a function used by the V_R4400 to process an ECC error in an exception handler. This function is defined to maintain the compatibility of the V_R4300 with the V_R4400 because the V_R4300 does not have an ECC and a parity function.

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

In Kernel mode, when *KX* = 0 in the *Status* register and the high-order three bits of the virtual address are 100, *kseg0* virtual address space is selected; it covers the current 2^{29}-byte (512 MB) address 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 controls cacheability. (Refer to **Chapter 6 Exception Processing**.)

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

In Kernel mode, when *KX* = 0 in the *Status* register and the high-order three bits of the virtual address are 101, *kseg1* virtual address space is selected; it covers the current 2^{29}-byte (512 MB) address 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 high-order three bits of the virtual address are 110, the *ksseg* virtual address space is selected; it covers the current 2^{29}-byte (512 MB) virtual address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

This space is referenced via TLB. Whether the cache can be used or not is determined by the value of the C bit of the TLB entry of each page.

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

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

This space is referenced via TLB. Whether the cache can be used or not is determined by the value of the C bit of the TLB entry of each page.

Table 5-4 64-Bit Kernel Mode Segments

Address Bit Values	Status Register Bit Value				Segment Name	Virtual Address	Physical Address	Segment Size
	KSU	EXL	ERL	KX				
A(63:62) = 00	KSU = 00 or EXL = 1 or ERL =1			1	xkuseg	0x0000 0000 0000 0000 through 0x0000 00FF FFFF FFFF	TLB map	1 TB (2^{40} bytes)
A(63:62) = 01				1	xksseg	0x4000 0000 0000 0000 through 0x4000 00FF FFFF FFFF	TLB map	1 TB (2^{40} bytes)
A(63:62) = 10				1	xkphys Refer to **64-bit Kernel Mode, Physical Spaces (xkphy)** on the following page.	0x8000 0000 0000 0000 through 0xBFFF FFFF FFFF FFFF	0x0000 0000 through 0xFFFF FFFF	2^{32} bytes
A(63:62) = 11				1	xkseg	0xC000 0000 0000 0000 through 0xC000 00FF 7FFF FFFF	TLB map	2^{40} to 2^{31} bytes
A(63:62) = 11 A(61:31) = –1				1	ckseg0	0xFFFF FFFF 8000 0000 through 0xFFFF FFFF 9FFF FFFF	0x0000 0000 through 0x1FFF FFFF	512 MB (2^{29} bytes)
A(63:62) = 11 A(61:31) = –1				1	ckseg1	0xFFFF FFFF A000 0000 through 0xFFFF FFFF BFFF FFFF	0x0000 0000 through 0x1FFF FFFF	512 MB (2^{29} bytes)
A(63:62) = 11 A(61:31) = –1				1	cksseg	0xFFFF FFFF C000 0000 through 0xFFFF FFFF DFFF FFFF	TLB map	512 MB (2^{29} bytes)
A(63:62) = 11 A(61:31) = –1				1	ckseg3	0xFFFF FFFF E000 0000 through 0xFFFF FFFF FFFF FFFF	TLB map	512 MB (2^{29} bytes)

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

In Kernel mode, when *KX* = 1 in the *Status* register and bits 63:62 of the virtual address are 00, the *xkuseg* virtual address space is selected; it covers the current 2^{40}-byte (1 TB) user address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

This space is referenced via TLB. Whether the cache can be used or not is determined by the value of the C bit of the TLB entry of each page.

If the *ERL* bit of the status register is 1, the user address area is a 2 GB area that cannot be cached without TLB mapping (i.e., the virtual addresses are used as physical addresses as is). However, this is a function used by the V_R4400 to process an ECC error in an exception handler. This function is defined to maintain the compatibility of the V_R4300 with the V_R4400 because the V_R4300 does not have an ECC and a parity function.

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

In Kernel mode, when *KX* = 1 in the *Status* register and bits 63:62 of the virtual address are 01, the *xksseg* virtual address space is selected; it covers 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.

This space is referenced via TLB. Whether the cache can be used or not is determined by the value of the C bit of the TLB entry of each page.

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

In Kernel mode, when *KX* = 1 in the *Status* register and bits 63:62 of the virtual address are 10, one of the eight unmapped *xkphys* address spaces are selected, either cached or uncached. Bits 31:0 of the virtual address are used as they are as the physical address. Accesses with address bits 58:32 including 1 cause an address error.

Use of the cache is indicated by the bits 61 through 59 of the virtual address. Table 5-5 shows the eight address spaces and use of the corresponding cache.

Table 5-5 Use of Cache and xkphys Address Space

Bits 61 – 59	Use of Cache	Address
0	Used	0x8000 0000 0000 0000 through 0x8000 0000 FFFF FFFF
1	Used	0x8800 0000 0000 0000 through 0x8800 0000 FFFF FFFF
2	Not used	0x9000 0000 0000 0000 through 0x9000 0000 FFFF FFFF
3	Used	0x9800 0000 0000 0000 through 0x9800 0000 FFFF FFFF
4	Used	0xA000 0000 0000 0000 through 0xA000 0000 FFFF FFFF
5	Used	0xA800 0000 0000 0000 through 0xA800 0000 FFFF FFFF
6	Used	0xB000 0000 0000 0000 through 0xB000 0000 FFFF FFFF
7	Used	0xB800 0000 0000 0000 through 0xB800 0000 FFFF FFFF

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

In Kernel mode, when *KX* = 1 in the *Status* register and bits 63:62 of the virtual address are 11 the address space is referred to as *xkseg*. 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
  This space is referenced via TLB. Whether the cache can be used or not is determined by the value of the C bit of the TLB entry of each page.
- 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, and bits 61:32 of the virtual address are 0xFFFF FFFF, bits 31:16 of the virtual address in the 64-bit mode are 0x8000-0xFFFF, as shown in Figure 5-6, select one of the following 512 MB compatibility spaces.

- *ckseg0*. This space is an unmapped region, compatible with the *kseg0* space in 32-bit mode. The *K0* field of the *Config* register controls cacheability and coherency.
- *ckseg1*. This space is an unmapped and uncached region, compatible with the *kseg1* space in 32-bit mode.
- *cksseg*. This space is the current supervisor virtual space, compatible with the *kssе*g space in 32-bit mode.
  This space is referenced via TLB. Whether the cache can be used or not is determined by the value of the C bit of the TLB entry of each page.
- *ckseg3*. This space is current supervisor virtual space, compatible with the *kseg3* space in 32-bit mode.
  This space is referenced via TLB. Whether the cache can be used or not is determined by the value of the C bit of the TLB entry of each page.

5.3 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 5-8 plus a 32-entry TLB. The sections that follow describe how the processor uses each of the TLB-related registers.

Remark Each register is assigned a number called a register number. For details, refer to **Chapter 1 General**. For the relations among the CP0 function, exception processing, and registers, refer to **Chapter 6 Exception Processing**.

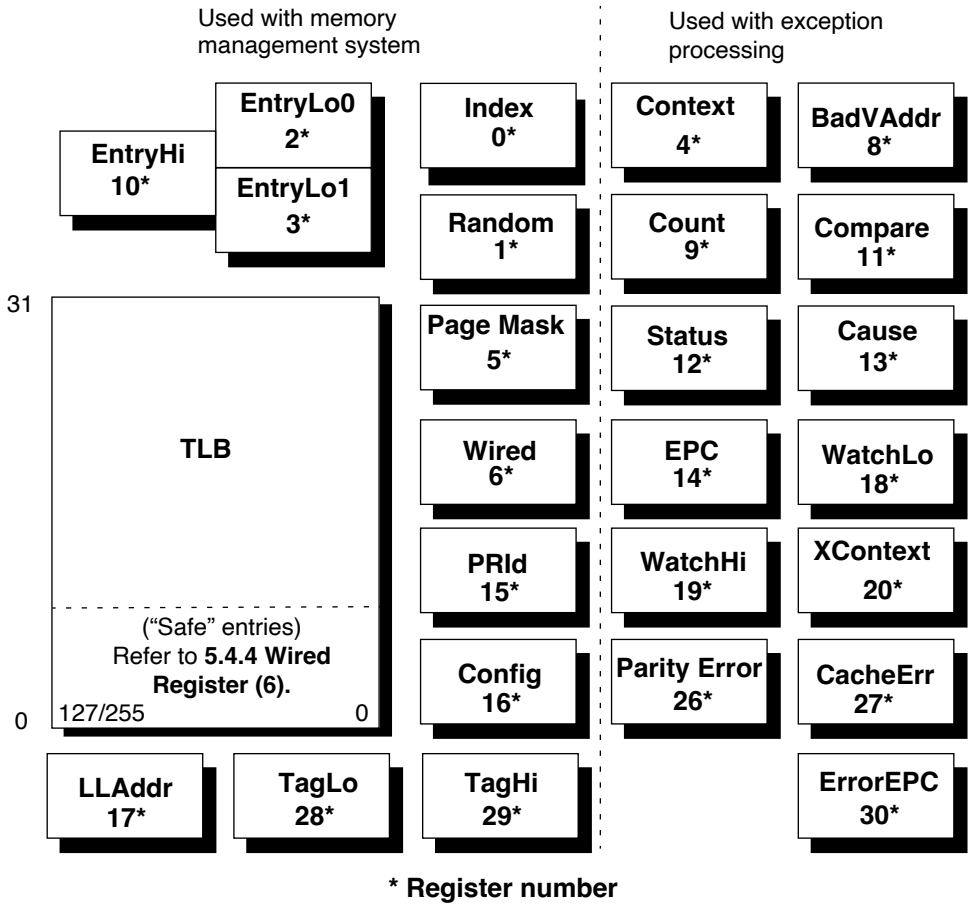


Figure 5-8 CP0 Registers and the TLB

5.3.1 Format of a TLB Entry

Figure 5-9 shows the TLB entry formats for both 32- and 64-bit modes. Each field of an entry has a corresponding field in the *EntryHi*, *EntryLo0*, *EntryLo1*, or *PageMask* registers.

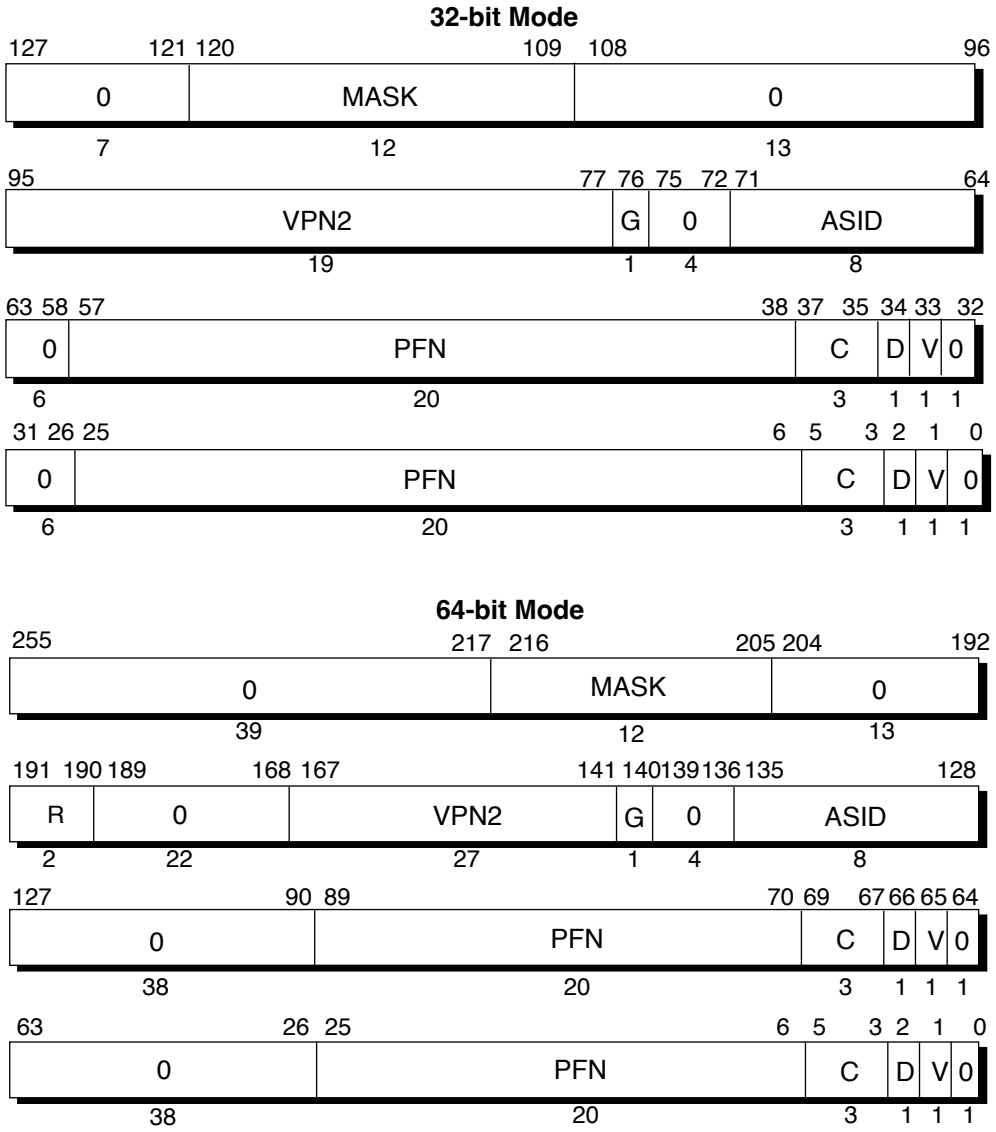
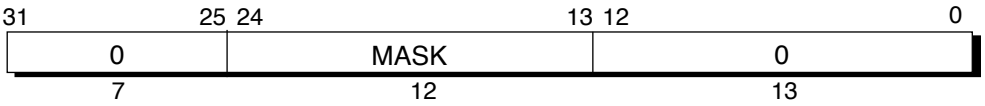


Figure 5-9 TLB Entry Format

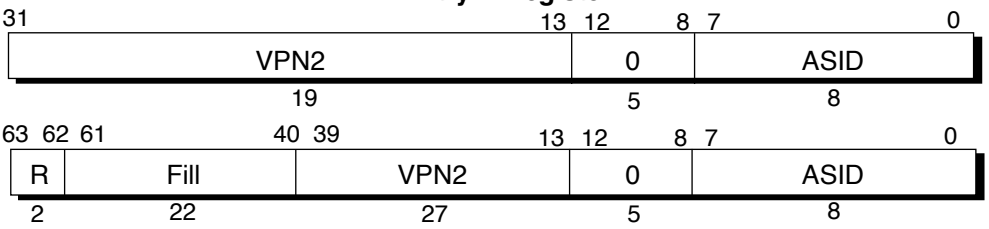
The formats of the *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers are almost the same as the TLB entry. However, the G bit of TLB is undefined with the entry Hi register.

PageMask Register



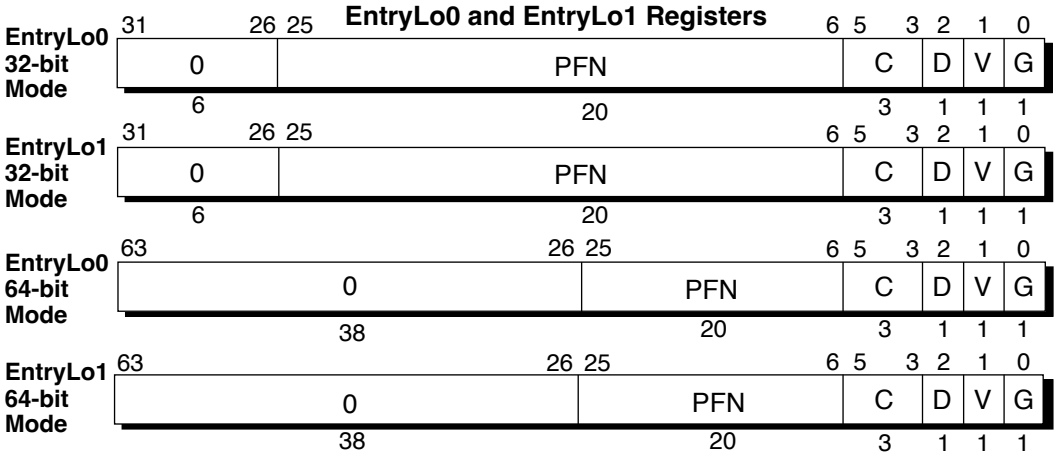
Mask : Page comparison mask. Determines the virtual page size of the corresponding entry.
0 : Reserved for future use (RFU). Must be written as zeroes, and returns zeroes when read.

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; virtual addresses for each process can be shared.
R : Region. (00 → user, 01 → supervisor, 11 → Kernel) used to match $vAddr_{63...62}$
Fill : RFU. Writing this data to this area is ignored. 0 is returned when this bit area read.
0 : RFU. Must be written as zeroes, and returns zeroes when read.

Figure 5-10 TLB Entry Registers (1/2)



PFN : Page frame number; the high-order bits of the physical address.
C : Specifies the TLB page attribute; refer to **Table 5-6**.
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 Entry Lo0 and Entry Lo1, then the processor ignores the ASID during TLB lookup.
0 : RFU. Must be written as zeroes, and returns zeroes when read.

Figure 5-10 TLB Entry Registers (2/2)

Whether the cache is used when a page is referenced is specified by the page coherency attribute (C) bit of the TLB. To use the cache, specify "cache is used" or "cache is not used" by algorithm as a page attribute. **Table 5-6** shows the page attributes selected by the *C* bit.

Table 5-6 Cache Algorithm

Value of C Bit	Cache Algorithm
0	Cache is used
1	Cache is used
2	Cache is not used
3	Cache is used
4	Cache is used
5	Cache is used
6	Cache is used
7	Cache is used

5.4 CP0 Registers

The following sections describe the *CP0* registers that can be accessed through the memory management system and software (each register is followed by its register number in parentheses).

5.4.1 Index Register (0)

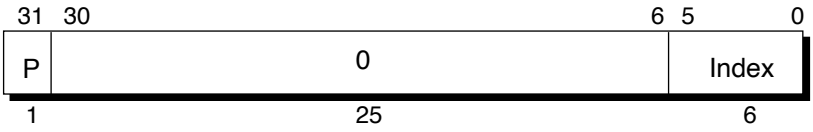
The *Index* register is a 32-bit, read/write register containing six bits to index an entry in the TLB. The most-significant bit of the register shows the success or failure of a TLB Probe (TLBP) instruction.

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

Although the *Index* register *Index* field is six bits wide, only the five least-significant bits (4:0) are used in TLB operations, since the V_R4300 TLB has 32 entries. Bit 5 is readable and writable, but is ignored during TLB operations.

The value of the index register on reset is undefined. Therefore, initialize the *Index* register in software.

Index Register



P : Probe success or failure. Set to 1 when the previous TLBProbe (TLBP) instruction was unsuccessful; set to 0 when successful.
Index : Index to the TLB entry affected by the TLBRead and TLBWrite instructions
0 : RFU. Must be written as zeroes, and returns zeroes when read.

Figure 5-11 Index Register

5.4.2 Random Register (1)

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

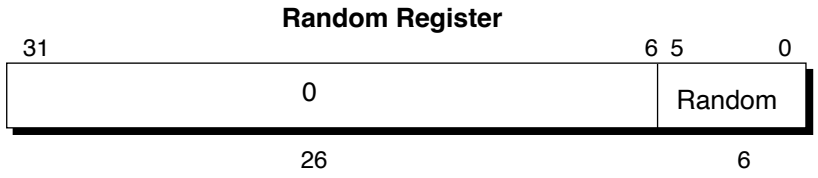
This register decrements as each instruction executes, and its values range between an upper and a lower bound, as follows:

- A lower bound is indicated by the contents of the *Wired* register.
- An upper bound limit is 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 5-12 shows the format of the *Random* register.



Random : TLB Random index.
0 : RFU. Must be written as zeroes, and returns zeroes when read.

Figure 5-12 Random Register

5.4.3 EntryHi (10), EntryLo0 (2), EntryLo1 (3), and PageMask (5) Registers

These registers are used to rewrite the TLB or to check coincidence of a TLB entry when addresses are converted. If the TLB exception occurs, information on the address that has caused the exception is loaded to these registers. Figure 5-10 shows the formats of the *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers.

The values of these registers on reset are undefined. Therefore, initialize the registers by software.

EntryHi Register

The *EntryHi* register is a read/write register and is used to access the high-order bits of the internal TLB.

The *EntryHi* register retains the contents of the high-order bits of a TLB entry when a TLB read or write operation is executed. If a TLB miss, TLB invalid, or TLB modification exception occurs, the virtual page number (VPN2) of the virtual address that has caused the exception and ASID are set to the *EntryHi* register. For the details of the TLB exception, refer to **Chapter 6 Exception Processing**.

ASID is used to write or read the ASID area of the TLB entry. When an address is converted, it is verified against the ASID of the TLB entry as the ASID of the virtual address.

To access this register, use the TLBP, TLBWR, TLBWI, or TLBR instruction.

EntryLo0 and EntryLo1 Registers

EntryLo consists of two registers: *EntryLo0* for even virtual pages and *EntryLo1* for odd virtual pages. *EntryLo0* and *Lo1* registers are read/write registers and are used to access the low-order bits of the internal TLB. When a TLB read/write operation is executed, *EntryLo0* and *Lo1* access the contents of the low-order bits of the TLB entry on an even and odd pages.

PageMask Register

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 5-7. There are seven page sizes selectable. TLB read and write operations use this register as either a destination or a source; when virtual addresses are presented for translation into physical address, the bits 24:13 which are used in the comparison are masked. When the *Mask* field is not one of the values shown in Table 5-7, the operation of the TLB is undefined.

Table 5-7 Mask Field Values for Page Sizes

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

5.4.4 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 5-13. 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.

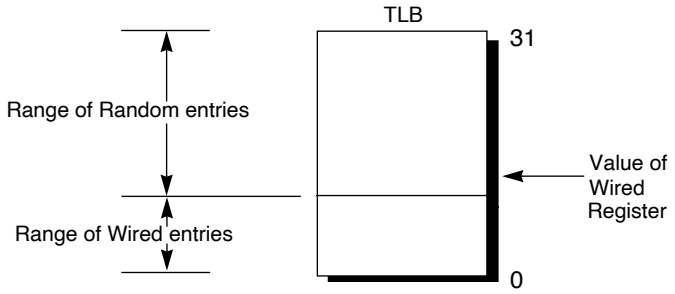
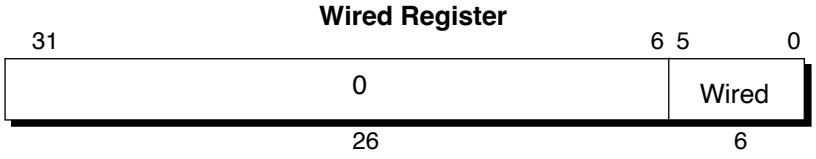


Figure 5-13 Wired Register Boundary

Although the *Wired* field is six bits wide, only the five low-order bits are used in TLB operations, since the V_R4300 TLB has 32 entries. Bit 5 is readable and writable by software, but is ignored during TLB operations.

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 (Refer to **5.4.2 Random Register (1)**). Figure 5-14 shows the format of the *Wired* register.



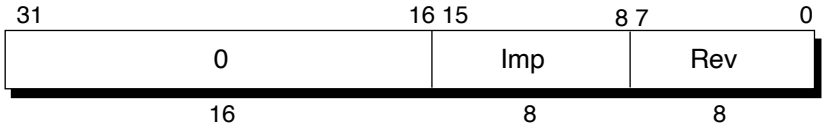
Wired : TLB Wired boundary.
0 : RFU. Must be written as zeroes, and returns zeroes when read.

Figure 5-14 Wired Register

5.4.5 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 5-15 shows the format of the *PRId* register.

PRId Register



Imp : Processor ID number (0x0B for the V_R4300 series™)
Rev : Processor revision number
0 : RFU. Must be written as zeroes, and returns zeroes when read.

Figure 5-15 Processor Revision Identifier Register

The processor revision number is a value in the format of yx. y is the major revision number contained in bits 7:4, and x is the minor revision number contained in bits 3:0.

The processor revision number identifies revision of the chip. However, revision of the chip is not always reflected on the PRID register. Conversely, a change in the revision number does not always reflect on the actual change of the chip. Therefore, develop your program so that it does not depend on the processor revision number area.

5.4.6 Config Register (16)

This register displays or sets various processor statuses of the V_R4300.

Although consideration is given to maintain compatibility of this register with the *Config* register of the V_R4400, some pins of this register are fixed to 0.

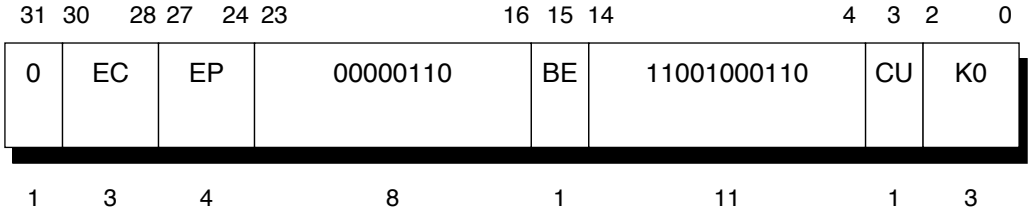
The EP and BE area are initialized on cold reset. These areas can be read or written by software. The default values of these areas are as follows:

EP: 0000

BE: 1

The *CU* bit and K0 area can be read or written in software. However, because these bit and area are not initialized, the user must set the default values to them after reset.

The values of the EP and BE areas can be changed only when initialization is executed in the non-cache area immediately after cold reset and before a store instruction is executed. The operation is not guaranteed if the values of these areas are changed at any other time. Figure 5-16 shows the format of the *Config* register.



EC : Operating frequency ratio (read-only). The value displayed corresponds to the frequency ratio set by the DivMode pins on power application.
(For details of DivMode pin setting, refer to **Table 2-2 Clock/Control Interface Signals.**)

μPD30200-80 (V_R4305)
110 → 1:1 (MasterClock: PClock)
111 → RFU
000 → 1:2
001 → 1:3
Others → RFU
μPD30200-100 (V_R4300)
110 → RFU
111 → 1:1.5 (MasterClock: PClock)
000 → 1:2
001 → 1:3
Others → RFU
μPD30200-133 (V_R4300)
110 → 1:4 (MasterClock: PClock)
111 → RFU
000 → 1:2
001 → 1:3
Others → RFU
μPD30210-133 (V_R4310)
010 → 1:5 (MasterClock: PClock)
011 → 1:6
100 → RFU
101 → 1:3
110 → 1:4
111 → RFU
000 → 1:2
001 → 1:3

Figure 5-16 Config Register (1 / 2)

μPD30210-167 (V_R4310)
010 → 1:5 (MasterClock: PClock)
011 → 1:6
100 → 1:2.5
101 → 1:3
110 → 1:4
111 → RFU
000 → 1:2
001 → 1:3

EP : Sets transfer data pattern (single/block write request).
0 → D (default on cold reset)
6 → DxxDxx: 2 doublewords/6 cycles
Others → RFU

BE : Sets BigEndianMem (endianness).
0 → Little endian
1 → Big endian (default on cold reset)

CU : RFU. However, can be read or written by software.

K0 : Sets coherency algorithm of kseg0 (refer to **Table 5-6 Cache Algorithm**).
010 → Cache is not used
Others → Cache is used

1 : Returns 1 when read.

0 : Returns 0 when read.

Caution If the *BE* bit of this register is changed by using the MTC0 instruction, insert two or more NOP instructions or an instruction other than the load/store instruction in between the MTC0 and load/store instructions.

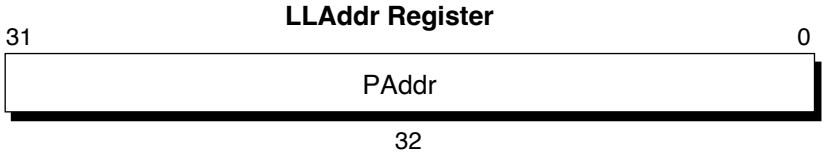
Figure 5-16 Config Register (2 / 2)

5.4.7 Load Linked Address (LLAddr) Register (17)

The read/write *Load Linked Address* (*LLAddr*) register contains the physical address read by the most recent Load Linked instruction. This register is for diagnostic purposes only.

Figure 5-17 shows the format of the *LLAddr* register. The Paddr area in the figure shows the value with the high-order four bits of the physical address PA(31:4) read on execution of the LL instruction zero-extended.

The contents of the LLAddr register are undefined on reset.



PAddr : Stores the bits 31 through 4 of the physical address read by the last LL instruction to bits 27 through 0, and 0 to bits 31 through 28.

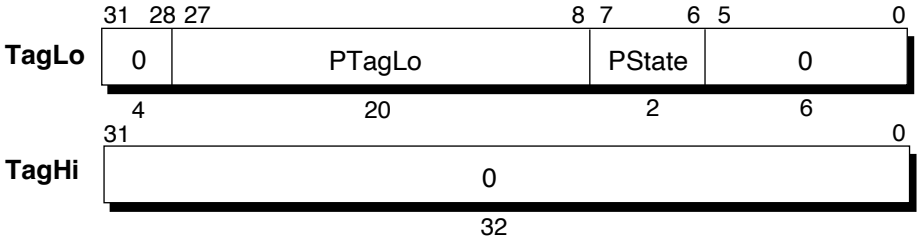
Figure 5-17 LLAddr Register

5.4.8 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 for cache initialization, cache diagnostics, or cache error processing. The *Tag* registers are written by the CACHE and MTC0 instructions.

Figure 5-18 shows the format of these registers.

The contents of these registers are undefined on reset.



PTagLo : Physical address bits 31:12
PState : Specifies the primary cache state
Data cache
11 = Valid
00 = Invalid
Instruction cache
10 = Valid
00 = Invalid
Others = Undefined
0 : RFU. Must be written as zeroes; returns zeroes when read

Cautions 1. If 10 is written to PState by using the CACHE (Index_Store_Tag) instruction, the CACHE is Clean. However, 11 is read when the PState value is read by using the CACHE (Index_Load_Tag) instruction.

2. If 01 is written to PState by using the CACHE (Index_Store_Tag) instruction, the CACHE operation is not guaranteed.

3. If 11 is written to PState by using the CACHE (Index_Store_Tag), the CACHE is Dirty.

Figure 5-18 TagLo and TagHi Register

5.4.9 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 high-order bits* of the virtual address are compared to the contents of the TLB entry, VPN2 (virtual page number divided by two).
- In 64-bit mode, the high-order bits* of the virtual address are compared to the contents of the TLB entry, 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 5-19 illustrates the TLB address translation process.

* The number of bits differs depending on the page size.
  Here are examples where the page size is 16 MB and 4 KB:

Page Size / Mode	16 MB	4 KB
32-bit mode	A (31:25)	A (31:13)
64-bit mode	A63, A62, and A (39:25)	A63, A62, and A (39:13)

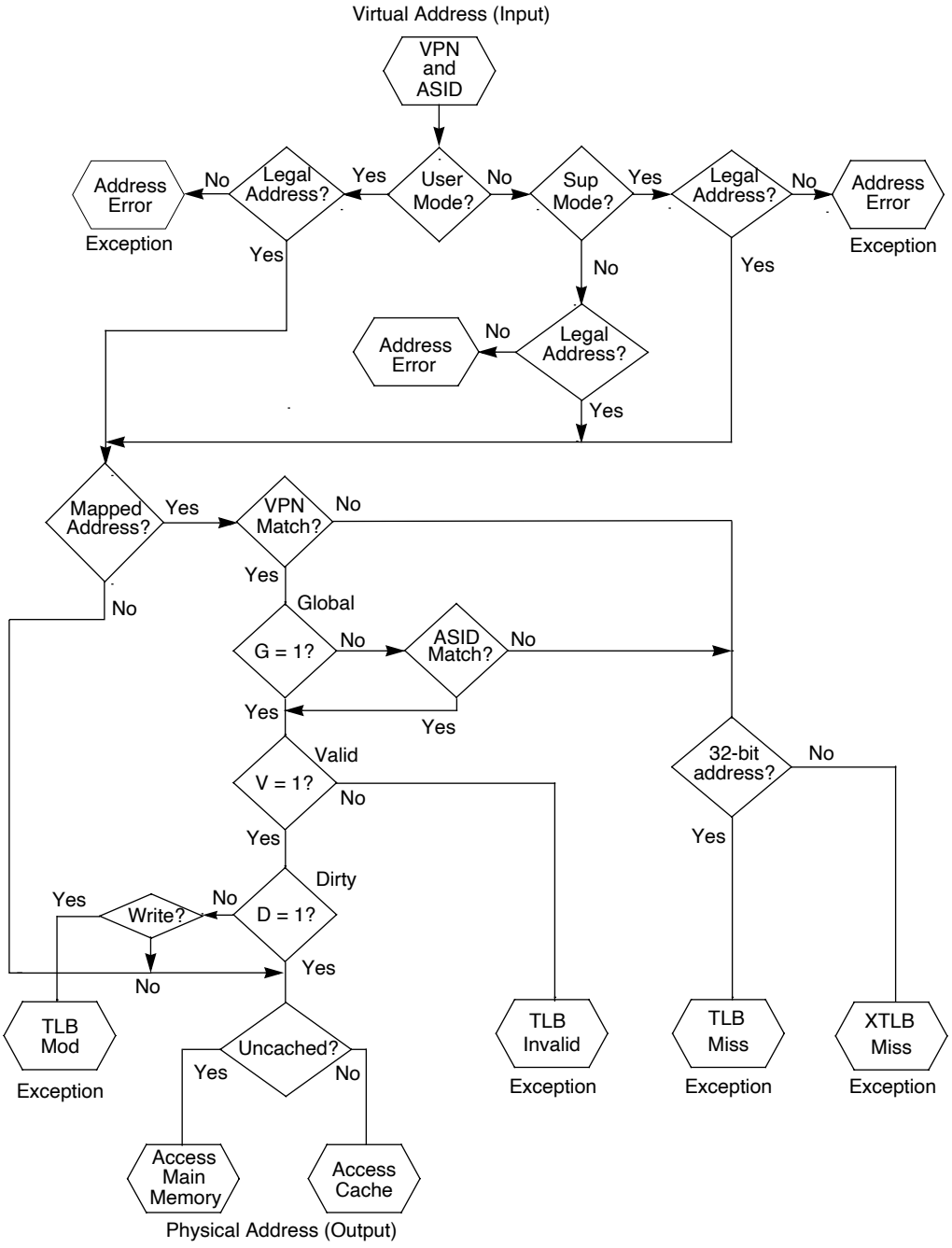


Figure 5-19 TLB Address Translation

5.4.10 TLB Misses

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

* TLB miss exceptions are described in **Chapter 6 Exception Processing**.

5.4.11 TLB Instructions

The following instructions are used to control the TLB.

TLBP (Translation Lookaside Buffer Probe)

Loads a TLB number that matches the contents of the *EntryHi* register to the *Index* register. If the TLB entry does not match, the most significant bit of the *Index* register is set.

TLBR (Translation Lookaside Buffer Read)

Writes the contents of the TLB entry indicated by the *Index* register to the *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers.

TLBWI (Translation Lookaside Buffer Write Index)

Writes the contents of the *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers to the TLB entry indicated by the contents of the *Index* register.

TLBWR (Translation Lookaside Buffer Write Random)

Writes the contents of the *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers to the TLB entry indicated by the contents of the *Random* register.

Exception Processing

6

This chapter describes the exception processing and the hardware used for the exception processing. For the FPU exception, refer to **Chapter 8 Floating-Point Exceptions**.

6.1 Exception Processing Operation

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 (refer to **Chapter 5 Memory Management System** for a description of system operating modes). The processor then disables interrupts and forces execution of a software exception process (called an exception *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 processing has been performed.

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

For the exception processing, the following modes can be set.

- Interrupt enable (*IE*)
- Base operating mode (User, Supervisor, or Kernel)
- Exception level (normal or exception, as indicated by the *EXL* bit in the *Status* register)
- Error level (normal or error, as indicated by the *ERL* bit in the *Status* register).

Each setting condition is described below.

Interrupt Enable

Interrupts are enabled if the following conditions are satisfied.

- *IE* (interrupt enable bit) = 1
- *EXL* bit = 0, *ERL* bit = 0
- Bit of corresponding IM area in status register = 1

Base Operating Mode

The operating mode that is the basis when the exception level is normal (0) is specified by the KSU area of the *Status* register.

Exception/Error Level

The Kernel mode is set when either of the *EXL* or *ERL* bit is set to 1.

When execution returns from exception processing, the exception level is reset to normal (0) (for details, refer to **ERET Instruction** of **Chapter 16 CPU Instruction Set Details)**.

In addition to the above, registers that hold information on addresses, causes, and statuses during exception processing are provided. For details, refer to **6.3 Exception Processing Registers**. For details of the exception processing, refer to **6.4 Exception Details**.

6.2 Precision of Exceptions

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

6.3 Exception Processing Registers

This section describes the *CP0* registers that are used in exception processing. Table 6-1 lists these registers, along with their number—each register has a unique identification number that is referred to as its *register number*. The remaining *CP0* registers are used in memory management, as described in **Chapter 5 Memory Management System**.

Software examines the *CP0* registers to determine the cause of the exception and the state of the CPU at the time the exception occurred. The registers in Table 6-1 are used in exception processing, and are described in the sections that follow.

Table 6-1 CP0 Exception Processing Registers

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

* This register is defined to maintain compatibility between the V_R4300 and V_R4200, and is not used with the hardware of the V_R4300.

Hazard of CP0

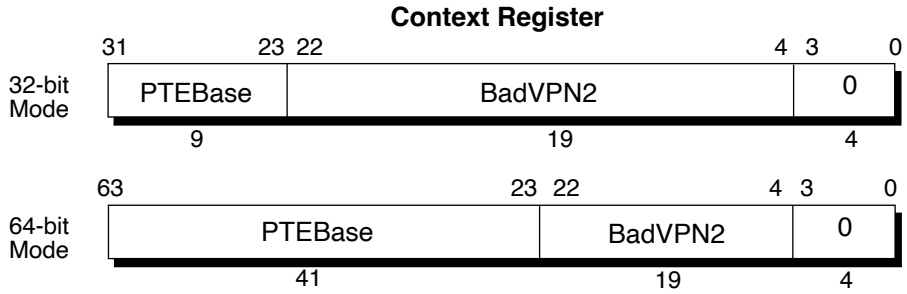
With the *General Purpose* registers of the CPU, when the result of an operation is to be used by the next instruction, the hardware generates a stall and waits until the result can be used. However, the *CP0* register and TLB do not generate a stall. If a value is stored to the *CP0* register, that value may not be used by the immediately following instruction because the value is stored in the register several cycles later. When designing a program, therefore, you must take this into consideration when setting values to the *CP0* register and TLB (for details, refer to **Chapter 19 Coprocessor 0 Hazards**).

6.3.1 Context Register (4)

The *Context* register is a read/write register containing the pointer to an entry in the page table entry (PTE) array on memory; this array is an operating system data structure that stores virtual-to-physical address translations. When there is a TLB miss, the operating system loads the TLB with the missing translation from the PTE array. The *Context* register is used by the TLB Miss exception handler to load the TLB entry.

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 6-1 shows the format of the *Context* register.



PTEBase : Base address of page table entry
BadVPN2 : Page number of virtual address whose translation is invalid divided by 2
0 : RFU. Must be written zeroes; returns zeroes when read

Figure 6-1 Context Register

The *Context* register bit field is described below.

BadVPN2 field is written by hardware on a TLB miss. It contains the virtual page number (VPN2), divided by 2, of the most recent virtual address that did not have a valid translation.

PTEBase area can be read or written and is controlled by the operating system. It is used only by the software as a pointer to the current PTE array on the memory.

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

6.3.2 BadVAddr Register (8)

The *Bad Virtual Address* (*BadVAddr*) register is a read-only register and holds a virtual address that was translated but became invalid last, or a virtual address at which an addressing error occurred. Figure 6-2 shows the format of the *BadVAddr* register.

Caution This register does not hold information even when a bus error exception occurs because it is not an address error exception.

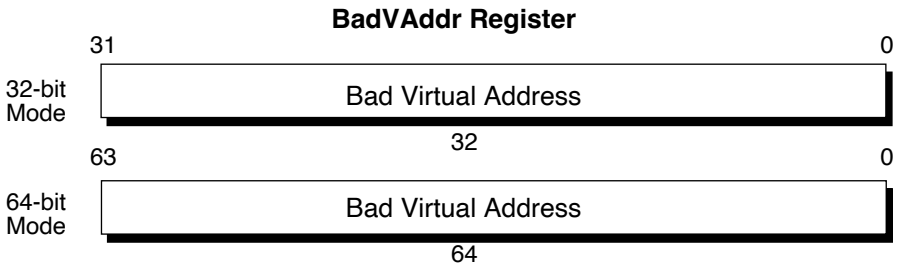


Figure 6-2 BadVAddr Register

6.3.3 Count Register (9)

The read/write *Count* register acts as a timer, incrementing at a constant rate—half the PClock speed—whether or not instructions are being executed. This register is a free-running type. When the register reaches all ones, it rolls over to zero and continues counting. This register can be used for diagnostic purposes, system initialization or synchronization between the processes.

Figure 6-3 shows the format of the *Count* register.

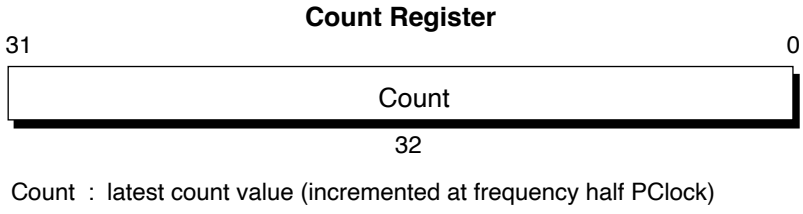
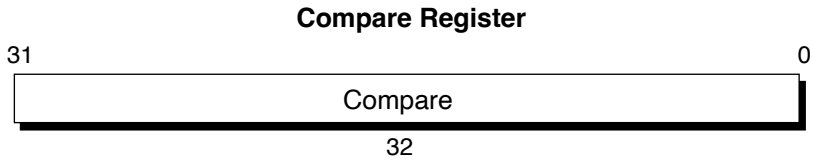


Figure 6-3 Count Register

6.3.4 Compare Register (11)

The *Compare* register is used to generate a timer interrupt; it maintains a stable value that does not change on its own. When the value of the *Compare* register equals the value of the *Count* register (refer to **6.3.3**), interrupt bit *IP(7)* in the *Cause* register is set. This causes an interrupt in the DF stage as soon as the interrupt is enabled. Writing a value to the *Compare* register, as a side effect, clears the timer interrupt.

For diagnostic purposes, the *Compare* register is a read/write register. However, it is usually used as a write register. Figure 6-4 shows the format of the *Compare* register.



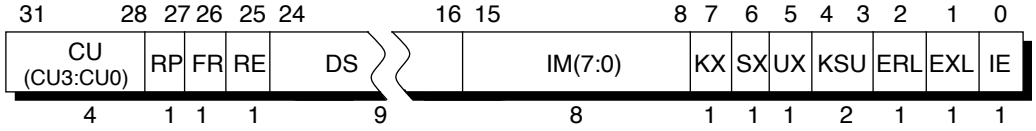
Compare : value to be compared with count register

Figure 6-4 Compare Register

6.3.5 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. Figure 6-5 shows the format of the entire register.

Status Register



CU : Controls the usability of each of the four coprocessor unit numbers.
(1 → usable, 0 → unusable)
CP0 is always usable when in Kernel mode, regardless of the setting of the CU0 bit.
CP2 and CP3 are reserved for future expansion.

RP : Enables low-power operation by reducing the internal clock frequency and the system interface clock frequency to one-quarter speed.
(0 → normal, 1 → low power mode)* (For details, refer to **15.1.2 Low Power Mode**.)

FR : Enables additional floating-point registers
(0 → 16 registers, 1 → 32 registers)

RE : Reverse-Endian bit, enables reverse of system endianness in User mode.
(0 → disabled, 1 → reversed)

DS : Diagnostic Status field (see Figure 6-6, for details).

IM(7:0) : *Interrupt Mask* field, enables external, internal, coprocessors or software interrupts.
(0 → disabled, 1 → enabled)
IM(7) : Mask bit for timer interrupt
IM(6:2) : Mask bits for external interrupts $\overline{\text{Int}}$[4:0], or external write requests
IM(1:0) : Mask bits for software interrupts and IP(1:0) of the Cause register

KX : Enables 64-bit addressing in Kernel mode. When this bit is set, XTLB miss exception is generated on TLB misses in Kernel mode addresses space.
(0 → 32-bit, 1 → 64-bit)
64-bit operation is always valid in Kernel mode.

SX : Enables 64-bit addressing and operations in Supervisor mode. When this bit is set, XTLB miss exception is generated on TLB misses in Supervisor mode addresses space.
(0 → 32-bit, 1 → 64-bit)

UX : Enables 64-bit addressing and operations in User mode. When this bit is set, XTLB miss exception is generated on TLB misses in User mode addresses space.
(0 → 32-bit, 1 → 64-bit)

KSU : Specifies and indicates mode bits
(10 → User, 01 → Supervisor, 00 → Kernel)

ERL : Specifies and indicates error level
(0 → normal, 1 → error)

EXL : Specifies and indicates exception level
(0 → normal, 1 → exception)

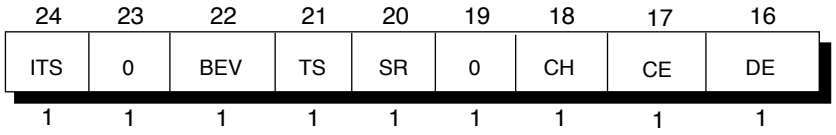
IE : Specifies and indicates global interrupt enable
(0 → disable interrupts, 1 → enable interrupts)

★ * The low power mode is supported only in the 100 MHz model of the V_R4300 and theV_R4305. Fix the RP bit of the 133 MHz model of the V_R4300 and the V_R4310 to 0.

Figure 6-5 Status Register

Figure 6-6 shows the format of the self-diagnostic status (DS) area. All the bits in the DS area, except the TS bit, can be read or written.

Self-Diagnostic Status Field



ITS : Enables Instruction Trace Support.
For details, refer to **9.3.5 Instruction Trace Support**.

BEV : Controls the location of TLB miss and general purpose 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.
0 → does not occur
1 → occur
After TLB shutdown, the processor must be reset to restart. TLB shutdown can occur even when a TLB entry with which the virtual address has matched is set to be invalid (V bit of the entry is cleared).

SR : 0 → Indicates a Soft Reset or NMI has not occurred.
1 → Indicates a Soft Reset or NMI has occurred.

CH : CP0 condition bit.
0 → false
1 → true
Read/write access by software only; not accessible by hardware.

CE, DE : These bits are defined to maintain compatibility with the V_R4200, and is not used by the hardware of the V_R4300.

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

Figure 6-6 Self-Diagnostic Status Field

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

Instruction Trace Support

The V_R4300 can output the physical address at the branch destination from SysAD(31:0) if the instruction address is internally changed by the branch or jump instruction, or occurrence of an exception. To use this function, set the ITS bit to 1.

An instruction cache miss is forcibly generated in the following cases to output the physical address at the branch destination.

- If the branch condition is satisfied when a branch instruction is executed
- If the value of PC is changed by a jump instruction or occurrence of an exception

If an instruction cache miss is generated, SysAD(31:0) issues a processor block read request, which allows an external device to learn a change of the address.

Return response data in response to the processor block read request in the same manner as to the ordinary request. The address to be output is not the value of the PC (virtual address), but a physical address.

Interrupt Enable

Interrupts are enabled when all of the following conditions are satisfied:

- *IE* = 1
- *EXL* = 0
- *ERL* = 0
- When corresponding bit of IM is set to 1

Operating Modes

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

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

32- and 64-bit Modes

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

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

Kernel Address Space Accesses

Access to the kernel address space is allowed when the processor is in Kernel mode.

Supervisor Address Space Accesses

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

User Address Space Accesses

Access to the user address space is allowed in any of the three operating modes.

Status on Reset

The contents of the *Status* register on reset are undefined except for the following bits:

- *TS* and *RP* = 0
- *ERL* and *BEV* = 1
- *SR* = 0 on cold reset; SR = 1 on soft reset or NMI interrupt

Inverting Endian

The V_R4300 is set to big endian at reset. After that, the endian setting can changed by using the *BE* bit of the *Config* register.

- When *RE* bit = 1

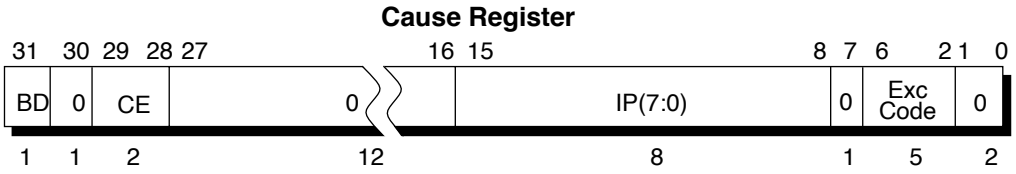
  The endian setting in the Kernel and supervisor modes is specified by the *BE* bit of the *Config* register. The endian setting in the User mode is opposite to the specified endian setting.

- When *RE* bit = 0

  The endian setting in the Kernel, Supervisor mode, and User mode is specified by the *BE* bit of the *Config* register.

6.3.6 Cause Register (13)

The *Cause* register is a 32-bit read/write register and holds the cause of the exception that has occurred last. The 5 bits in the exception code area of this register indicate the cause of the exception (refer to **Table 6-2**). The remaining areas hold detailed information on a specific exception. All the bits, except IP1 and IP0, are read-only. The IP1 and IP0 bits are used to generate the software interrupt. Figure 6-7 shows the format of the *Cause* register, and Table 6-2 describes the exception code area.



BD : Indicates whether the last exception occurred has been executed in a branch delay slot.
1 → delay slot
0 → normal

CE : Coprocessor unit number referenced when a Coprocessor Unusable exception has occurred. If this exception does not occur, undefined.

IP(7:0) : Indicates an interrupt is pending.
1 → interrupt pending
0 → no interrupt
IP(7) : Timer interrupt
IP(6:2) : External normal interrupts. Controlled by $\overline{\text{Int}[4:0]}$, or external write requests
IP(1:0) : Software interrupts. Only these bits can cause interrupt exception when they are set to 1 by software.

ExcCode : Exception code field (refer to **Table 6-2** for details.)

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

Figure 6-7 Cause Register

Table 6-2 Cause Register ExcCode Field

Exception Code Value	Mnemonic	Description
0	Int	Interrupt
1	Mod	TLB Modification exception
2	TLBL	TLB Miss exception (load or instruction fetch)
3	TLBS	TLB Miss 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	–	RFU
15	FPE	Floating-Point exception
16–22	–	RFU
23	WATCH	Watch exception
24–31	–	RFU

The V_R4300 has eight interrupt requests: IP7 through IP0. These interrupt requests are used for the following purposes.

IP7

Indicates whether a timer interrupt request has been issued. This interrupt request is set when the contents of the *Count* register have become equal to those of the compare register.

IP6 through IP2

IP6 through IP2 reflect the logical sum of the two internal registers of the V_R4300. One is the register that latches the status of an interrupt request pin in each cycle, and the other is a register to which data is written by the external write request of the system interface.

IP1 and IP0

IP1 and IP0 set or clear the software interrupt request by manipulating each bit.

For details, refer to **Chapter 14 Interrupts**.

The floating-point exception uses the exception code contained in the floating-point control/status register (refer to **Chapter 8 Floating-Point Exceptions**).

6.3.7 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 that was the direct cause of the exception 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 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 6-8 shows the format of the *EPC* register.

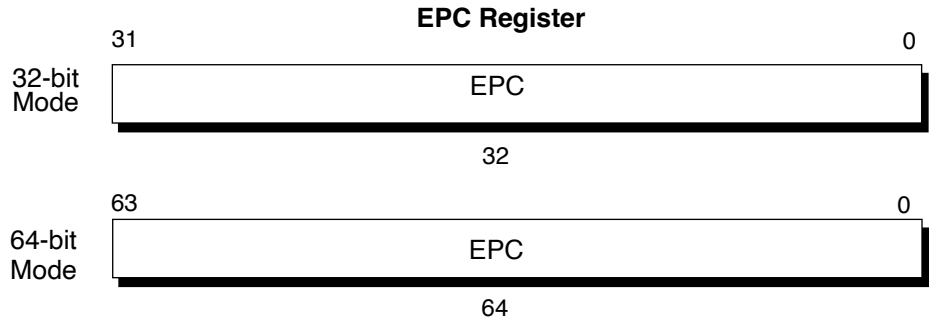
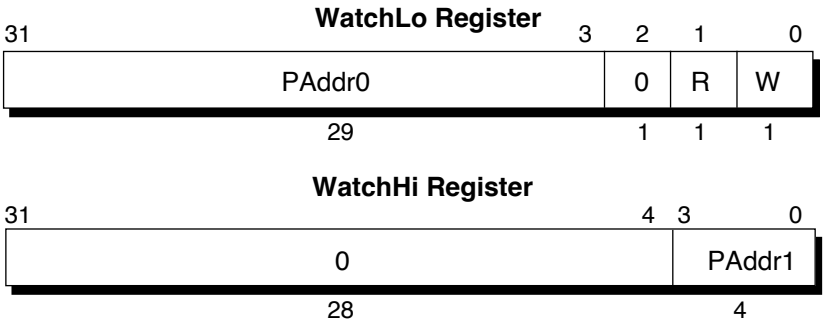


Figure 6-8 EPC Register

6.3.8 WatchLo (18) and WatchHi (19) Registers

The V_R4300 processor provides a debugging feature to detect request of references to a selected physical address; load and store operations cause a Watch exception. Figure 6-9 shows the format of the *WatchLo* and *WatchHi* registers.

Initialize the values of these registers in software since these values are undefined on reset.



PAddr1 : Bits 35:32 of a physical address.
Because the most significant bit of a physical address handled by the V_R4300 is bit 31, the value in this area is invalid.
This area is provided to maintain software compatibility of the V_R4300 with the V_R4400 and V_R4200, and all the 4 bits of this area can be read.

PAddr0 : Bits 31:3 of the physical address

R : Exception occurs when load instruction is executed if set to 1.

W : Exception occurs when store instruction is executed if set to 1.

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

Figure 6-9 WatchLo and WatchHi Registers

6.3.9 XContext Register (20)

The *XContext* register is a read/write register and indicates one entry of the page table entry array (PTE) on the memory. The PTE array is the data structure of the operating system and preserves a conversion table that translates virtual addresses into physical addresses. If a TLB miss occurs, the operating system loads the data that has caused the miss from the PTE to the TLB, and a remedial action is executed by the software.

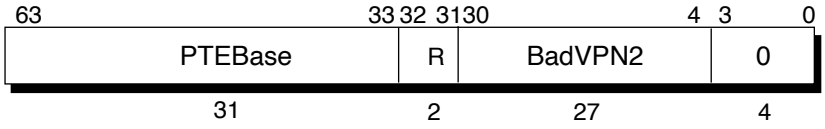
The *XContext* register is used by the XTLB miss exception handler that loads a TLB entry in the 64-bit addressing mode.

Although this register contains several pieces of information that overlap with those of the *BadVAddr* register, it is in the format easy to be used by the XTLB exception handler.

This register is used by the operating system only. The PTEBase area of this register is set as necessary.

Figure 6-10 shows the format of the *XContext* register.

XContext Register



PTEBase : Base address of page table entry
R : Space identifier (bits 63 and 62 of virtual address)
00 → User
01 → Supervisor
11 → Kernel
BadVPN2 : Virtual address whose translation is invalid (bits 39:13)
0 : Must be written as zeroes, and returns zeroes when read.

Figure 6-10 XContext Register

Each bit area of the *XContext* register is described next.

BadVPN2 Area

The BadVPN2 area is written by the hardware in case of a TLB miss.

R Area

The R area is written by the hardware in case of a TLB miss.

PTEBase Area

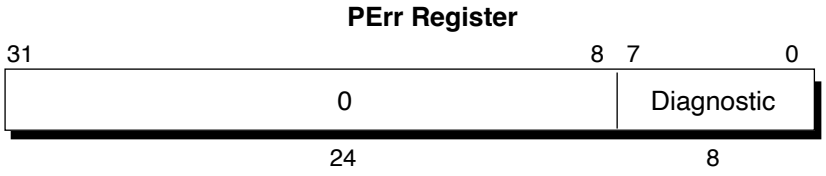
The PTEBase area is a read/write area and is used by the operating system.

The 27-bit BadVPN2 area holds the values of the bits 39:13 of the virtual address that has caused a TLB miss. Because a TLB entry consists of a pair of an even page and an odd page, it does not include bit 12. This register can be used as a pointer that references an 8-byte PTE pair table as it is where the page size is 4 KB. With the page size of 16 KB or more, an appropriate PTE reference address can be generated by shifting or masking the value of this register.

6.3.10 Parity Error (PErr) Register (26)

The *Parity Error* register is a read/write register. This register is defined to maintain the software compatibility of the V_R4300 with the V_R4200. Because the V_R4300 does not have a parity, this register is not used by the hardware.

Figure 6-11 shows the format of the *Parity Error* register.



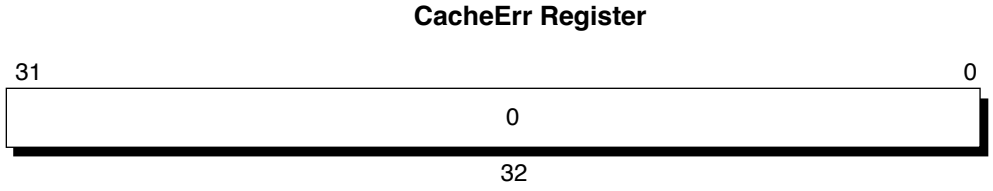
Diagnostic : 8-bit self-diagnosis area
0 : RFU. Must be written as zeroes, and returns zeroes when read.

Figure 6-11 PErr Register

6.3.11 Cache Error (CacheErr) Register (27)

The *Cache Error* register is a read-only register. This register is defined to maintain the compatibility of the V_R4300 with the V_R4200. Because the V_R4300 does not generate a cache error, this register is not used by the hardware.

Figure 6-12 shows the format of the *Cache Error* register.



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

Figure 6-12 CacheErr Register

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

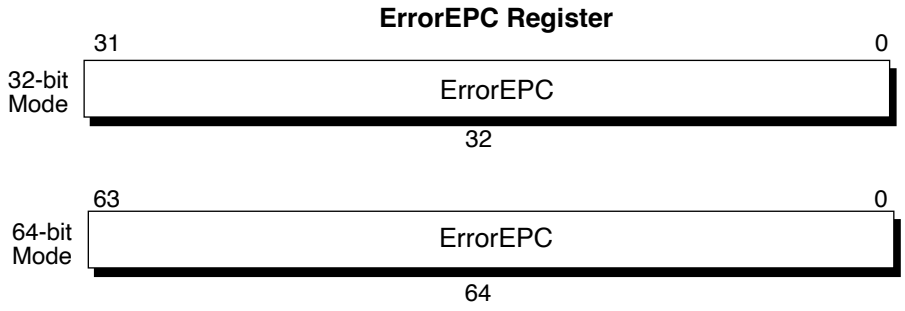
The *ErrorEPC* register is similar to the *EPC* register. It is also used to store the program counter (PC) on Cold Reset, Soft Reset, and nonmaskable interrupt (NMI) exceptions.

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 which is the cause of the error exception is in a branch delay slot.

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

Figure 6-13 shows the format of the *ErrorEPC* register.



ErrorEPC : Indicates the program counter on cold reset or soft reset, or in case of the NMI exception.

Figure 6-13 ErrorEPC Register

6.4 Exception Details

This section describes the processor exceptions (cause, processing, manipulation).

6.4.1 Exception Types

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

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

When the *EXL* and *ERL* bits in the *Status* register are 0 in normal operation either User, Supervisor, or Kernel operating mode is specified by the *KSU* bits in the *Status* register. If one of the *EXL* and *REL* bits is 1, the processor is in the Kernel mode.

If an exception occurs in the processor, the *EXL* bit is set to 1, and the system enters the Kernel mode. After information has been saved, the *EXL* bit is reset to 0 by an exception handler in most of the cases. The *EXL* bit is set to 1 again by an exception handler so that the information that has been saved is not lost due to occurrence of another exception while the information is restored.

When execution exits from the exception processing, the *EXL* bit is reset to 0. For details, refer to **ERET Instruction** of **Chapter 16 CPU Instruction Set Details**.

6.4.2 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

These addresses are a non-cache, non-TLB mapping area.

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

64-bit mode exception and 32-bit mode exception vectors, and their offsets are shown next.

Table 6-3 64-Bit Mode Exception Vector Base Addresses

	Vector Base Address	Vector Offset
Cold Reset, Soft Reset, and NMI	0xFFFF FFFF BFC0 0000 (BEV bit is automatically set to 1.)	0x0000
TLB Miss, EXL=0	0xFFFF FFFF 8000 0000 (BEV=0) 0xFFFF FFFF BFC0 0200 (BEV=1)	0x0000
XTLB Miss, EXL=0		0x0080
Other		0x0180

Table 6-4 32-Bit Mode Exception Vector Base Addresses

	Vector Base Address	Vector Offset
Cold Reset, Soft Reset, and NMI	0xBFC0 0000 (BEV bit is automatically set to 1.)	0x0000
TLB Miss, EXL=0	0x8000 0000 (BEV=0) 0xBFC0 0200 (BEV=1)	0x0000
XTLB Miss, EXL=0		0x0080
Other		0x0180

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

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

E.g. General Exception vector: When *BEV* = 0, the vector base address for this exception vector is in *kseg0* (uncached, unmapped space) (0x8000 0180 in 32-bit mode, 0xFFFF FFFF 8000 0180 in 64-bit mode).

When *BEV* = 1, the vector base address for this exception vector is in *kseg1* (uncached, TLB unmapped space) (0x8000 0180 in 32-bit mode and 0xFFFF FFFF BFC0 0380 in 64-bit mode).

This space is an uncached and TLB unmapped space, allowing the exception handler to bypass the cache and TLB.

6.4.3 Priority of Exceptions

While more than one exception can occur for a single instruction, only the exception with the highest priority is reported.

The priority is as follows:

Table 6-5 Exception Priority Order

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

Generally speaking, the exceptions described in the following sections are handled (“processing”) by hardware; these exceptions are handled (“servicing”) by software.

6.4.4 Cold Reset Exception

Cause

The Cold Reset exception occurs when the **$\overline{\text{ColdReset}}$** signal is asserted and then deasserted. This exception is not maskable.

Processing

The CPU provides a special interrupt vector for this reset 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:

- The *TS*, *SR*, and *RP* bits of the *Status* register and the *EP*(3:0) bits of the *Config* register are cleared to 0.
- The *ERL* and *BEV* bits of the *Status* register and the *BE* bit of the *Config* register are set to 1.
- The *Random* register is set to the upper-limit value (31).
- The *EC*(2:0) bits of the *Config* register are set to the contents of the DivMode(1:0)* pins.

* In V_R4300 and V_R4305. In V_R4310, DivMode(2:0).

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

6.4.5 Soft Reset Exception

Cause

A Soft Reset (sometimes called Warm Reset) occurs when the **ColdReset** signal remains deasserted while the **Reset** pin is deasserted after assertion of more than 16 **MasterClock** cycles.

A Soft Reset immediately resets all state machines, and sets the *SR* bit of the *Status* register. Execution begins at the reset vector when a Soft Reset occurs.

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:

- The program counter value when this exception occurs is set to the *ErrorEPC* register, when the *ERL* bit of the *Status* register is 0.
- *TS* and *RP* bits of the *Status* register are cleared to 0.
- *ERL*, *SR*, and *BEV* bits of the *Status* register are set to 1.

Because the Soft Reset can abort cache and access to the system interface, cache and memory state is undefined when this exception occurs.

Servicing

The Soft Reset exception is serviced by saving the current processor state for self-diagnostic purposes, and reinitializing the system in the same manner as the Cold Reset exception.

6.4.6 Non-Maskable Interrupt (NMI) Exception

Cause

The Non-maskable Interrupt (NMI) exception occurs in response to the falling edge of the $\overline{\text{NMI}}$ pin. An NMI can also be set by externally writing 1 to the bit 6 of the internal interrupt register through the **SysAD6** 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 this exception. When an NMI exception occurs, the *SR* bit of the *Status* register is set to differentiate this exception from a 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:

- The program counter value when this exception occurs is set to the *ErrorEPC* register.
- *TS* bit of the *Status* register are cleared to 0.
- *ERL*, *SR*, and *BEV* bits of the *Status* register are set to 1.

Servicing

The NMI exception is serviced by saving the current processor state for self-diagnostic purposes, and reinitializing the system in the same manner as the Cold Reset exception.

6.4.7 Address Error Exception

Cause

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

- Execute the LW or SW instruction to the word data that is not located at the word boundary.
- Execute the LH or SH instruction to the halfword data that is not located at the halfword boundary.
- Execute the LD or SD instruction to the doubleword data that is not located at the doubleword boundary.
- Reference the Kernel address space from User or Supervisor mode
- Reference the supervisor address space from User mode
- Reference an address not in Kernel, Supervisor, or User space in 64-bit Kernel, Supervisor, or User mode.

This exception is not maskable.

Processing

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

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.

Servicing

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

6.4.8 TLB Exceptions

Three types of TLB exceptions can occur:

- TLB Miss 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 (V bit = 0).
- TLB Modification 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, D bit = 0). As a result, this exception only occurs for the data cache, resulting in a lower priority for this exception.

The following describe these TLB exceptions.

TLB Miss Exception (32-bit mode)/XTLB Miss Exception (64-bit mode)

Cause

The TLB (XTLB) Miss exception occurs when there is no TLB entry to match an address to be referenced. This exception is not maskable.

Processing

There are two special vectors for this exception. One is for the 32-bit mode, and the other is for the 64-bit mode. 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 Miss exceptions use these two special vectors when the *EXL* bit is set to 0 in the *Status* register, and they use the common exception vector when the *EXL* bit is set to 1 in the *Status* register.

This exception sets the *TLBL* or *TLBS* code to the *ExcCode* area of the *Cause* register. If the cause of the exception is an instruction reference or load operation, the *TLBL* code is set; if the cause is a store operation, the *TLBS* code is set.

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 load memory words containing the physical page frame and access control bits to a pair of TLB entries. Memory words are written into the TLB through the *EntryLo0*/*EntryLo1*/*EntryHi* register.

It is possible that the page frame and access control bit are placed on a page where the virtual address is not resident in the TLB. This condition is processed by allowing a TLB Miss exception in the TLB Miss exception handler. This second exception goes to the common exception vector because the *EXL* bit of the *Status* register is set.

TLB Invalid Exception

Cause

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

Processing

The common exception vector is used for this exception. The *TLBL* or *TLBS* code is set to the *ExcCode* field of the *Cause* register. If the cause of the exception is an instruction reference or load operation, the *TLBL* code is set; if the cause is a store operation, the *TLBS* code 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 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 removing the cause of a TLB Invalid exception, place another entry to the location of the TLB entry where the exception has occurred by the TLB Probe (TLBP) instruction and set 1 to the *V* bit.

TLB Modification Exception

Cause

The TLB change exception occurs if the TLB entry that matches the virtual address referenced by the store instruction is disabled from being written (the *D* bit is 0), though the TLB entry is valid (*V* bit is 1). This exception occurs only when an attempt is made to write the data cache. Note, however, that the priority of this exception is low.

Processing

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

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 bits. 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 contents of the *EntryHi* and *EntryLo* registers are written into the TLB.

6.4.9 Bus Error Exception

Cause

A Bus Error exception is raised by board-level circuitry for events such as bus time-out, local 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 field reference, or unbuffered write occurs synchronously; in concrete terms, a Bus Error exception occurs if SysCmd(0) indicates that the data contains an error when it is transferred on the system bus, regardless of the direction of the transfer between the system and the processor. An exception for the local bus error of the system resulting from a buffered write transaction is generated using the interrupt exception.

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. If the cause of the exception is an instruction reference (instruction fetch), the *IBE* code is set. If the cause is a data reference (load/store), the *DBE* code is set.

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 system control coprocessor registers.

- If the *IBE* code in the *Cause* register is set (indicating an instruction fetch), 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), the virtual address of the instruction that caused the exception (the address of the preceding branch instruction if the *BD* bit of the *Cause* register is set) is stored 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.

6.4.10 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 is set to the *ExcCode* field in the *Cause* register.

The *EPC* register contains the address of the SYSCALL instruction unless it is in a branch delay slot. If the SYSCALL instruction 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; 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, the branch instruction is decoded to branch and re-execute.

6.4.11 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 is set to the *ExcCode* in the *Cause* register.

The *EPC* register contains the address of the BREAK instruction unless it is in a branch delay slot. If the BREAK instruction 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, otherwise the bit is cleared.

Servicing

When the Breakpoint exception occurs, servicing is transferred to the applicable system routine. Additional information can be passed using the unused bits of the BREAK instruction (bits 25:6). This information can be obtained by reading the contents indicated by the *EPC* register as data. (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, decode the branch instruction to get the branch destination and resume execution.

6.4.12 Coprocessor Unusable Exception

Cause

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

- If use of the corresponding coprocessor unit is not marked usable (*CU* bits (3:1) of the *Status* register = 0).
- If the CP0 instruction is executed in the User or Supervisor mode when CP0 cannot be used (*CU0* bit of the *Status* register = 0).

This exception is not maskable.

Processing

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

The *CE* bits of the *Cause* register indicate which of the four coprocessors was referenced.

The *EPC* register indicates the coprocessor instruction that caused an exception. If the coprocessor instruction that caused the exception is in a branch delay slot, the *EPC* register indicates 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 *CE* bit of the *Cause* register, process as follows by a handler.

- If the process is entitled access to the coprocessor, the coprocessor is marked usable and the coprocessor resumes execution.
- If the process is entitled access to the coprocessor, but the coprocessor does not exist or has failed, decoding of the coprocessor instruction is possible.
- If the *BD* bit is set in the *Cause* register, the branch instruction must be decoded; then the coprocessor instruction can be emulated and execution resumed by making the contents of the *EPC* register advanced past the coprocessor instruction.

- If the process is not entitled access to the coprocessor, the Kernel informs the current process of the UNIX SIGILL/ILL_PRIVIN_ FAULT (illegal instruction/privileged instruction fault) signal. This exception is usually fatal.

6.4.13 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 opcode (bits 31:26)
- an attempt is made to execute a SPECIAL instruction with an undefined sub-opcode (bits 5:0)
- an attempt is made to execute a REGIMM instruction with an undefined sub-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 is set in the *ExcCode* field in the *Cause* register.

The *EPC* register indicates the instruction that caused an exception if the reserved instruction is not in a branch delay slot, in which case the *EPC* register indicates the preceding branch instruction and the *BD* bit of the *Cause* register is set.

Servicing

All instructions in the MIPS ISA that are currently defined can be executed.

The process executing at the time of this exception is handled by a UNIX SIGILL/ ILL_RESOP_FAULT (illegal instruction/reserved operand fault) signal. This exception is usually fatal.

6.4.14 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 is set in the *ExcCode* field in the *Cause* register.

The *EPC* register indicates the Trap instruction that caused the exception. If the instruction is in a branch delay slot, the *EPC* register indicates 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 by Kernel. This exception is usually a fatal error.

6.4.15 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 is set in the ExcCode field in the *Cause* register.

The *EPC* register indicates the instruction that caused the exception. If the instruction is in a branch delay slot, the *EPC* register indicates 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 by Kernel. This exception is usually a fatal error to the current process.

6.4.16 Floating-Point Exception

Cause

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

Processing

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

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

The *EPC* register indicates the reserved instruction if the instruction is not in a branch delay slot. If the instruction is in the branch delay slot, the *EPC* register indicates the preceding branch instruction and the *BD* bit of the *Cause* register is set.

Servicing

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

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

6.4.17 Watch Exception

Cause

A Watch exception occurs when a load or store instruction references the physical address specified in the *WatchLo*/*WatchHi* registers. The exception is caused by the following instructions: a load instruction when the *R* bit is set in the *WatchLo* register; a store instruction when the *W* bit is set in the *WatchLo* register; a load or store instruction when both the *R* and *W* bits are set in the *WatchLo* register.

The CACHE instruction never causes a Watch exception.

The Watch exception is postponed if the *EXL* bit is set in the *Status* register. The ★ Watch exception is maskable by setting the *EXL* bit in the *Status* register to 1 or by clearing the *R* and *W* bits in the *WatchLo* register to 0.

Processing

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

The *EPC* register indicates the Load and Store instructions if they are not in a branch delay slot. If these instructions are in the branch delay slot, the *EPC* register indicates 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 masked to execute the faulting instruction. The Watch exception must then be reenabled.

★ Because the contents of the *WatchLo*/*WatchHi* registers become undefined after reset, initialize the registers by software (especially clear the *R* and *W* bits to 0). If not initialized, the Watch exception may occur.

6.4.18 Interrupt Exception

Cause

The Interrupt exception occurs when one of the eight interrupt conditions (one for timer interrupt; five for hardware interrupt; two for software interrupt) is asserted. The significance of these interrupts is dependent upon the specific system implementation. An interrupt request signal from a pin is detected by the level.

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, setting the *EXL* bit, or setting the *ERL* bit of the *Status* register.

Processing

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

The *IP* field of the *Cause* register indicates current interrupt requests. It is possible before this register is read that more than one of the bits can be simultaneously set if the interrupt request signal is asserted; or that more than one of the bits can be simultaneously cleared if the interrupt request signal is deasserted.

If the instruction that causes an exception is not in a branch delay slot the *EPC* register indicates that instruction. If the instruction is in the branch delay slot, the *EPC* register indicates 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 an interrupt is generated by the hardware, the interrupt is cleared by asserting inactive the interrupt request signal that has caused the interrupt.

If the timer interrupt request is generated, either clear the *IP7* bit of the *Cause* register or change the contents of the *Compare* register, to clear this interrupt.

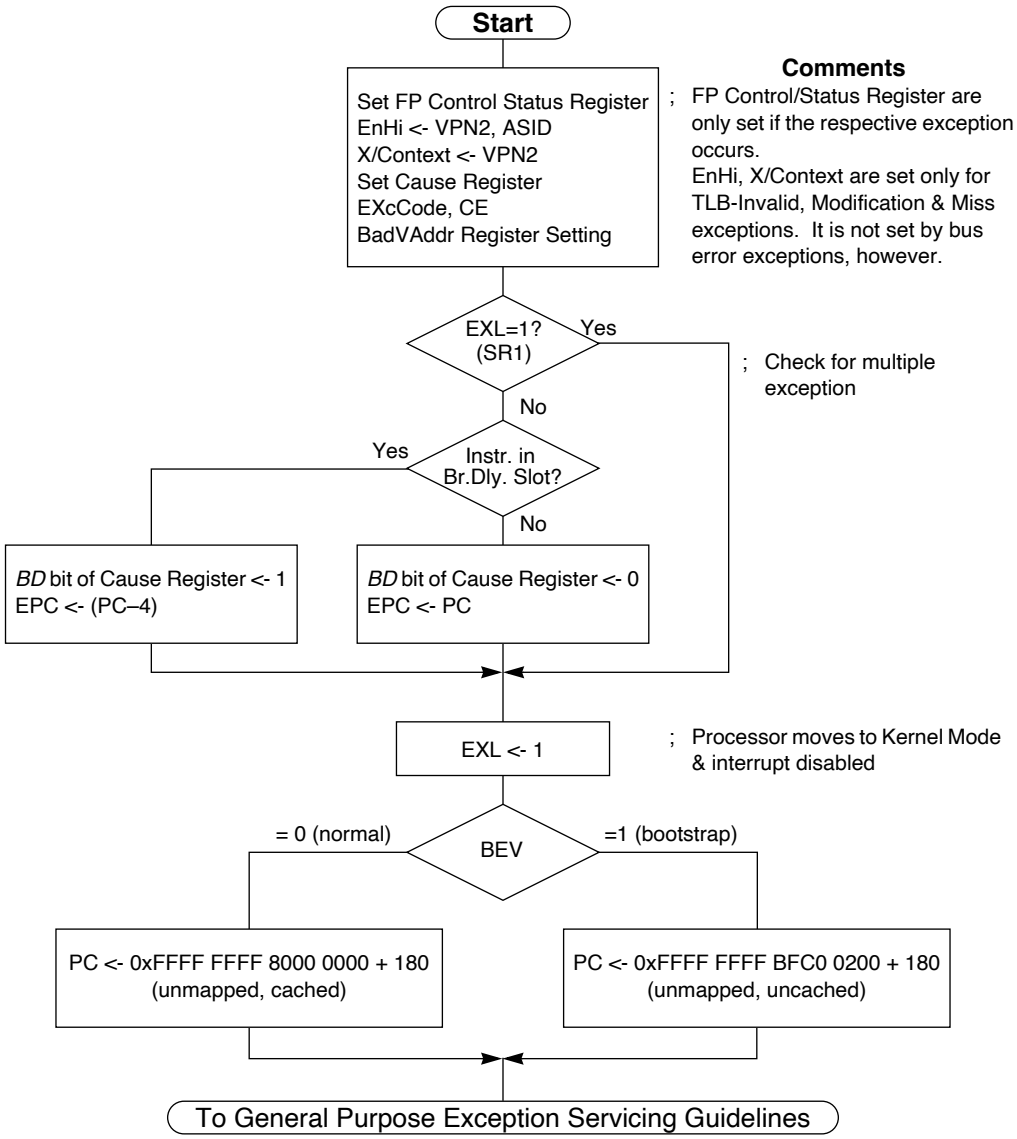
6.5 Exception Handling and Servicing Flowcharts

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

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

Generally speaking, the exceptions are handled (“processing”) by hardware; the exceptions are then handled (“servicing”) by software.

(a) Exceptions other than Cold Reset, Soft Reset, NMI, or TLB/XTLB Miss Handling (Hardware)



Remark Interrupts can be masked by IE or IMs and Watch is postponed if EXL = 1

Figure 6-14 General Purpose Exception Handler (1/2)

(b) General Purpose Exception Servicing Guidelines (Software)

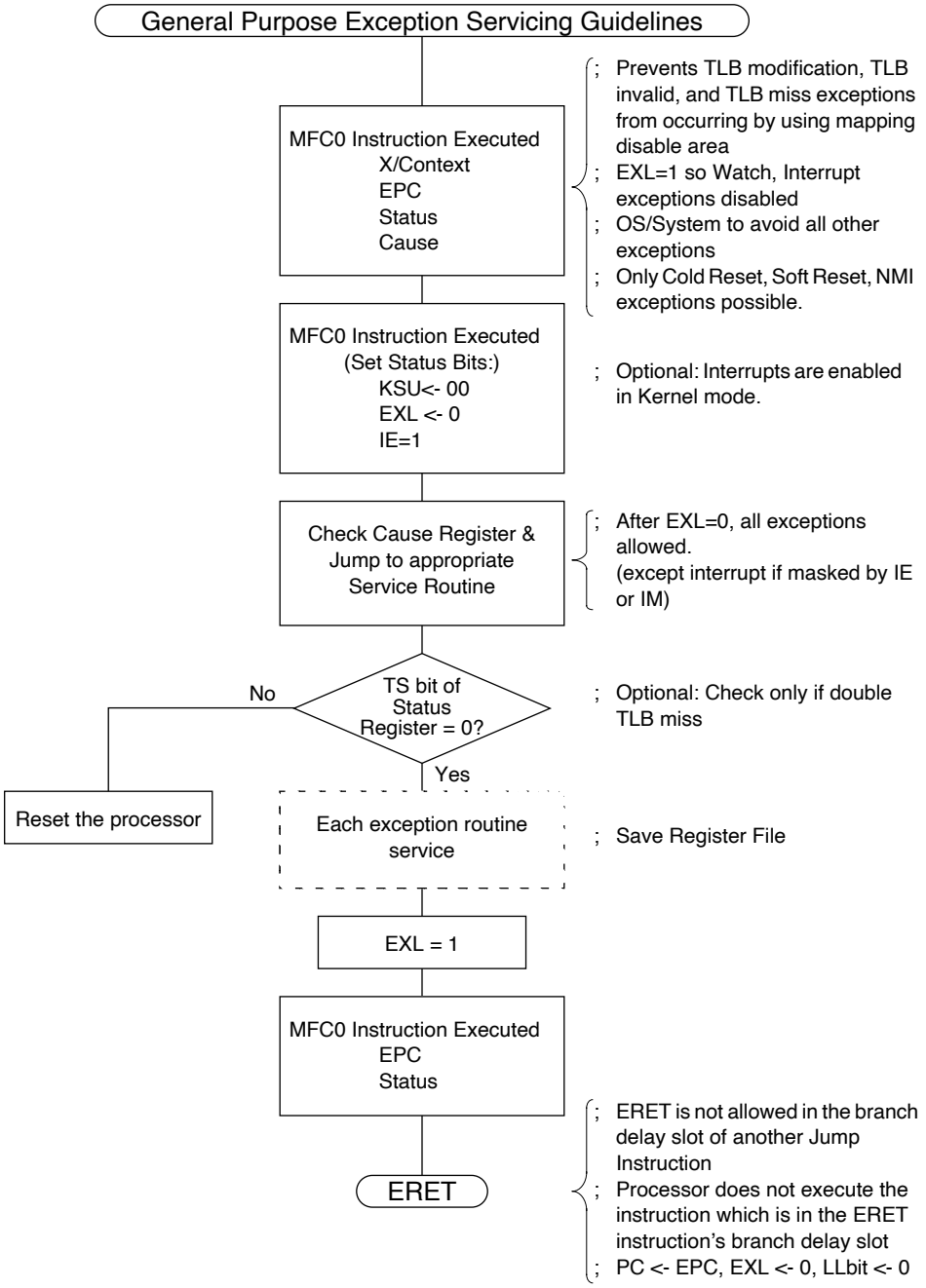


Figure 6-14 General Purpose Exception Handler (2/2)

(a) Hardware

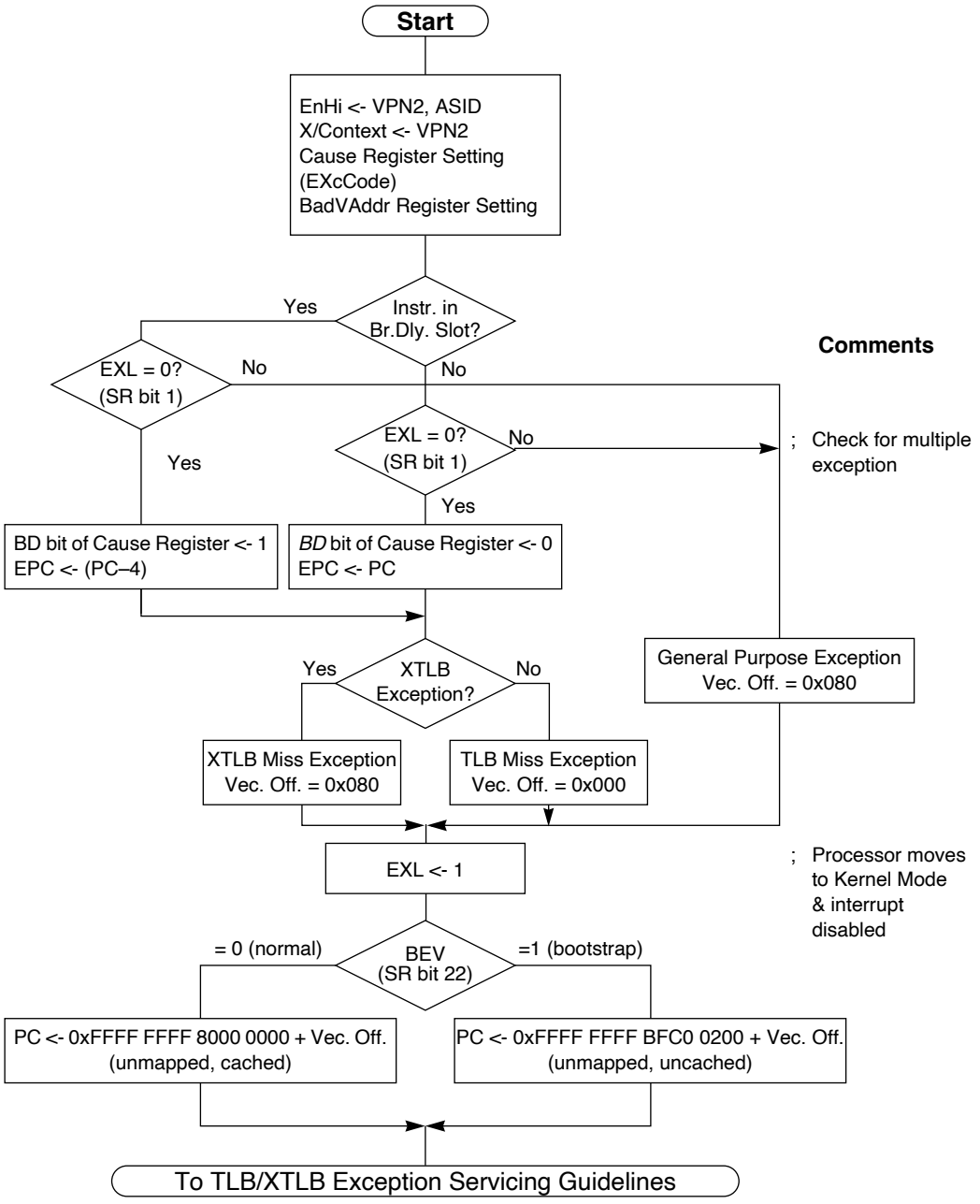


Figure 6-15 TLB/XTLB Miss Exception Handler (1/2)

(b) TLB/XTLB Exception Servicing Guidelines (Software)

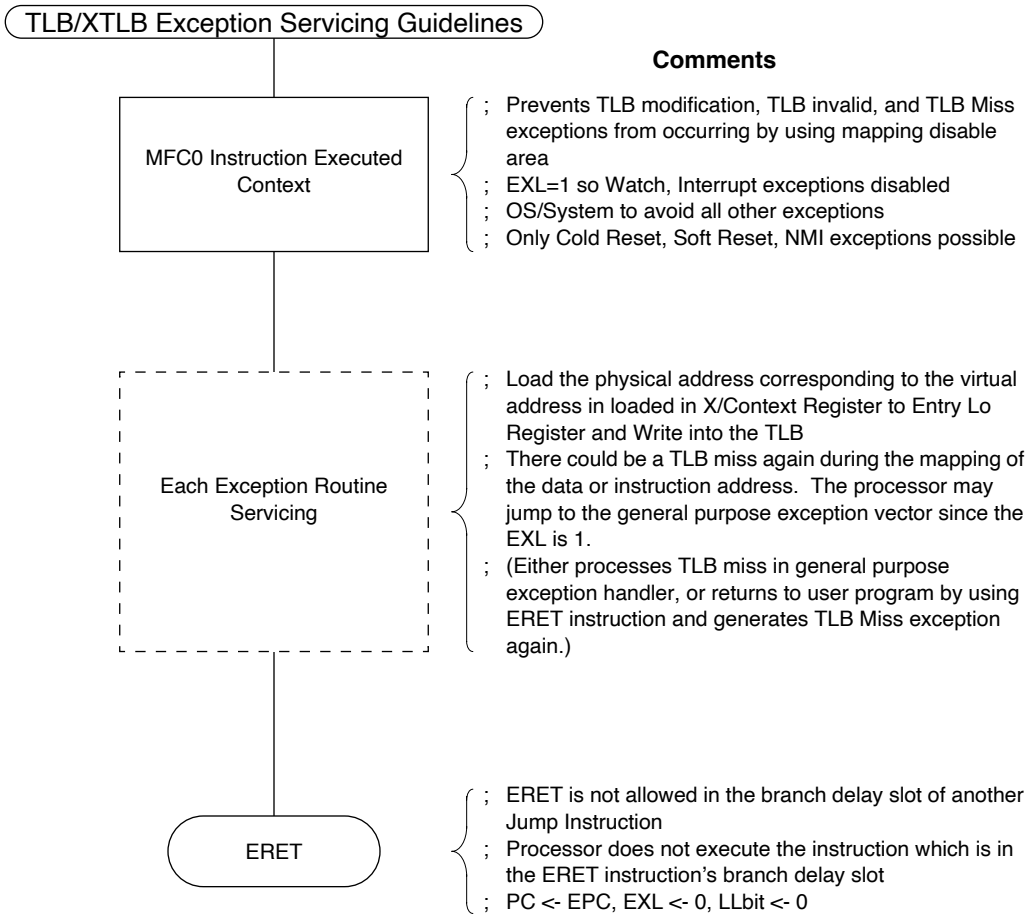


Figure 6-15 TLB/XTLB Miss Exception Handler (2/2)

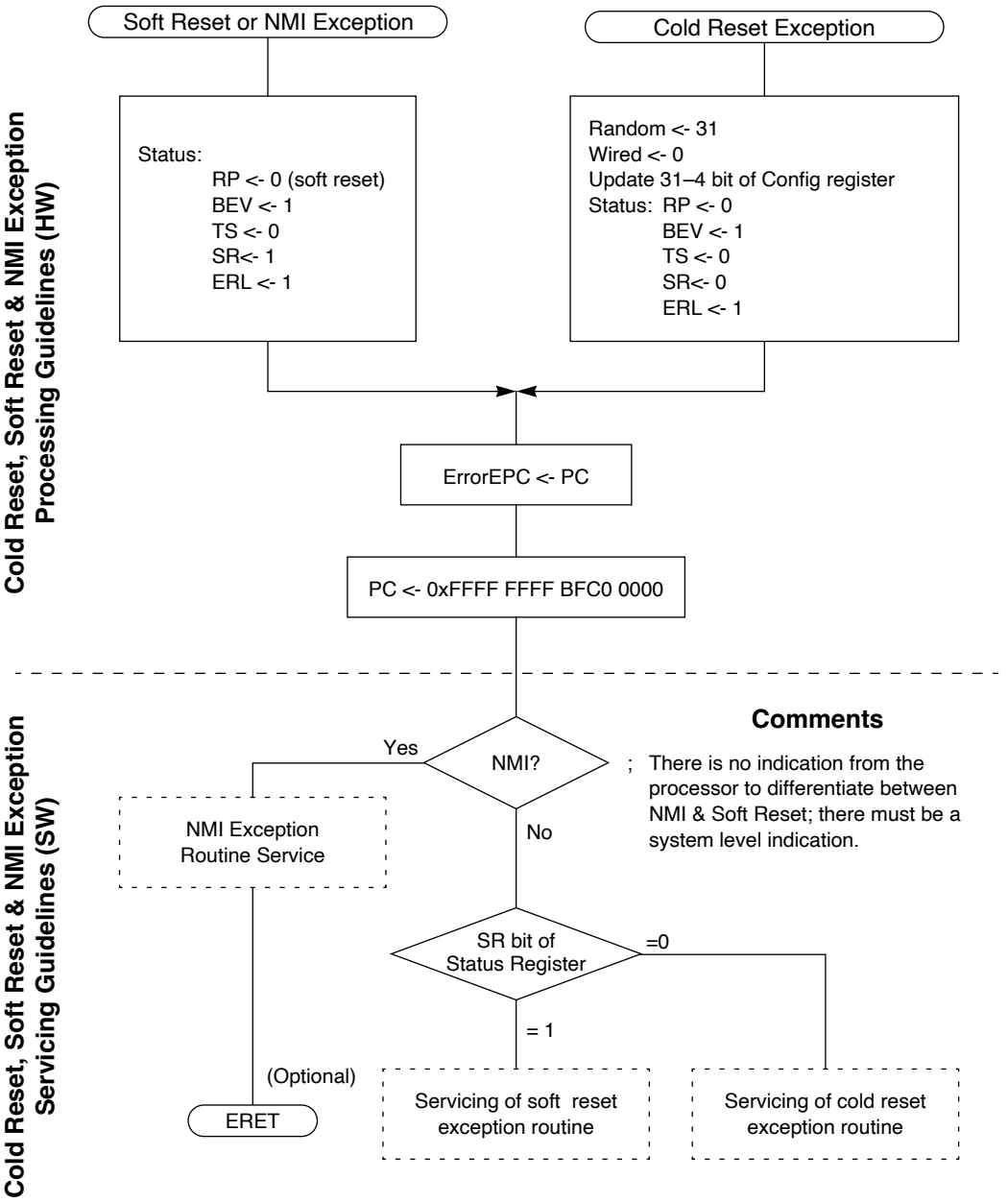


Figure 6-16 Cold Reset, Soft Reset & NMI Exception Handler

[MEMO]

Floating-Point Operations

7

7.1 Overview

All floating-point instructions, as defined in the MIPS ISA for the floating-point coprocessor, CP1, can be processed by the V_R4300. Logically, the Floating-Point Arithmetic Unit (FPU) exists as an individual coprocessor; however, unlike those of the V_R4400, the V_R4300 FPU is physically integrated into the Integer Arithmetic Unit (CPU). The CPU and the FPU use a common datapath and FPU instructions are fully-implemented in the CPU hardware. Unlike the V_R4400 implementation, V_R4300 integer instructions cannot be executed until a multicycle floating-point instruction has been completed.

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

7.2 FPU Programming Model

This section describes the structure of the registers, memory, and data, and usable *General Purpose* registers. Moreover, the *FPU* registers are described in detail.

7.2.1 Floating-Point General Purpose Register (FGR)

The FPU has one set of floating-point general purpose register (FGR) and two *Control* registers (*Control/Status* register: FCR31, *Implementation/Revision* register: FCR0). The general purpose register can be used in the following three ways.

- As 32 *General Purpose* registers (32 FGRs), each of which is 32 bits wide when the *FR* bit in the *Status* register equals 0; or as 32 *General Purpose* registers (32 FGRs), each of which is 64-bits wide when *FR* equals 1. The CPU accesses these registers through load, store, and transfer instructions.
- As 16 floating-point registers (FPR) (see the next section for a description of FPRs), each of which is 64-bits wide, when the *FR* bit in the *Status* register equals 0. The FPRs hold values in either single- or double-precision floating-point format. Each FPR corresponds to adjacently numbered FGRs as shown in Figure 7-1.
- As 32 floating-point registers (FPR) (see the next section for a description of FPRs), each of which is 64-bits wide, when the *FR* bit in the *Status* register equals 1. The FPRs hold values in either single- or double-precision floating-point format. Each FPR corresponds to an individual FGR as shown in Figure 7-1.

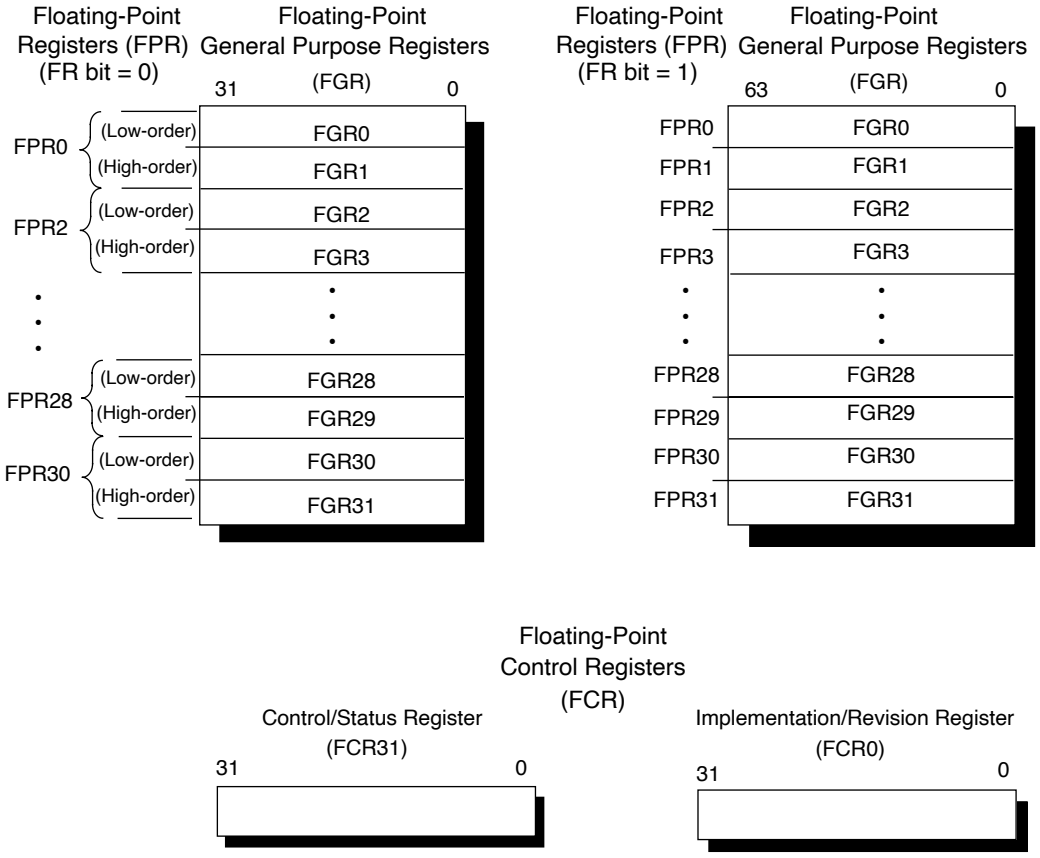


Figure 7-1 FPU Registers

7.2.2 Floating-Point Registers (FPR)

CP1 provides:

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

FPR possesses logical 64-bit registers, holds floating-point values during floating-point operations, and is physically formed from the *General Purpose* registers (*FGR*s). *FPR* can be accessed through a Floating-Point Arithmetic Instruction. *FPR* is physically configured with *General Purpose* registers (*FGR*s). When the *FR* bit in the *Status* register equals 0, the *FPR* is configured with two 32-bit *FGR*s. When the *FR* bit in the *Status* register equals 1, the *FPR* is configured with a single 64-bit *FGR*.

The *FPR*s hold values in either single- or double-precision floating-point format. If the *FR* bit equals 0, only even numbers (the *least* register, as shown in Figure 7-1) can be used to address *FPR*s. When the *FR* bit equals 1, all *FPR* register numbers are valid. If the *FR* bit equals 0 during a double-precision floating-point operation, the *FGR* can be used in double pairs. Thus, in a double-precision operation, selecting *Floating-Point Register 0* (*FPR0*) actually uses adjacent *Floating-Point General Purpose* registers *FGR0* and *FGR1*.

7.2.3 Floating-Point Control Registers (FCRs)

The FPU in the V_R4000 Series (excluding V_R4100) has 32 control registers. With the V_R4300, the following two FCRs are valid.

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

Table 7-1 lists the assignments of the *FCR*s.

Table 7-1 Floating-Point Control Register Assignments

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

7.2.4 Control/Status Register (FCR31)

The *Control/Status* register (*FCR31*) is a read/write register, and holds control data and status data. *FCR31* controls the rounding mode and enables occurrence of the floating-point exception. It also indicates the information on the exception that has caused by the instruction executed last and information on the exceptions that have been masked and therefore have not occurred. Figure 7-2 shows the configuration of *FCR31*.

Control/Status Register (FCR31)

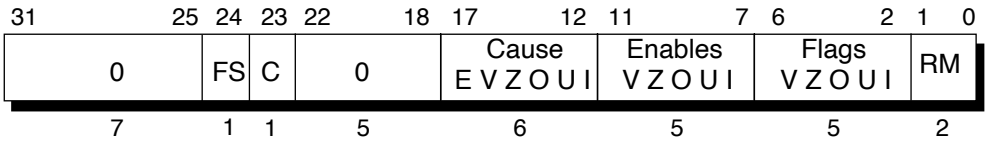


Figure 7-2 Control/Status Register Bit Assignments

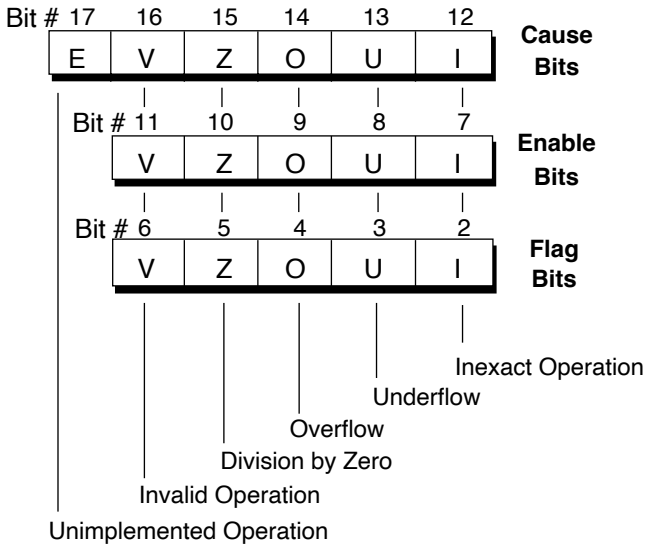


Figure 7-3 Control/Status Register (FCR31) Cause, Enable, and Flag Bit Fields

The contents of *FCR31* and *FCR0* can be read by using the CFC1 instruction.

The bits of *FCR31* can be set or cleared by using the CTC1 instruction. *FCR0* is a read-only register. The contents of a register to which data is to be written are undefined when an instruction that immediately follows the instruction that writes data to the register is executed. The pipeline does not interlock.

The IEEE754 specifies detection of an exception during a floating-point operation, setting flags, and calling an exception handler in case of an exception. With the MIPS architecture, these specifications are realized by the cause, enable, and flag bits of the *Control/Status* register. The flag bit conforms to the exception status flag of the IEEE754, and the cause and enable bits conform to the exception handler of the IEEE754.

Each bit of *FCR31* is described next.

FS bit

The *FS* bit enables a value that cannot be normalized (denormalized number) to be flashed. When the *FS* bit is set and the enable bit is not set for the underflow exception and illegal exception, the result of the denormalized number does not cause the unimplemented operation exception, but is flushed. Whether the flushed result is 0 or the minimum normalized value is determined depending on the rounding mode (refer to **Table 7-2**). If the result is flushed, the *Flag* and *Cause* bits are set for the underflow and illegal exceptions.

Table 7-2 Flush Values of Denormalized Number Results

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

C Bit

When a floating-point Compare operation takes place, the result is stored at bit 23, the *Condition* bit. The *C* bit is set to 1 if the condition is true; the bit is cleared to 0 if the condition is false. Bit 23 is affected only by compare and CTC1 instructions.

Cause, Flag, and Enable Fields

Figure 7-3 illustrates the *Cause*, *Enable*, and *Flag* fields of the *FCR31*.

The *Cause* and *Flag* fields are updated by all conversion, computational (except MOV.fmt), CTC1, reserved, and unimplemented operation instructions. All other instructions have no affect on these fields.

Cause Bits

Bits 17:12 in the *FCR31* contain *Cause* bits which reflect the results of the most recently executed floating-point instruction. The *Cause* bits are a logical extension of the CP0 *Cause* register; they identify the exceptions raised by the last floating-point operation; and generate exceptions if the corresponding *Enable* bit is set. If more than one exception occurs on a single instruction, each appropriate bit is set.

The *Cause* bits are updated by the floating-point operations (except load, store, and transfer instructions). The unimplemented operation instruction (*E*) bit is set to a 1 if software emulation is required, otherwise it remains 0. The other bits are set to 0 or 1 to indicate the occurrence or non-occurrence (respectively) of an IEEE754 exception.

If the floating-point operation exception occurs, the operation result is not stored, and only the *Cause* bit is influenced. The type of the exception that has been caused by the most-recently-executed floating-point operation can be identified by reading the *Cause* bit.

Enable Bits

A floating-point exception is generated any time a *Cause* bit and the corresponding *Enable* bit are set. As soon as the *Cause* bit enabled through the Floating-point operation, an exception occurs. When both *Cause* and *Enable* bits are set by the CTC1 instruction, an exception also occurs.

There is no enable bit for unimplemented operation instruction (*E*). An Unimplemented exception always generates a floating-point exception.

Before returning from a floating-point exception, software must first clear the *Cause* bits that are enabled to generate exceptions to prevent a repeat of exceptions. Thus, User mode programs cannot observe the set *Cause* bits. To use the information by the handler in *User* mode, save the value of the *Status* register and then call the handler in *User* mode.

If the *Cause* bit is set but the corresponding *Enable* is not set, no floating-point exception occurs and the default result defined by IEEE754 is stored. In this case, whether the exceptions were caused by the immediately previous floating-point operation can be determined by reading the *Cause* bit.

Flag Bits

The *Flag* bits are cumulative and indicate the exceptions that were raised after reset. *Flag* bits are set to 1 if an IEEE754 exception is raised but the occurrence of the exception is prohibited. Otherwise, they remain unchanged. The *Flag* bits are never cleared as a side effect of floating-point operations; however, they can be set or cleared by writing a new value into the *FCR31*, using a CTC1 instruction.

Rounding Mode Control Bits

Bits 1 and 0 in the *FCR31* register constitute the *Rounding Mode* (*RM*) bits. These bits specify the rounding mode that FPU uses for all floating-point operations.

Table 7-3 Rounding Mode Control Bits

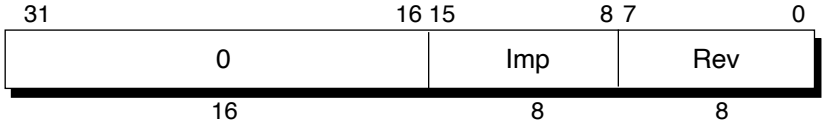
RM bits		Mnemonic	Description
Bit 1	Bit 0		
0	0	RN	Round result to nearest representable value; round to value with least-significant bit 0 when the two nearest representable values are equally near.
0	1	RZ	Round toward 0: round to value closest to and not greater in magnitude than the infinitely precise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

7.2.5 Implementation/Revision Register (FCR0)

The *Implementation/Revision* register (FCR0) is a read-only register and holds the implementation identification number and implementation revision number of the FPU. This information is used to revise the coprocessor, determine the performance level, and to execute self-diagnosis.

Figure 7-4 shows the layout of the register.

Implementation/Revision Register (FCR0)



Imp : Implementation number (0x0B)
Rev : Revision number in the form of y.x
0 : RFU. Returns zeroes when read.

Figure 7-4 Implementation/Revision Register

The implementation revision number is a value in the format of y.x, where y is the major revision number stored to the bits 7:4, and x is the minor revision number stored to bits 3:0. Revision of the chip can be identified by the implementation revision number. However, the fact that a chip has been changed is not always reflected on the revision number. Conversely, a change in the revision number does not always reflect an actual change of the chip. Therefore, design the program so that it does not depend on the revision number of this register.

7.3 Floating-Point Formats

The FPU supports the performances of both 32-bit (single-precision) and 64-bit (double-precision) IEEE754 standard floating-point operations. The 32-bit single-precision format has a 24-bit signed fraction field (*s+f*) and an 8-bit exponent (*e*), as shown in Figure 7-5.

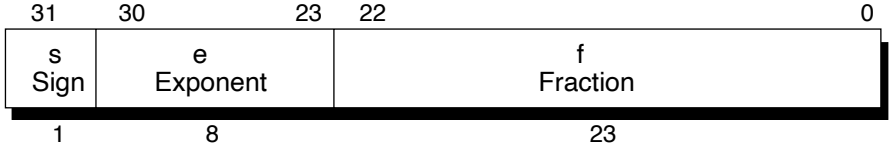


Figure 7-5 Single-Precision Floating-Point Format

The double-precision format has a 53-bit signed fraction field (*s+f)* and an 11-bit exponent, as shown in Figure 7-6.

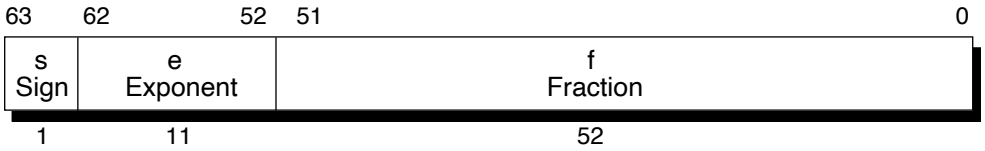


Figure 7-6 Double-Precision Floating-Point Format

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

- sign field, *s*
- exponent, $e = E + bias$
- fraction, $f = b_1 b_2 ... b_{P-1}$ (value at first decimal place or beyond)

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

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

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

For single- and double-precision formats, the value of a number, *v*, is determined by the equations shown in Table 7-4.

Table 7-4 Equations for Calculating Values in Single-and Double-Precision Floating-Point Format

No.	Equation
NaN (Not a Number)	if $E = E_{max}+1$ and $f \neq 0$, then v is NaN, regardless of s
± ∞ (Infinite number)	if $E = E_{max}+1$ and $f = 0$, then $v = (-1)^s \infty$
Normalized number	if $E_{min} \leq E \leq E_{max}$, then $v = (-1)^s 2^E (1.f)$
Denormalized number	if $E = E_{min}-1$ and $f \neq 0$, then $v = (-1)^s 2^{Emin} (0.f)$
±0 (Zero)	if $E = E_{min}-1$ and $f = 0$, then $v = (-1)^s 0$

NaN (Not a Number)

The IEEE754 specifies a floating-point value called NaN (Not a Number). This is not a numeric value and therefore, is not greater or smaller than anything.

For all floating-point formats, if *v* is NaN, the most-significant bit of *f* determines whether the value is a signaling or quiet NaN: *v* is a signaling NaN if the most-significant bit of *f* is set, otherwise, *v* is a quiet NaN. Table 7-5 defines the values for the format parameters.

Table 7-5 Floating-Point Format Parameter Values

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

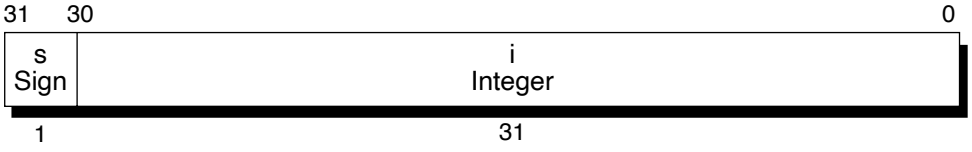
The minimum and maximum values that can be expressed in this floating-point format are shown in Table 7-6.

Table 7-6 Minimum and Maximum Floating-Point Values

Type	Value
Single-precision floating-point Minimum	$1.40129846e^{-45}$
Single-precision floating-point Minimum (Normal)	$1.17549435e^{-38}$
Single-precision floating-point Maximum	$3.40282347e^{+38}$
Double-precision floating-point Minimum	$4.9406564584124654e^{-324}$
Double-precision floating-point Minimum (Normal)	$2.2250738585072014e^{-308}$
Double-precision floating-point Maximum	$1.7976931348623157e^{+308}$

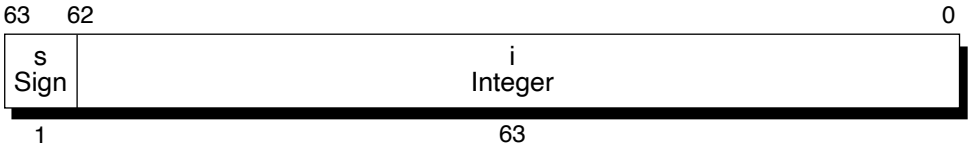
7.4 Fixed-Point Format

Fixed-point values are held in 2's complement format. Unsigned fixed-point values are not directly provided by the floating-point instruction set. Figure 7-7 illustrates 32-bit fixed-point format and Figure 7-8 illustrates 64-bit fixed-point format.



s : sign bit
i : integer value (2's complement)

Figure 7-7 32-Bit Fixed-Point Format



s : sign bit
i : integer value (2's complement)

Figure 7-8 64-Bit Fixed-Point Format

7.5 FPU Set Overview

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

- **Load/Store/Transfer** instructions move data between the FPU *General Purpose* register, *Control* register, CPU, and memory.
- **Conversion** instructions perform conversion operations between the various data formats.
- **Computational** instructions perform arithmetic operations on floating-point values in *FPU* registers.
- **Compare** instructions perform comparisons of the contents of registers and set the results to a *condition* bit of the *FCR31*.
- **FPU Branch** instructions perform a branch to the specified target if the specified coprocessor condition is met.

For details of each instruction, refer to **Chapter 17 FPU Instruction Set Details**.

7.5.1 Floating-Point Load/Store/Transfer Instructions

Loads/Stores from/to CP1 and Memory

Loads/Stores from/to CP1 and memory are accomplished by using one of the following instructions:

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

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

Transfers Between CP1 and CPU

Data can also be moved directly between *CP1 General Purpose* registers and the CPU by using one of the following instructions:

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

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

Data transfer between *CP1 control* registers and the CPU is accomplished with the following instructions:

- Move Control Word To Coprocessor 1 (CTC1)
- Move Control Word From Coprocessor 1 (CFC1)

Load Delay and Hardware Interlocks

The instruction immediately following a load or a MTC1 can use the contents of the loaded register. In such cases the hardware interlocks, requiring additional real cycles; for this reason, scheduling load delay slots is desirable to avoid the interlocks.

Data Alignment

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

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

Endianness

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

Table 7-7 lists load, store, and transfer instructions.

Table 7-7 Load/Store/Transfer Instructions

Instruction	Format and Description	op	base	ft	offset
Load Word To FPU	LWC1 ft, offset (base) Sign-extends the 16-bit offset and adds it to the CPU register base to generate an address. Loads the contents of the word specified by the address to the FPU general purpose register ft.				
Store Word From FPU	SWC1 ft, offset (base) Sign-extends the 16-bit offset and adds it to the CPU register base to generate an address. Stores the contents of the FPU general purpose register ft to the memory position specified by the address.				
Load Doubleword To FPU	LDC1 ft, offset (base) Sign-extends the 16-bit offset and adds it to the CPU register base to generate an address. Loads the contents of the doubleword specified by the address to the FPU general purpose registers ft and ft+1 when FR = 0, or to the FPU general purpose register ft when FR = 1.				
Store Doubleword From FPU	SDC1 ft, offset (base) Sign-extends the 16-bit offset and adds it to the CPU register base to generate an address. Stores the contents of the FPU general purpose registers ft and ft+1 to the memory position specified by the address when FR = 0, and the contents of the FPU general purpose register ft when FR = 1.				

Instruction	Format and Description	COP1	sub	rt	fs	0
Move Word To FPU	MTC1 rt, fs Transfers the contents of CPU general purpose register rt to FPU general purpose register fs.					
Move Word From FPU	MFC1 rt, ft Transfers the contents of FPU general purpose register fs to CPU general purpose register rt.					
Move Control Word To FPU	CTC1 rt, fs Transfers the contents of CPU general purpose register rt to FPU control register fs.					
Move Control Word From FPU	CFC1 rt, fs Transfers the contents of FPU control register fs to CPU general purpose register rt.					
Doubleword Move To FPU	DMTC1 rt, fs Transfers the contents of CPU general purpose register rt to FPU general purpose register fs.					
Doubleword Move From FPU	DMFC1 rt, fs Transfers the contents of FPU general purpose register fs to CPU general purpose register rt.					

7.5.2 Convert Instructions

Convert instructions perform conversions between the various data formats such as single- or double-precision, fixed- or floating-point formats. Table 7-8 lists conversion instructions.

When converting a long integer to a single- or double-precision floating-point number (CVT. [S,D]. L), bits 63:55 of the 64-bit integer must be all zeroes or ones, otherwise the V_R4300 processor raises a floating-point instruction exception. The floating-point instruction exception allows these cases to be handled by software.

Table 7-8 Convert Instruction (1/2)

Instruction	Format and Description	COP1	fmt	0	fs	fd	funct
Floating-point Convert To Single Floating-point Format	CVT.S.fmt fd, fs Converts the contents of floating-point register fs from the specified format (fmt) to a single-precision floating-point format. Stores the rounded result to floating-point register fd.						
Floating-point Convert To Double Floating-point Format	CVT.D.fmt fd, fs Converts the contents of floating-point register fs from the specified format (fmt) to a double-precision floating-point format. Stores the rounded result to floating-point register fd.						
Floating-point Convert To Long Fixed-point Format	CVT.L.fmt fd, fs Converts the contents of floating-point register fs from the specified format (fmt) to a 64-bit fixed-point format. Stores the rounded result to floating-point register fd.						
Floating-point Convert To Single Fixed-point Format	CVT.W.fmt fd, fs Converts the contents of floating-point register fs from the specified format (fmt) to a 32-bit fixed-point format. Stores the rounded result to floating-point register fd.						
Floating-point Round To Long Fixed-point Format	ROUND.L.fmt fd, fs Rounds the contents of floating-point register fs to a value closest to the 64-bit fixed-point format and converts them from the specified format (fmt). Stores the result to floating-point register fd.						
Floating-point Round To Single Fixed-point Format	ROUND.W.fmt fd, fs Rounds the contents of floating-point register fs to a value closest to the 32-bit fixed-point format and converts them from the specified format (fmt). Stores the result to floating-point register fd.						
Floating-point Truncate To Long Fixed-point Format	TRUNC.L.fmt fd, fs Rounds the contents of floating-point register fs toward 0 and converts them from the specified format (fmt) to a 64-bit fixed-point format. Stores the result to floating-point register fd.						

Table 7-8 Convert Instruction (2/2)

Instruction	Format and Description	COP1	fmt	0	fs	fd	funct
Floating-point Truncate To Single Fixed-point Format	TRUNC.W.fmt fd, fs Rounds the contents of floating-point register fs toward 0 and converts them from the specified format (fmt) to a 32-bit fixed-point format. Stores the result to floating-point register fd.						
Floating-point Ceiling To Long Fixed-point Format	CEIL.L.fmt fd,fs Rounds the contents of floating-point register fs toward +∞ and converts them from the specified format (fmt) to a 64-bit fixed-point format. Stores the result to floating-point register fd.						
Floating-point Ceiling To Single Fixed-point Format	CEIL.W.fmt fd,fs Rounds the contents of floating-point register fs toward +∞ and converts them from the specified format (fmt) to a 32-bit fixed-point format. Stores the result to floating-point register fd.						
Floating-point Floor To Long Fixed-point Format	FLOOR.L.fmt fd, fs Rounds the contents of floating-point register fs toward -∞ and converts them from the specified format (fmt) to a 64-bit fixed-point format. Stores the result to floating-point register fd.						
Floating-point Floor To Single Fixed-point Format	FLOOR.W.fmt fd, fs Rounds the contents of floating-point register fs toward -∞ and converts them from the specified format (fmt) to a 32-bit fixed-point format. Stores the result to floating-point register fd.						

7.5.3 Computational Instructions

Computational instructions perform arithmetic operations on floating-point values, in registers. Table 7-9 lists the computational instructions. There are two categories of computational instructions:

- 3-Operand Register-Type instructions, which perform floating-point add, subtract, multiply, and divide operations
- 2-Operand Register-Type instructions, which perform floating-point absolute value, transfer, square root, and negate operations.

Table 7-9 Computational Instructions

Instruction	Format and Description	COP1	fmt	ft	fs	fd	funct
Floating-point Add	ADD.fmt fd, fs, ft Arithmetically adds the contents of floating-point registers fs and ft in the specified format (fmt). Stores the rounded result to floating-point register fd.						
Floating-point Subtract	SUB.fmt fd, fs, ft Arithmetically subtracts the contents of floating-point registers fs and ft in the specified format (fmt). Stores the rounded result to floating-point register fd.						
Floating-point Multiply	MUL.fmt fd, fs, ft Arithmetically multiplies the contents of floating-point registers fs and ft in the specified format (fmt). Stores the rounded result to floating-point register fd.						
Floating-point Divide	DIV.fmt fd, fs, ft Arithmetically divides the contents of floating-point registers fs and ft in the specified format (fmt). Stores the rounded result to floating-point register fd.						
Floating-point Absolute Value	ABS.fmt fd, fs Calculates the arithmetic absolute value of the contents of floating-point register fs in the specified format (fmt). Stores the result to floating-point register fd.						
Floating-point Move	MOV.fmt fd, fs Copies the contents of floating-point register fs to floating-point register fd in the specified format (fmt).						
Floating-point Negate	NEG.fmt fd, fs Arithmetically negates the contents of floating-point register fs in the specified format (fmt). Stores the result to floating-point register fd.						
Floating-point Square Root	SQRT.fmt fd, fs Calculates arithmetic positive square root of the contents of floating-point register fs in the specified format. Stores the rounded result to floating-point register fd.						

fmt appended to the instruction op code of the arithmetic operation and compare instruction indicates the data format. S indicates the single-precision floating decimal point, D indicates the double-precision floating decimal point, L indicates the 64-bit fixed decimal point, and W indicates the 32-bit fixed decimal point. For example, "ADD.D" means that the operand of the addition instruction is a double-precision floating-point value.

If the *FR* bit is 0, an odd-numbered register cannot be specified.

7.5.4 Compare Instructions

The floating-point compare (C.cond.fmt) instructions interpret the contents of two FPU registers (*fs, ft*) in the specified format (*fmt*) and arithmetically compare them. A result is determined based on the comparison and conditions (*cond*) specified in the instruction. Table 7-10 lists the compare instructions. Table 7-11 lists the mnemonics for the compare instruction conditions.

Table 7-10 Compare Instruction

Instruction	Format and Description	COP1	fmt	ft	fs	0	funct
Floating-point Compare	C.cond.fmt fs, ft Interprets and arithmetically compares the contents of FPU registers fs and ft in the specified format (fmt). The result is identified by comparison and the specified condition (cond). After a delay of one instruction, the comparison result can be used by the FPU branch instruction of the CPU.						

Table 7-11 Mnemonics and Definitions of Compare Instruction Conditions

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

7.5.5 FPU Branch Instructions

Table 7-12 lists the FPU branch instructions. These instructions can be used to test the result of the compare (C.cond.fmt) instruction. The delay slot in this table indicates the instruction that immediately follows a branch instruction. For details, refer to **Chapter 4 Pipeline**.

Table 7-12 FPU Branch Instructions

Instruction	Format and Description: COP1 | BC | br | offset
Branch On FPU True	BC1T offset Adds the instruction address in the delay slot and a 16-bit offset (shifted 2 bits to the left and sign-extended) to calculate the branch target address. If the FPU condition line is true, branches to the target address (delay of one instruction).
Branch On FPU False	BC1F offset Adds the instruction address in the delay slot and a 16-bit offset (shifted 2 bits to the left and sign-extended) to calculate the branch target address. If the FPU condition line is false, branches to the target address (delay of one instruction).
Branch On FPU True Likely	BC1TL offset Adds the instruction address in the delay slot and a 16-bit offset (shifted 2 bits to the left and sign-extended) to calculate the branch target address. If the FPU condition line is true, branches to the target address (delay of one instruction). If conditional branch does not take place, the instruction in the delay slot is invalidated.
Branch On FPU False Likely	BC1FL offset Adds the instruction address in the delay slot and a 16-bit offset (shifted 2 bits to the left and sign-extended) to calculate the branch target address. If the FPU condition line is false, branches to the target address (delay of one instruction). If conditional branch does not take place, the instruction in the delay slot is invalidated.

7.5.6 FPU Instruction Execution Time

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

All data transfer between the floating-point and memory is accomplished by coprocessor load and store operations. Data may be directly moved between the floating-point coprocessor and the integer processor by load to and load from coprocessor instructions as shown below:

Table 7-13 Number of Load/Store/Transfer Instruction Execution Cycles

Instruction	Cycles
LWC1	2/1*
SWC1	1
LDC1	2/1*
SDC1	1
MTC1	1
MFC1	1
DMTC1	1
DMFC1	1
CTC1	1
CFC1	1

* The hardware interlocks for one cycle if the load result is used by the instruction in the load delay slot.

To obtain optimum performance, the V_R4300 pipeline does not perform a bypass from EX to EX stage of the next instruction for the floating-point result of a compare, computational, LWC1, or LDC1 instruction. If the subsequent EX-stage floating-point instruction depends on the result of the current EX-stage floating-point instruction, the current floating-point instruction completes and its EX-stage result is registered in the DC stage and the bypass is enabled. Meanwhile, the RF-stage floating-point instruction advances to the EX-stage, where it is stalled for one pipeline clock to wait for the result to be bypassed from DC to EX, before it begins execution.

Caution **This limitation on bypass from EX to EX stage of the next instruction does not apply to integer operations nor to floating-point load/store/transfer instructions (except LWC1 and LDC1).**

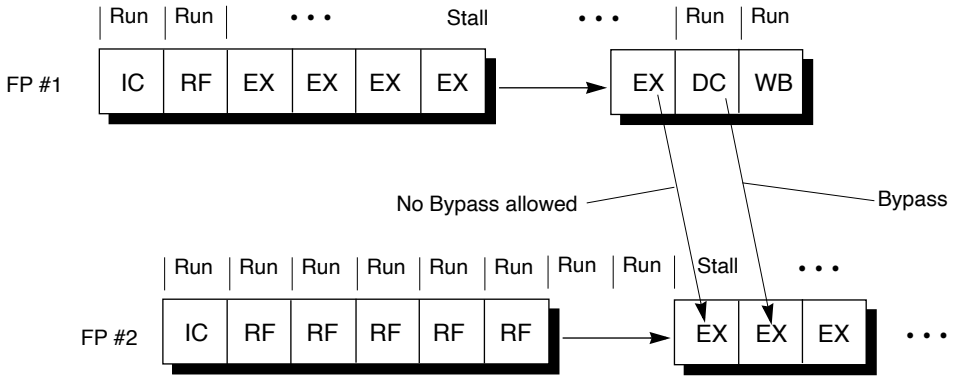


Figure 7-9 DC-to-EX Hardware Interlock Bypass

The execution unit of the V_R4300 can shorten the delay time of almost all the floating-point instructions depending on the circumstances. By using this feature, the performance can be improved and design can be simplified. Changes in the delay time are simplified as much as possible. If occurrence of an exception is detected by checking the source operand when a multicycle instruction is executed (if a source exception occurs), this multicycle instruction is executed for only 2 cycles, and exception processing is started. Similarly, if the result of an operation is found to be the value that does not cause an exception (zero or infinite) as a result of checking the operand, the result (e.g., a value other than $\infty \times 0$) is written back 2 cycles after, and the operation ends.

Floating-point exceptions, except the source exception, are not aborted until instruction execution is completed. In other words, an exception is reported not when it has been found, but when instruction execution has been completed.

Next, the execution time of each instruction is described.

Floating-point Add/Subtract Instructions

Floating point add and subtract terminate on the second cycle if a source exception occurs, or if at least one operand is zero or infinity. The instruction completes on the third cycle in all other cases.

Floating-point Multiply Instruction

A floating point multiply completes in two cycles if a source exception is detected, or if, during the first cycle, the result can be determined to be zero or infinity. A floating-point multiply also finishes in the second cycle if at least one of the operands is a power of 2. In all other cases it takes the full number (the maximum specified for each format) of cycles to complete. Thus, multiply does not finish as soon as the remaining bits are zero. Also, there can be no overlap between multiply and add.

Floating-point Divide/Square Root Instructions

Floating Point divide and square root complete in the second cycle on either a source exception or if, during the first cycle, the result can be determined to be either zero or infinity. Otherwise they continue, taking the maximum amount of cycles.

Floating-point Convert Instruction

Floating-point convert instructions also complete in the second cycle for trivial cases.

Execution cycle numbers of floating-point instructions are listed in Table 7-14. If a floating-point result for these instructions is needed by the subsequent instruction, the latency is the execution rate plus one, due to the fact that an EX-to-RF bypass is not performed for the results of these instructions. All CPU/FPU instruction delay times that are not mentioned in these tables have a latency of one pipeline clock cycle (1PClock).

7.6 FPU Pipeline Synchronization

Since the integer and floating-point units share a common hardware pipeline, a CFC1 instruction is not needed to synchronize the pipeline operation.

Table 7-14 Number of FPU Instruction Delay Cycles[*1]

Instruction	Pipeline Cycles [*2]			
	S	***D***	***W***	***L***
Add.fmt	3	3		
Sub.fmt	3	3		
Mul.fmt	5	8		
Div.fmt	29	58		
Sqrt.fmt	29	58		
Abs.fmt	1	1		
Mov.fmt	1	1		
Neg.fmt	1	1		
Round.W.fmt	5	5		
Trunc.W.fmt	5	5		
Ceil.W.fmt	5	5		
Floor.W.fmt	5	5		
Round.L.fmt	5	5		
Trunc.L.fmt	5	5		
Ceil.L.fmt	5	5		
Floor.L.fmt	5	5		
Cvt.S.fmt	-	2	5	5
Cvt.D.fmt	1	-	5	5
Cvt.W.fmt	5	5		
Cvt.L.fmt	5	5		
C.cond.fmt	1	1		
BC1T[*3]	1			
BC1F[*3]	1			
BC1TL[*3]	1			
BC1FL[*3]	1			

*1. If the result of a floating-point instruction is needed by the subsequent instruction, one additional pipeline clock is required to perform a hardware interlock bypass.

*2. The multicycle floating-point operation instructions whose results are obvious are not described in this table; it takes two pipeline clocks to complete.

*3. The architecturally defined branch delay slot of one cycle also applies to all FPU branch instructions.

[MEMO]

Floating-Point Exceptions

8

This chapter explains how the FPU handles the floating-point exception.

8.1 Types of Exceptions

The floating-point exception occurs if a floating-point operation or the result of the operation cannot be handled by the ordinary method.

The FPU performs either of the following two operations in case of an exception.

- **When exception is enabled**
  Sets the *Cause* bit of the *Control/Status* register (*FCR31*) of the FPU, and transfers servicing to the exception handler routine (software servicing).
- **When exception is disabled**
  Stores an appropriate value (default value) to the *Destination* register of the FPU, sets the *Cause* bit and flag bit of *FCR31*, and continues execution.

The FPU supports the five IEEE754 exceptions:

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

Cause bits, *Enable* bits, and *Flag* bits (*Status* flags) are used.

FPU has an unimplemented operation (E) as the sixth exception cause, which is used when the floating-point operation cannot be executed with the standard MIPS architecture (including when the FPU cannot correctly process exceptions). This exception requires service by the software. The *E* bit does not exit in the *Enable* or *Flag* bit. When this exception occurs, unimplemented exception processing is executed (when interrupt input by the FPU to the CPU is enabled).

Figure 8-1 shows the bits of the *FCR31* used to support the exception.

Remark The unimplemented operation exception is defined by the IEEE754 standard. With the V_R4300, however, this is an exception that occurs if an operation not supported by the hardware is executed.

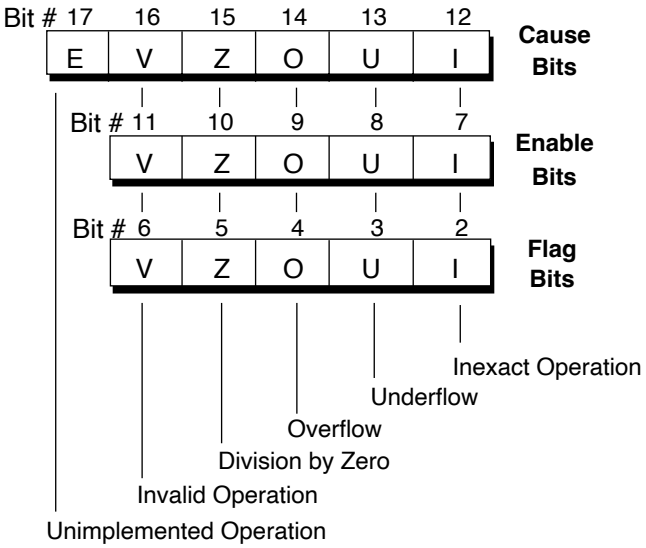


Figure 8-1 FCR31 Cause/Enable/Flag Bits

The five exceptions (V, Z, O, U, and I) of the IEEE754 are enabled when the *Enable* bit is set. When an exception occurs, the corresponding *Cause* bit is set. If the corresponding *Enable* bit is set, the FPU generates an interrupt to the CPU, and starts exception processing. If occurrence of the exception is disabled, the *Cause* and *Flag* bits corresponding to the exception are set.

8.2 Exception Processing

When a floating-point exception is taken, the *Cause* register of the CP0 indicates the FPU is the cause of the exception. The Floating-Point Exception (FPE) code is used, and the *Cause* bits of the *FCR31* indicate the reason for the floating-point exception. These bits are, in effect, an extension of the CP0 *Cause* register.

8.2.1 Flags

Flag bits corresponding to the respective IEEE754 exceptions are provided. The *Flag* bit is set when occurrence of the corresponding exception is disabled and when the condition of the exception is detected. The flag bit can be reset by writing a new value to the *Status* register by using the CTC1 instruction.

If an exception is disabled by the corresponding *Enable* bit, the FPU performs predetermined processing. This processing gives the default value as the result, instead of the result of the floating-point operation. This default value is determined by the type of the exception. In the case of the overflow and underflow exceptions, the default value differs depending on the rounding mode used at that time. Table 8-1 shows the default values to be given by the respective IEEE754 exceptions of the FPU.

Table 8-1 Default FPU IEEE754 Exception Values

Field	Description	Rounding Mode	Default Values
V	Invalid operation	–	Supply a Quiet Not a Number (Q-NaN)
Z	Division by zero	–	Supply a properly signed ∞
O	Overflow	RN	∞ signed with intermediate result
		RZ	Maximum normal number signed with intermediate result
		RP	Negative overflow: maximum negative normal number Positive overflow: +∞
		RM	Positive overflow: maximum positive normal number Negative overflow: -∞
U	Underflow	RN	0 signed with intermediate result
		RZ	0 signed with intermediate result
		RP	Positive underflow: minimum positive normal number Negative underflow: 0
		RM	Negative underflow: minimum negative normal number Positive underflow: 0
I	Inexact exception	–	Supply a rounded result

The FPU detects the nine exception causes internally. When the FPU detects one of these unusual situations, it causes either an IEEE754 exception or an unimplemented operation exception (E). Table 8-2 lists the exception-causing situations and compares the contents of the *Cause* bits of the FPU with the IEEE754 standard when each exception occurs.

Table 8-2 FPU Internal Results and Flag Status

FPU Internal Result	IEEE754	Exception Enable	Exception Disable	Remarks
Inexact result	I	I	I	Loss of accuracy
Exponent overflow	O,I*1	O,I	O,I	Normalized exponent > E_{max}
Division by zero	Z	Z	Z	Zero is (exponent = E_{min}-1, mantissa = 0)
Overflow on convert to integer	V	E	E	Source out of integer range
Signaling NaN (S-NaN) source	V	V	V	
Invalid operation	V	V	V*2	0/0, etc.
Exponent underflow	U	E	U, I	Normalized exponent < E_{min}
Denormalized source	None	E	E	Exponent = E_{min}-1 and mantissa ≠ 0
Q-NaN	None	E	E	

*1. With the IEEE754, the inexact operation exception occurs only if an overflow occurs only when the overflow exception is disabled. However, the V_R4300 always generates the overflow exception and inexact operation exception when an overflow occurs.

*2. If both the underflow exception and inexact operation exception are disabled when the exponent underflow occurs, and if the *FS* bit of *FCR31* is set, the *Cause* bit and *Flag* bit of the underflow exception and inexact operation exception are set. Otherwise, the *Cause* bit of the unimplemented operation exception is set.

Next, each FPU exception is described.

8.2.2 Inexact Exception (I)

The FPU generates the inexact operation exception in the following cases.

- If the accuracy of the rounded result drops
- If the rounded result overflows
- If the rounded result underflows and if the *FS* bit of *FCR31* is set with the underflow and illegal operation exceptions disabled

If Exception Is Enabled:

The *Destination* register is not modified, the *Source* registers are preserved and an Inexact Operation exception occurs.

If Exception Is Not Enabled:

The rounded result or underflowed/overflowed result is delivered to the *Destination* register if no other exception occurs.

8.2.3 Invalid Operation Exception (V)

The Invalid Operation exception is generated if one or both of the operands are invalid. When the exception is not enabled, the MIPS ISA defines the result as a Quiet Not a Number (Q-NaN). The invalid operations are:

- Add or subtract: Add and Subtract of infinities, such as: $(+\infty) + (-\infty)$ or $(-\infty) - (-\infty)$
- Multiply: $\pm 0 \times \pm \infty$
- Divide: $\pm 0 \div \pm 0$, or $\pm \infty \div \pm \infty$
- Compare of predicates involving **<** or **>** without **?**, when the operands are unordered
- Any arithmetic operation, when one or both operands is a S-NaN. A transfer (MOV) operation is not considered to be an arithmetic operation, but absolute value (ABS) and negate (NEG) are.
- Compare or convert to floating-point operation when the operand is S-NaN.
- Square root: $\sqrt{x}$, where x is less than zero.

Software can simulate the Invalid Operation exception for other operations that are invalid for the given source operands. Examples of these operations include IEEE754-specified functions implemented in software, such as Remainder *x* REM *y*, where *y* is 0 or *x* is infinite; conversion of a floating-point number to a decimal format whose value causes an overflow, is infinity, or is NaN; and transcendental functions, such as ln (–5) or $\cos^{-1}(3)$. Refer to **Chapter 17 FPU Instruction Set Details**. Refer to **Appendix B** for examples or for routines to handle these cases.

If Exception Is Enabled:

The Destination register is not modified, the *Source* registers are preserved, and the Invalid Operation Exception occurs.

If Exception Is Not Enabled:

If any other exception does not occur, Q-NaN is stored to the *Destination* register.

8.2.4 Divide-by-Zero Exception (Z)

The Division-by-Zero exception occurs if the divisor is zero and the dividend is a finite nonzero number. This exception occurs due to other operations that produce a signed infinity, such as ln(0), sec(π/2) or Q^{-1}.

If Exception Is Enabled:

The contents of the *Destination* register are not changed, the contents of the *Source* register are preserved, and the zero division exception occurs.

If Exception Is Not Enabled:

If any other exception does not occur, the infinite number ($\pm\infty$) determined by the sign of the operand is stored to the *Destination* register.

8.2.5 Overflow Exception (O)

The Overflow exception occurs when the magnitude of the rounded floating-point result, with an unbounded exponent range, is larger than the largest finite number of the destination format. (An Inexact exception and *Flag* bit is set.)

If Exception Is Enabled:

The contents of the *Destination* register is not modified, and the *Source* registers are preserved, and the overflow exception occurs.

If Exception Is Not Enabled:

If any other exception does not occur, the default value determined by the rounding mode is stored to the *Destination* register (refer to **Table 8-1 Default FPU IEEE754 Exception Values**).

8.2.6 Underflow Exception (U)

Two related events generate the Underflow exception:

- If the operation result is -2^{Emin} to $+2^{Emin}$ (other than 0)
- extraordinary loss of accuracy during the arithmetic operation of such tiny numbers by denormalized numbers.

The IEEE754 provides several methods of underflow detection. Note, however, that the same detection method must be used for any processing.

The following two methods are used to detect an underflow.

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

The MIPS architecture detects an underflow after rounding.

To detect a drop in the accuracy, the following two methods are used.

- Denormalize loss (if a given result differs from the result calculated when the exponent range is infinite)
- Inexact result (if a given result differs from the result calculated when the exponent range and accuracy are infinite)

The MIPS architecture detects a drop in the accuracy as an inexact result.

If Exception Is Enabled:

If the underflow exception or inexact operation exception is enabled, or if the *FS* bit of the *FCR31* register is not set, the unimplemented operation exception (E) occurs. At this time, the contents of the destination register are not changed.

If Exception Is Not Enabled:

If the underflow exception and inexact operation exception are disabled, and if the *FS* bit of the *FCR31* register are set, the default value determined by the rounding mode is stored to the *Destination* register (refer to **Table 8-1 Default FPU IEEE754 Exception Values**).

8.2.7 Unimplemented Operation Exception (E)

If an attempt is made to execute an instruction of an operation code or format code reserved for future expansion, the *E* bit is set and an exception occurs. The operand and the contents of the *Destination* register are not changed. Usually, instructions are emulated by software. If the IEEE754 exceptions occur from an emulated operation, simulate those exceptions.

The unimplemented operation exception also occurs in the following cases. These are cases where an abnormal operand that cannot be handled correctly by hardware, or an abnormal result is detected.

- If the operand is a denormalized number (except compare instruction)
- If the operand is Q-NaN (except compare instruction)
- If the result is a denormalized number or underflows when the underflow/inexact operation exception is enabled and when the *FS* bit of the *FCR31* register is set
- If a reserved instruction is executed
- If a unimplemented format is used
- If a format whose operation is invalid is used (e.g., CVT.S.S)

Caution If the type conversion or arithmetic operation instruction is executed and if the operand is a denormalized number or NaN, the exception occurs. The exception does not occur even if the operand is a denormalized number of NaN when the transfer instruction is executed.

How to use the unimplemented operation exception is arbitrarily determined by the system. To maintain complete compatibility with the IEEE754, the unimplemented operation exception can be handled by software if occurs.

If Exception Is Enabled:

The contents of the *Destination* register are not changed, the contents of the source register are preserved, and the unimplemented operation exception occurs.

If Exception Is Not Enabled:

This exception cannot be disabled because there is no corresponding *Enable* bit.

★ **Restrictions:**

An unimplemented operation exception will occur in response to the execution of a type conversion instruction in the following cases.

- If an overflow occurs during conversion to integer format
- If the source operand is an infinite number
- If the source operand is NaN

The type conversion instructions affected by this restriction are as follows.

CEIL.L.fmt	fd, fs	FLOOR.L.fmt	fd, fs
CEIL.W.fmt	fd, fs	FLOOR.W.fmt	fd, fs
CVT.D.fmt	fd, fs	ROUND.L.fmt	fd, fs
CVT.L.fmt	fd, fs	ROUND.W.fmt	fd, fs
CVT.S.fmt	fd, fs	TRUNC.L.fmt	fd, fs
CVT.W.fmt	fd, fs	TRUNC.W.fmt	fd, fs

8.3 Saving and Returning State

Sixteen doubleword* LDC1 or SDC1 operations save or return the coprocessor floating-point register state in memory. The information in the *Control* and *Status* register can be saved or returned to the CPU register through CFC1 and CTC1 instructions. Normally, the *Control*/*Status* register is saved first and returned last.

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

Writing a zero value to the *Cause* field of *FCR31* register clears all pending exceptions, permitting normal processing to restart after the floating-point register state is returned.

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

8.4 Handling of IEEE754 Exceptions

The IEEE754 recommends the exception handler for any of the five standard exceptions; the exception handler can compute and restore a substitute result in the *Destination* register.

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

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

To obtain the correct rounded result if the overflow, underflow (except when the conversion instruction is executed), or inexact operation exception occurs, develop software that checks the *Source* register or that simulates the instructions while an exception handler is executed.

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

The IEEE754 recommends that, if enabled, the overflow and underflow exceptions take precedence over a separate inexact exception. This prioritization is accomplished in software; hardware sets the bits for both the Overflow or Underflow exception and the Inexact exception.

[MEMO]

Initialization Interface

9

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

9.1 Functional Overview

The V_R4300 processor has the following three types of resets; they use the **$\overline{\text{ColdReset}}$** and **$\overline{\text{Reset}}$** signals.

- **Power-ON Reset**: When the **$\overline{\text{ColdReset}}$** signal is asserted active after the power is applied and has become stable all clocks are restarted. A Power-ON Reset completely initializes the internal state of the processor without saving any state information.
- **Cold Reset**: When the **$\overline{\text{ColdReset}}$** signal is asserted active while the processor is operating all clocks are restarted. A Cold Reset completely initializes the internal state of the processor without saving any state information.
- **Soft Reset**: restarts processor, but does not affect clocks. The major part of the initial status of the processor can be retained by using soft reset.

After reset, the processor is bus master and drives the **SysAD(31:0)** bus.

Care must be taken to coordinate system reset with other system elements. In general, bus errors immediately before, during, or after a reset may result in undefined operations. Since the initialization of the internal state by a reset of the V_R4300 processor is performed only for some parts, make sure to completely initialize the processor through software.

The operation of each type of reset is described in sections that follow. Refer to **Figures 9-1** to **9-3** later in this chapter for timing diagrams of the Power-ON, Cold, and Soft Resets.

9.2 Reset Signal Description

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

ColdReset signal

The **ColdReset** signal must be asserted active to initialize the processor using Power-ON Reset or Cold Reset. At this time, the **RESET** signal can be asserted active or inactive. Set **DivMode (1:0)*** before the **Power-ON** Reset.

Do not deassert the **ColdReset** signal inactive at least for 64000 **MasterClock** Cycles after the signal has been asserted active. The **ColdReset** signal may be controlled not in synchronization with the **MasterClock**. When the **ColdReset** signal is deasserted inactive, the **SClock**, **TClock**, and **SyncOut** clock signals start operating in synchronization with the **MasterClock**.

* In V_R4300 and V_R4305. In V_R4310, DivMode(2:0).

Reset signal

Assert this pin active or inactive in synchronization with **MasterClock**, or keep it inactive at Power-ON Reset or Cold Reset.

Assert this pin active or inactive in synchronization with **MasterClock** at soft reset.

9.2.1 Power-ON Reset

Power-ON Reset is used to completely reset the processor. As a result:

- The *TS*, *SR*, and *RP* bits of the *Status* register and *EP* (3:0) bits of the *Config* register are cleared to 0.
- The *ERL* and *REV* bits of the *Status* register and *BE* bit of the *Config* register are set to 1.
- The upper-limit value (31) is assigned to the *Random* register.
- The *EC* (2:0) bits of the *Config* register are assigned to the contents of the **DivMode (1:0)*** pins.
- All the other internal statuses are undefined.

* In V_R4300 and V_R4305. In V_R4310, DivMode(2:0).

After the power supply to the processor has stabilized after Power-ON Reset, assert the **ColdReset** signal active for the duration of 64000 MasterClock cycles or more (0.96 ms during external 66.7-MHz operation).

Determine the **DivMode** signal until the **$\overline{\text{ColdReset}}$** signal is asserted active. The **DivMode** signal cannot be changed after that. If the **DivMode** signal is changed after the **$\overline{\text{ColdReset}}$** signal has been asserted active, the operation of the processor is not guaranteed.

When asserting the **$\overline{\text{ColdReset}}$** signal active, the **$\overline{\text{Reset}}$** signal may be active or inactive. However, do not change the value of the **$\overline{\text{Reset}}$** signal during the reset sequence.

Keep the **$\overline{\text{Reset}}$** signal active for the duration of 16 **MasterClock** cycles immediately after the **$\overline{\text{ColdReset}}$** signal has been deasserted inactive.

The output signals of the system interface are as follows during the reset period.

- **$\overline{\text{PValid}}$** signal : 1
- **$\overline{\text{PReq}}$** signal : 1
- **$\overline{\text{PMaster}}$** signal : 0
- **SysAD (31:0)** : Undefined
- **SysCmd (4:0)** : Undefined

When resetting has been completed, the processor serves as the bus master and drives **SysAD (31:0)**. The processor branches to a reset exception vector and starts executing a reset exception code.

9.2.2 Cold Reset

A Cold Reset is used to completely reset the processor.

- the *TS*, *SR*, and *RP* bits of the *Status* register and the *EP* (3:0) bits of the *Config* register are cleared to 0
- the *ER*L and *BEV* bits of the *Status* register and the *BE* bit of the *Config* register are set to 1
- the value of the upper bound (31) is set to the *Random* register
- all states other than above are undefined

When executing cold reset, keep the **$\overline{\text{ColdReset}}$** signal active for the duration of 64000 **MasterClock** cycles or more (0.96 ms during external 66.7-MHz operation).

When asserting the **$\overline{\text{ColdReset}}$** signal active, the **$\overline{\text{Reset}}$** signal may be active or inactive. However, do not change the value of the **$\overline{\text{Reset}}$** signal during reset sequence.

Keep the **$\overline{\text{Reset}}$** signal active for the duration of 16 **MasterClock** cycles immediately after the **$\overline{\text{ColdReset}}$** signal has been deasserted inactive.

The output signals of the system interface are as follows during the reset period.

- **$\overline{\text{PValid}}$** signal : 1
- **$\overline{\text{PReq}}$** signal : 1
- **$\overline{\text{PMaster}}$** signal : 0
- **SysAD (31:0)** : Undefined
- **SysCmd (4:0)** : Undefined

When resetting has been completed, the processor serves as the bus master and drives **SysAD (31:0)**. The processor branches to a reset exception vector and starts executing a reset exception code.

9.2.3 Soft Reset

A Soft Reset is used to reset the processor without affecting the output 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:

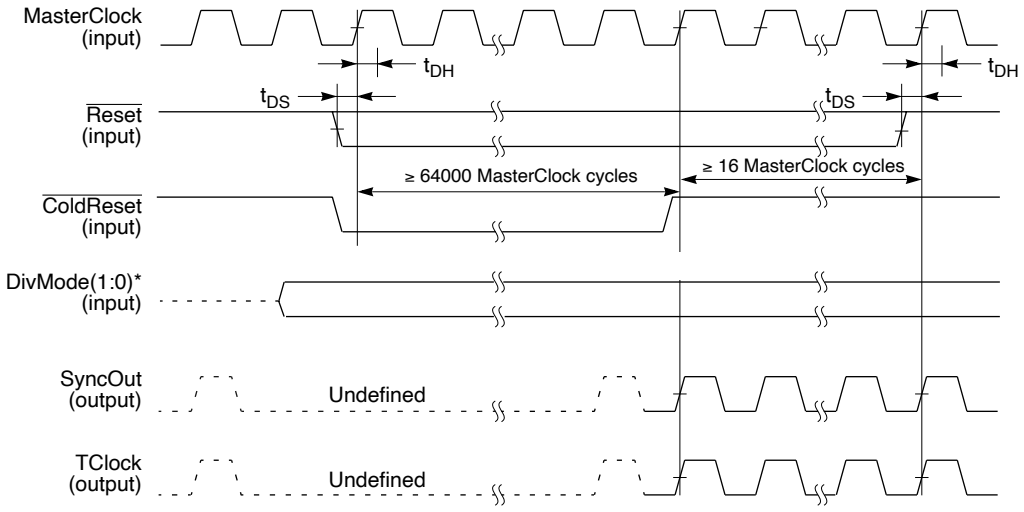
- the *Status* register *BEV*, *SR*, and *ERL* bits are set (to 1)
- the *Status* register *TS* and *RP* bit is cleared (to 0)

Because soft reset is executed as soon as the **$\overline{\text{Reset}}$** signal has asserted active, undefined data remains as a result if a multicycle instruction or floating-point instruction such as cache miss is executed.

Keep the **$\overline{\text{Reset}}$** signal asserted active at least for the duration of 16 **MasterClock** cycles. At this time, satisfy the setup and hold times with the **MasterClock**.

After the reset is completed, the processor becomes bus master and drives the **SysAD(31:0)** bus, the processor branches to the Reset exception vector and begins executing the reset exception code.

If **$\overline{\text{Reset}}$** signal is asserted in the middle of a **SysAD(31:0)** transaction, care must be taken to reset all external agents to avoid **SysAD(31:0)** bus contention.



* Determine the **DivMode** signal before the **$\overline{\text{ColdReset}}$** signal is asserted active.
In V_R4300 and V_R4305. In V_R4310, DivMode(2:0).

Figure 9-1 Power-ON Reset

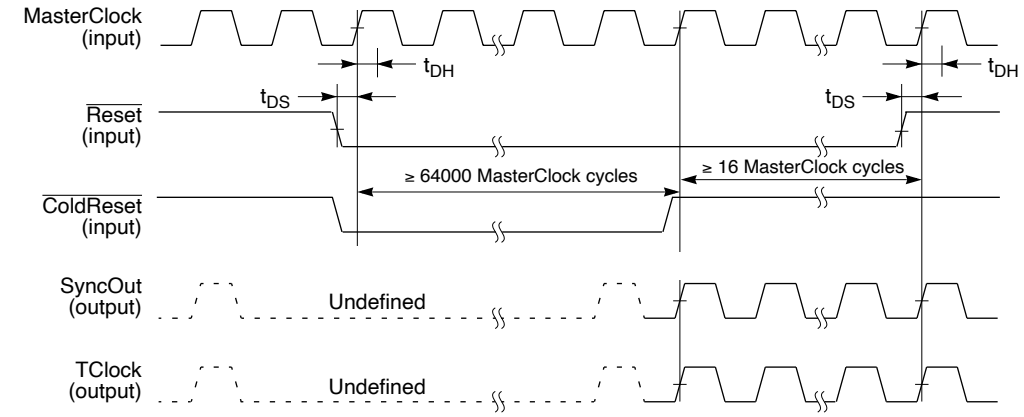


Figure 9-2 Cold Reset

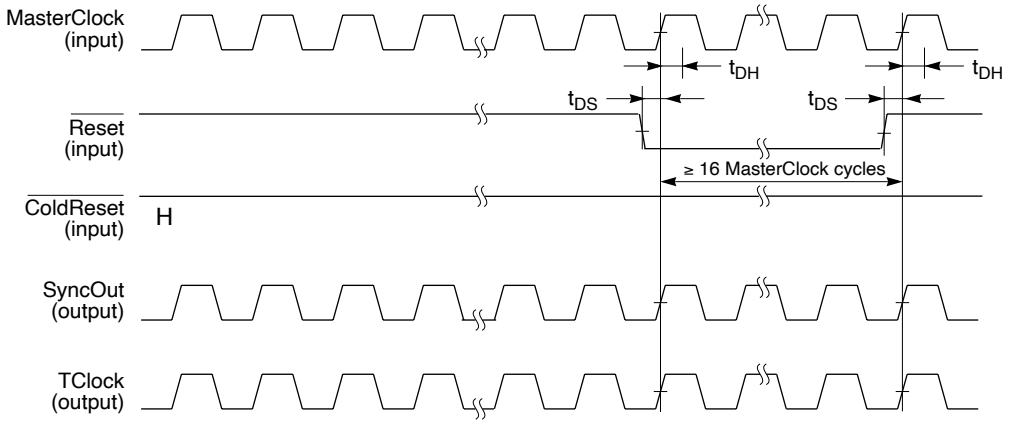


Figure 9-3 Soft Reset

9.3 V_R4300 Processor Modes

The V_R4300 processor supports several user-selectable modes. All modes except **DivMode** are set/reset by writing to the *Config* register.

9.3.1 Power Modes

★ The V_R4300 supports three power modes: normal power, low power (100 MHz model of the V_R4300 and the V_R4305 only), and power-off.

Normal Power Mode

Normally the processor clock (**PClock**) is generated from the input clock (**MasterClock**). The frequency ratio of the **PClock** to the **MasterClock** is set by the **DivMode(1:0)***. For the setting, refer to **Table 2-2 Clock/Control Interface Signals**. The frequency of the system interface clock (**SClock**) is the same as those of the **MasterClock**.

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

* In V_R4300 and V_R4305. In V_R4310, DivMode(2:0).

★ Low Power Mode (100 MHz model of V_R4300 and V_R4305 only)

The user may set the processor to low power mode by setting the *RP* bit of the *Status* register to 1. In *RP* mode, the processor stalls the pipeline and goes into a quiescent state—the store buffers empty and all cache misses resolved. However, the *RP* mode operation is guaranteed only when the **MasterClock** is 40 MHz or more. The frequency of **PClock** drops to the 1/4 of the normal level. The speeds of **SClock** and **TClock** also drop to the 1/4 of the normal level.

This feature reduces the power consumed by the processor chip to 25% of its normal value.

Software must guarantee the proper operation of the system upon setting or clearing the *RP* bit.

1. The functions of circuits such as the DRAM refresh counter change if the operating frequency changes. Therefore, write new values to the registers of the external agent that are directly affected by changes in frequency.
2. Set the system interface in the inactive status. For example, execute a read instruction to the non-cache area, and make the write buffer empty before completion of the instruction execution. Then the *RP* bit can be set or cleared.

3. Make sure that the eight instructions before and after the MTC0 instruction that sets or clears the *RP* bit do not generate exceptions such as cache miss and TLB miss.

Power Off Mode

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

9.3.2 Privilege Modes

The V_R4300 supports three modes of system privilege: Kernel, Supervisor, and User Extended addressing. This section describes these three modes.

Kernel Extended Addressing

When the *KX* bit is set to 1 by the *Status* register, the expansion TLB miss exception vector is used if the TLB miss exception of the Kernel address occurs. In the Kernel mode, the MIPSIII instruction set can be always used regardless of the *KX* bit.

Supervisor Extended Addressing

If the *SX* bit is set to 1 by the *Status* register, the MIPSIII instruction set can be used in the supervisor mode, and the expansion TLB miss exception vector is used if the TLB miss exception of the supervisor address occurs. If this bit is cleared, the MIPSI and II instruction sets and 32-bit virtual addresses are used.

User Extended Addressing

If the *UX* bit is set to 1 by the *Status* register, the MIPSIII instruction set can be used in the User mode, and the expansion TLB miss exception vector is used if the TLB miss exception of the user address occurs. If this bit is cleared, the MIPSI and II instruction sets and 32-bit virtual addresses are used.

9.3.3 Floating-Point Registers

If the *FR* bit of the *Status* register is set to 1, all the thirty-two 64-bit floating-point registers defined by the MIPSIII architecture can be accessed. If this bit is cleared, the processor accesses the sixteen 64-bit floating-point registers defined by the MIPSII architecture.

9.3.4 Reverse Endianness

If the *RE* bit of the *Status* register is set to 1, the endian in the User mode is reversed.

9.3.5 Instruction Trace Support

If the *ITS* bit of the *Status* register is set to 1, the physical address at the branch destination can be output from **SysAD(31:0)** when the instruction address is changed by execution of a jump or branch instruction or by occurrence of an exception. This function is disabled when the *ITS* bit is cleared.

Use this function to forcibly generate an instruction cache miss in the following cases.

- If the branch condition is satisfied when a branch instruction is executed
- If the contents of the PC are changed by execution of a jump instruction or by occurrence of an exception

When the instruction cache miss occurs, a processor block read request is issued from the **SysAD(31:0)**. This informs the change in the address to the outside. Return the response data to the processor block read request in the same manner as for a normal request.

The address to be output is not a PC value (virtual address) but a physical address.

9.3.6 Bootstrap Exception Vector (BEV)

This bit is used when diagnostic tests cause exceptions to occur prior to verifying proper operation of the cache and main memory system. The Bootstrap Exception Vector (*BEV*) bit is automatically set to 1 at cold reset or soft reset and on occurrence of the NMI exception. This bit can also be set by software.

When set, the Bootstrap Exception Vector (*BEV)* bit in the *Status* register causes the TLB miss 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).

9.3.7 Interrupt Enable (IE)

When the *IE* bit in the *Status* register is cleared, interrupts are not allowed, with the exception of reset and the non-maskable interrupt.

Clock Interface

10

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

10.1 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 that is taken for a signal to move from the input of a device (*clock*) to the output of the device (*Q*).

Figures 10-1 and 10-2 illustrate these terms.

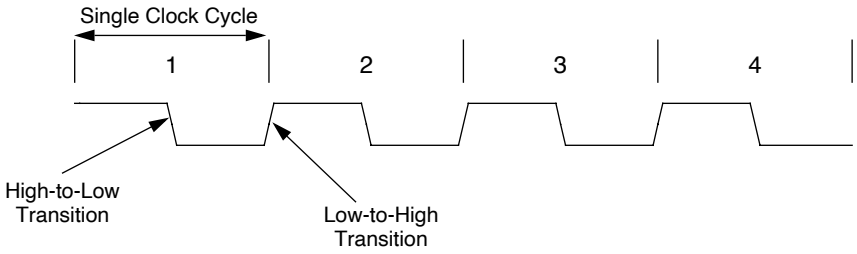


Figure 10-1 Signal Transitions

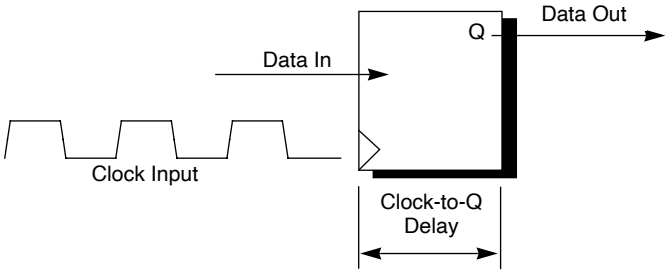


Figure 10-2 Clock-to-Q Delay

10.2 Basic System Clocks

The various clock signals used in the V_R4300 processor are described below.

MasterClock

The internal and external (system interface) clocks of the V_R4300 are generated and operate based on the **MasterClock**.

SyncIn/SyncOut

The V_R4300 processor generates **SyncOut** at the same frequency as **MasterClock** and aligns **SyncIn** with **MasterClock**.

SyncOut must be connected to **SyncIn** either directly, or through an external buffer. The processor can compensate for both output driver and input buffer delays when aligning **SyncIn** with **MasterClock**. When **SyncOut** is connected to **SyncIn** through an external buffer as illustrated in Figure 10-7, delay caused by external buffers connected to clock outputs can also be compensated.

PClock

The **PClock** is selected by setting the frequency ratio between the **PClock** and the **MasterClock**.

This ratio is set by the **DivMode** pins on power application. Table 10-1 indicates the selectable frequency ratio. For details of the **DivMode** pins settings, refer to **Table 2-2 Clock/Control Interface Signals**.

★ When the low power mode (100 MHz model of the V_R4300 and the V_R4305 only) is set by setting the *RP* bit of the *Status* register, the frequency of **PClock** decreases to the 1/4 of the normal level.

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

Table 10-1 Frequency Ratio Between PClock and MasterClock

Product Name	DivMode Pin	Selectable Frequency Ratio (MasterClock : PClock)
V_R4300	DivMode (1 : 0)	1 : 1.5[*1], 1 : 2, 1 : 3, 1 : 4[*2]
V_R4305	DivMode (1 : 0)	1 : 1, 1 : 2, 1 : 3
V_R4310	DivMode (2 : 0)	1 : 2, 1 : 2.5[*3], 1 : 3, 1 : 4, 1 : 5, 1 : 6

***1.** Selectable with the 100 MHz model only (With the 133 MHz model, this setting is reserved.)
2. Selectable with the 133 MHz model only (With the 100 MHz model, this setting is reserved.)
3. Selectable with the 167 MHz model only (With the 133 MHz model, this setting is reserved.)

SClock

The frequency of the system interface clock (**SClock**) is equal to that of **MasterClock**, and **SClock** is synchronized with **MasterClock**. Because **SClock** is generated from **PClock**, the frequency of **SClock** also drops to the 1/4 of the normal level, like the frequency of **PClock**, when the low power mode (100 MHz ★ model of the V_R4300 and the V_R4305 only) is set. The output of the V_R4300 is driven at the edge of **SClock**.

SClock rises in synchronization with the first rising edge of **MasterClock** immediately after $\overline{\textbf{ColdReset}}$ is deasserted inactive.

TClock

TClock (transfer/receive clock) is the reference clock of the output and input registers of the external agent. It is also used as the global clock of the external agent, and a clock can be supplied to all the logic circuits in the external agent.

TClock is the same as **SClock** in frequency, and its edge is accurately synchronized with that of **SClock**. When **SyncIn** is connected to **SyncOut**, **TClock** can also be synchronized with **MasterClock**.

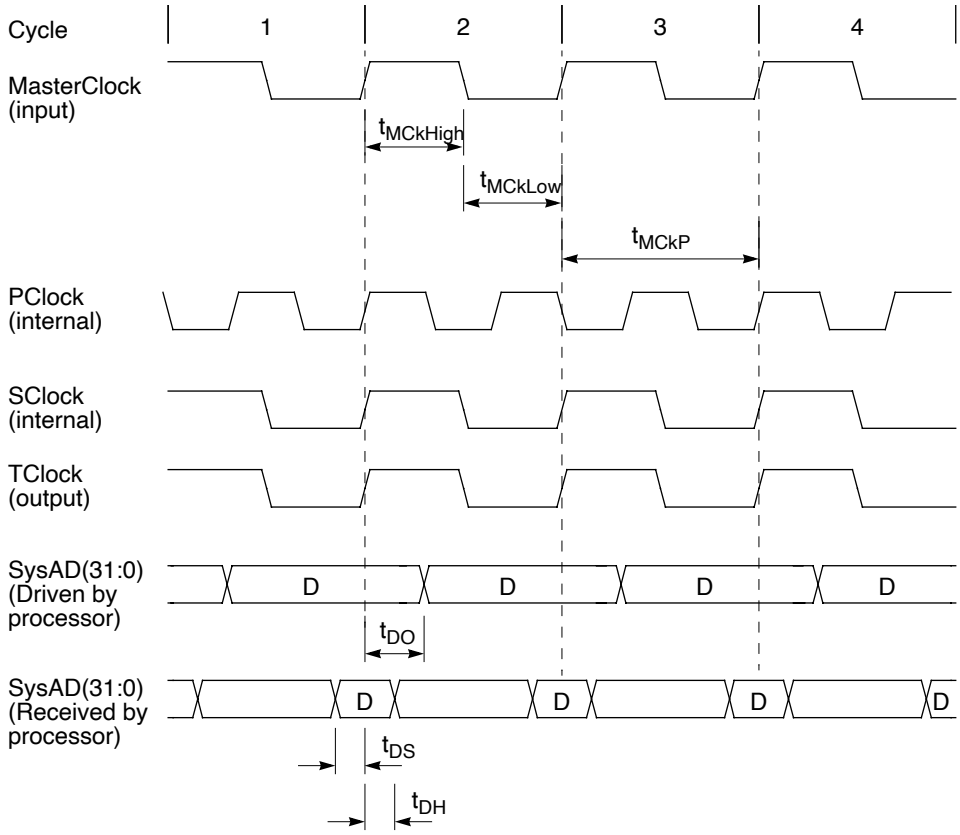


Figure 10-3 When Frequency Ratio of MasterClock to PClock is 1:1.5

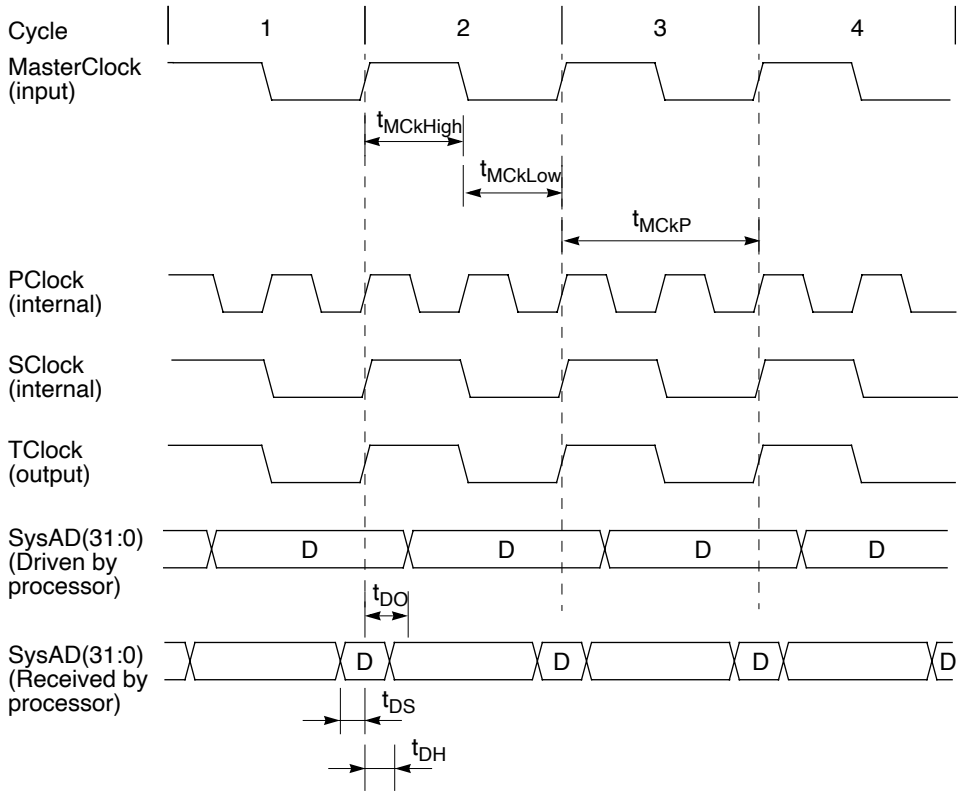


Figure 10-4 When Frequency Ratio of MasterClock to PClock is 1:2

10.3 System Timing Parameters

As shown in Figures 10-3 and 10-4, data provided to the processor must be stable a minimum of t_{DS} nanoseconds (ns) before the rising edge of **SClock** and be held valid for a minimum of t_{DH} ns after the rising edge of **SClock**.

10.3.1 Synchronization with SClock

Processor data becomes stable 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.

10.3.2 Synchronization with MasterClock

Certain processor inputs (specifically $\overline{\textbf{Reset}}$) are sampled based on **MasterClock**. The same setup, hold, and off time, t_{DS}, t_{DH}, and t_{DO}, shown in Figures 10-3 and 10-4, apply to these inputs, measured by **MasterClock**.

10.3.3 Phase-Locked Loop (PLL)

The processor synchronizes **SyncOut**, **PClock**, **SClock**, and **TClock** with internal phase-locked loop (PLL) circuits that generate aligned clocks based on **SyncOut**/**SyncIn**. By their nature, PLL circuits are only capable of generating synchronized clocks with the **MasterClock** frequencies within a limited range.

Clocks generated using PLL circuits contain some inherent inaccuracy, or *jitter*; a clock synchronized with **MasterClock** by the PLL can lead or trail **MasterClock** by as much as the related maximum jitter ($t_{MCJitter}$).

10.4 Low Power Mode Operation

Usually, **PClock** is generated based on **MasterClock** at the frequency ratio set by the **DivMode(1:0)**[*1] pins (for the setting, refer to **Table 2-2 Clock/Control Interface Signals**). The frequency of the system interface clock (**SClock**) is the same as that of **MasterClock**.

★ To set the low power mode (RP)[*2], set the *RP* bit of the *Status* register by using a transfer instruction. When the RP mode has been set, the processor stalls the pipeline which then enters the pause (quiescent) status (in other words, the store buffer becomes empty and all cache misses are solved). Next, the frequency of **PClock** drops to the 1/4 in the normal mode. The frequency of **SClock** also drops to the 1/4 of the normal level (10 MHz).

The normal clocks can be restored by executing reset.

For the procedure to set or clear the *RP* bit, refer to **Low Power Mode** in **9.3.1**.

***1.** In V_R4300 and V_R4305. In V_R4310, DivMode(2:0).
2. 100 MHz model of the V_R4300 and the V_R4305 only

10.5 Connecting Clocks to a Phase-Locked System

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

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

Figure 10-5 shows a block diagram of a phase-locked system using the V_R4300 processor.

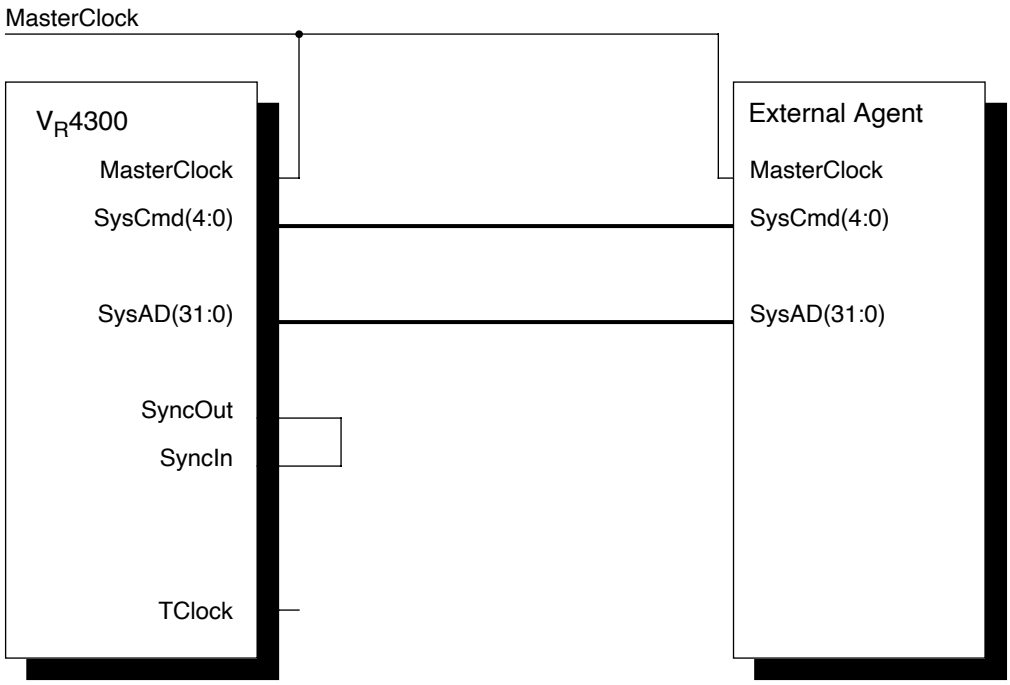


Figure 10-5 Phase-Locked System

10.6 Connecting Clocks to a System without Phase Locking

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

10.6.1 Connecting to a Gate-Array Device

When the processor is connected to a gate array device, **TClock** is used as the transmit/receive clock in the gate array.

Figure 10-6 is a block diagram of a system without phase lock, using the V_R4300 processor with an external agent implemented as a gate array.

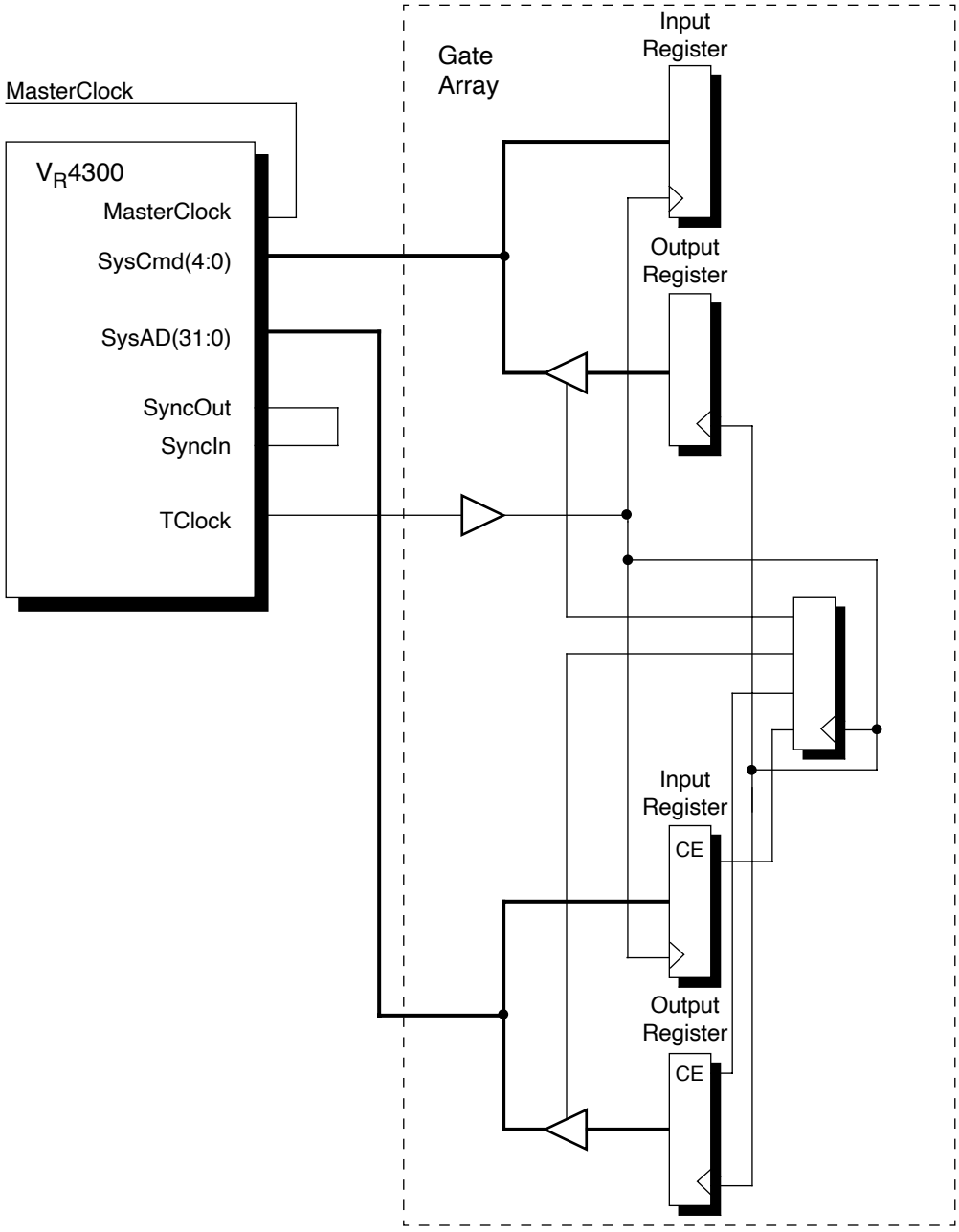


Figure 10-6 Gate-Array System without Phase Lock, Using the V_R4300 Processor

Signal Transmission Time from Processor to External Agent

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 = (1TClock period) – (t_{DO} for V_R4300)
+ (Minimum External Clock Buffer Delay)
– (External Input Register Setup Time)
– (Maximum Clock Jitter for V_R4300 Internal Clocks)
– (Maximum Clock Jitter for TClock)

Signal Transmission Time from External Agent Processor

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

Transmission Time = (1TClock period) – (t_{DS} for V_R4300)
– (Maximum External Clock Buffer Delay)
– (Maximum External Output Register Clock-to-Q Delay)
– (Maximum Clock Jitter for TClock)
– (Maximum Clock Jitter for V_R4300 Internal Clocks)

10.6.2 Connecting to a CMOS Discrete Device

The processor uses a clock buffer that corrects the delay to supply a synchronous clock to an external CMOS discrete device. The clock buffer that corrects the delay is inserted into the **SyncOut**/**SyncIn** synchronization bus of the processor to adjust the skew of **SyncOut** and **TClock** by delaying **PClock** synchronized with **MasterClock**, and advances **SyncOut** and **TClock** from **MasterClock** by the buffer delay.

When using **TClock** whose buffer delay has been corrected, the other delay correcting clock buffers can be used.

The phase error of the buffered **TClock** can be obtained by adding up the maximum delay error of the delay correcting clock buffer and the maximum clock jitter of **TClock**.

Functioning as the global clock of the CMOS discrete devices that form the external agent, the buffered **TClock** supplies a clock to the register that samples the processor output and the register that drives the processor input.

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

Transmission Time = (1TClock period) – (t_{DO} for V_R4300)
– (External Input Register Setup Time)
– (Maximum External Clock Buffer Delay Mismatch)
– (Maximum Clock Jitter for V_R4300 Internal Clocks)
– (Maximum Clock Jitter for TClock)

Figure 10-7 is a block diagram of a system without phase lock, employing the V_R4300 processor and an external agent composed of both a gate array and CMOS discrete devices.

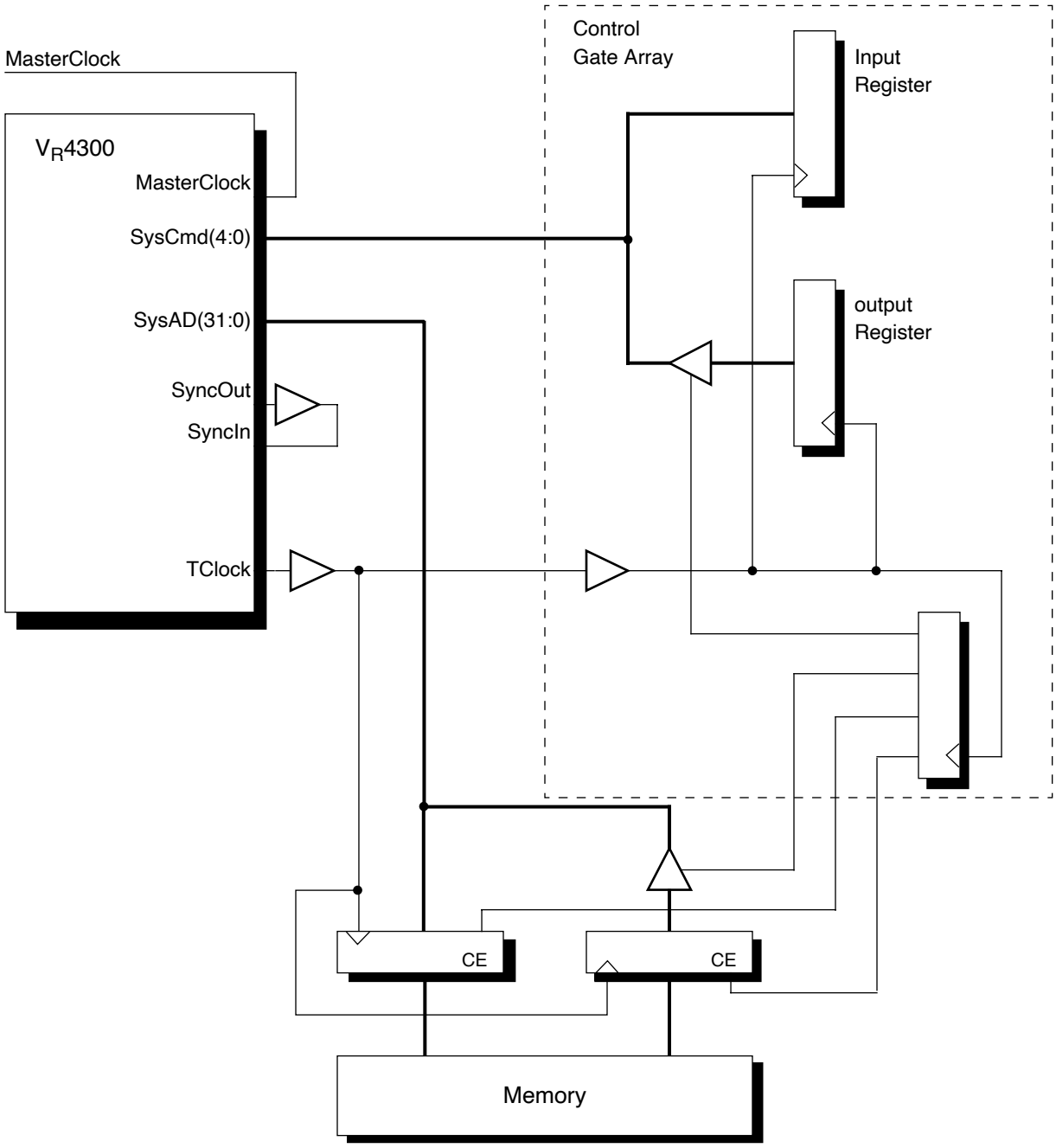


Figure 10-7 Gate-Array and CMOS System without Phase Lock, Using the V_R4300 Processor

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

Transmission Time = (1TClock period) – (t_{DS} for V_R4300)
– (Maximum External Output Register Clock-to-Q Delay)
– (Maximum External Clock Buffer Delay Mismatch)
– (Maximum Clock Jitter for V_R4300 Internal Clocks)
– (Maximum Clock Jitter for TClock)

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

Minimum Hold Time for the External Input Register
+ Maximum Clock Jitter for V_R4300 Internal Clocks
+ Maximum Clock Jitter for TClock
+ Maximum Delay Mismatch of the External Clock Buffers

[MEMO]

Cache Memory

11

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

This chapter uses the following terminology:

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

These terms are used interchangeably throughout this book.

11.1 Memory Organization

Figure 11-1 shows the V_R4300 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 11-1 has the capacity to hold more data than the block above it. For instance, physical main memory has a larger capacity than the 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.

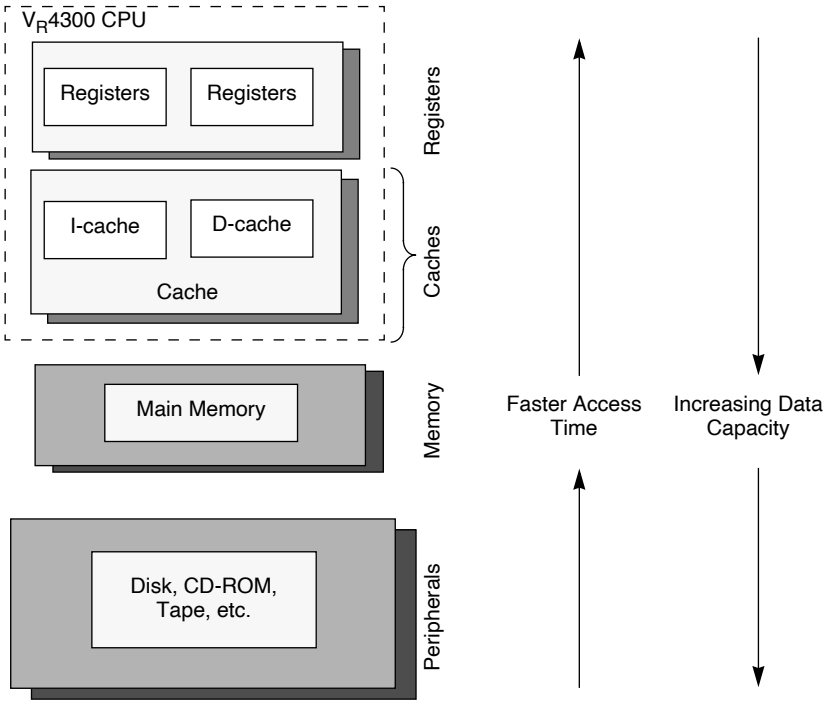


Figure 11-1 Logical Hierarchy of Memory

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

Data writes take two **PClock** cycles. In the first cycle, the store address is generated and the tag is checked; in the second cycle, the data is written into the data RAM.

11.2 Cache Organization

This section describes the organization of the on-chip data and instruction caches. Figure 11-2 provides a block diagram of the V_R4300 cache and memory model.

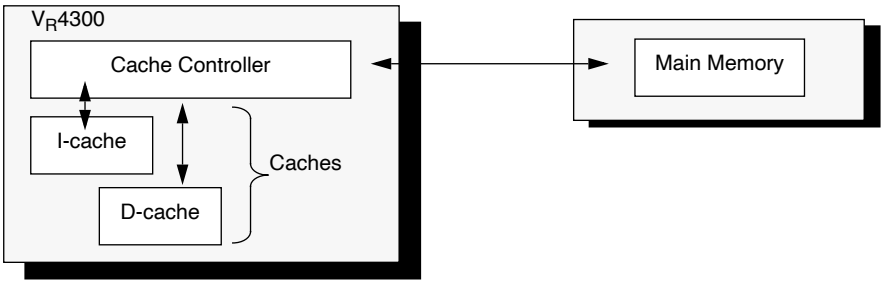


Figure 11-2 V_R4300 Cache Support

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

For cache tags, refer to **11.2.1 Organization of the Instruction Cache (I-Cache)** and **11.2.2 Organization of the Data Cache (D-Cache)**.

Cache Sizes

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

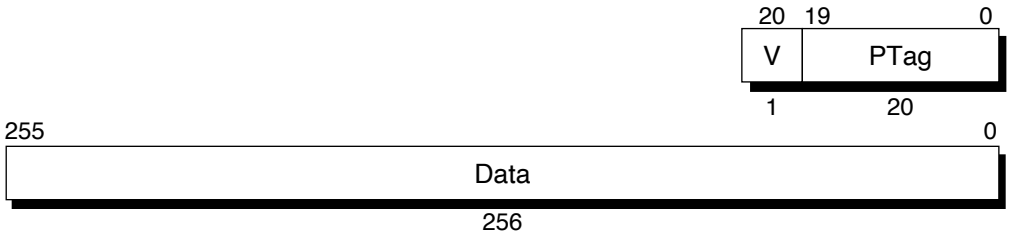
11.2.1 Organization of the Instruction Cache (I-Cache)

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

The V_R4300 processor I-cache has the following characteristics:

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

Figure 11-3 shows the format of an 8-word (32-byte) I-cache line.



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

Figure 11-3 V_R4300 8-Word I-Cache Line Format

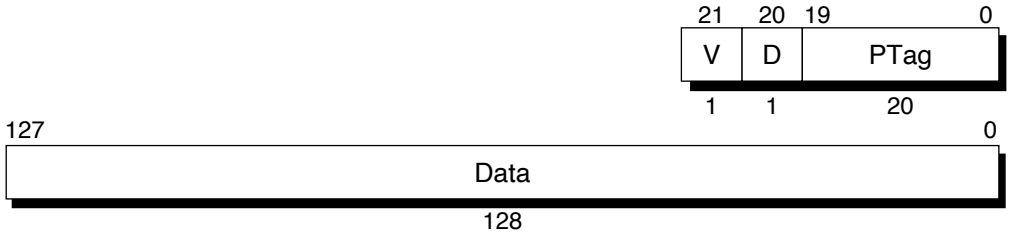
11.2.2 Organization of the Data Cache (D-Cache)

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

The V_R4300 processor D-cache has the following characteristics:

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

Figure 11-4 shows the format of a 4-word (16-byte) D-cache line.



V : Valid bit
D : Dirty bit (refer to **11.4 Cache States**)
PTag : Physical tag (bits 31:12 of the physical address)
Data : D-cache data

Figure 11-4 V_R4300 4-Word Data Cache Line Format

11.2.3 Accessing the Caches

Figure 11-5 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.

Data Cache Addressing

VA(12:4) is used. Since the cache size is 8 KB, the most significant bit is VA12. Furthermore, since the line size is 4 words (16 bytes), the least-significant bit is VA4.

Instruction Cache Addressing

VA(13:5) is used. Since the cache size is 16 KB, the most-significant bit is VA13. Furthermore, since the line size is 8 words (32 bytes), the least-significant bit is VA5.

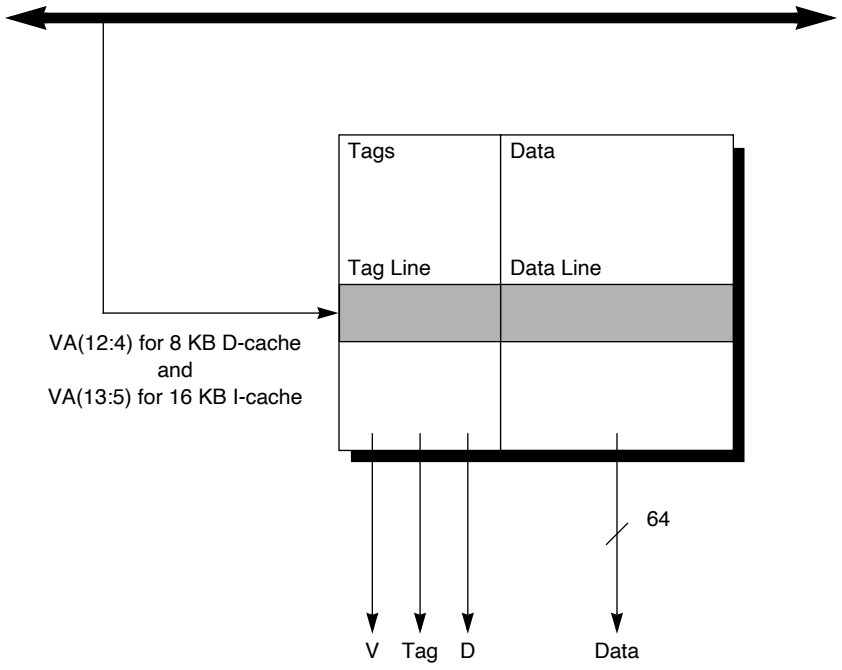


Figure 11-5 Cache Data and Tag Organization

11.3 Cache Operations

As described earlier, caches provide 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 requested 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 main 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 main memory until the cache line is to be replaced.

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

11.3.1 Cache Write Policy

The V_R4300 processor manages its data cache by using a write-back policy; that is, it stores write data into the cache, instead of writing it directly to the main memory.* Some time later this data is independently transferred into the main memory. In the V_R4300 implementation, a modified cache line is not written back to the main 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 cache-miss occurs and the processor writes the contents of a cache line back to the main memory, it does not ordinarily retain a copy of the cache line, and the state of the cache line is changed to Clean.

11.3.2 Data Cache Line Replacement

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

The line replacement sequence is based on a "Critical Doubleword First" scheme refer to **subblock ordering** in **12.2.1 Physical Addresses**. The processor restarts its pipeline as soon as the main memory supplies the desired word in the first doubleword of a block transfer. This sequence is summarized as follows:

1. Move the data physical address to the **SysAD(31:0)**. At the same time, move the dirty cache line to the write buffer.
2. At the timing of SClock rising edge, read the data from the main memory, receiving the desired doubleword in two word data first.
3. Receive remaining doubleword in word data units. For all loads move the data to target register. For byte, halfword and word stores, it is necessary to do a read in the main memory followed by a write procedure—read the 64-bit data, write new data to this read data, then write the 64-bit data to cache. As this is being done, interlock the data cache to prevent it from being accessed by any subsequent instruction that tries to access this particular cache line.

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

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

Data Load Miss

If the missed cache line is not dirty, it is replaced with a new line.

If the missed line is dirty, it is moved to the write buffer. A new line replaces the missed line, and the data in the write buffer is written to the main memory.

Data Store Miss

If the missed cache line is not dirty, it is replaced with the new cache line merged with the store data.

If the missed cache line is dirty, it is moved to the write buffer. A new cache line is merged with the store data and written to cache, and data in the write buffer is written to the memory. The data is written sequentially, starting from the first address of the block (refer to **sequential ordering** in **12.2.1 Physical Addresses**).

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

Table 11-1 Stall Cycle Count for Data Cache Miss

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

11.3.3 Instruction Cache Line Replacement

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

During an instruction cache miss, a memory read request is issued by the processor. That is the requested cache line is read from the main memory and written to the instruction cache. At this time the pipeline resumes execution, and the instruction cache is reaccessed.

The replacement sequence for an instruction cache miss is:

1. Move the instruction physical address to the **SysAD(31:0)**.
2. Read the instruction data at the timing of **SClock** rising edge from the main memory and write it out to the instruction cache.
3. Restart the pipeline operation.

The instruction cache miss stall in number of **PClock** cycles is:

Table 11-2 Stall Cycle Count for Instruction Cache Miss

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

11.4 Cache States

Cache Line

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

- **Valid**: a cache line that contains valid information.
- **Dirty**: a cache line containing data that has changed in valid status since it was loaded from memory.
- **Clean**: a cache line containing data that has not changed in valid status since it was loaded from the main 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.
  Neither a cold reset nor a soft reset makes the state of a cache invalid. Software invalidates it.

Data Cache

The data cache supports three cache states:

- invalid
- clean
- dirty

Instruction Cache

The instruction cache supports two cache states:

- invalid
- valid

The cache line that contains valid information may be changed when the processor executes the CACHE operation. For CACHE operation, refer to **Chapter 16 CPU Instruction Set Details**.

11.5 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.

11.5.1 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 the 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 the processor to cache, which induces no cache state transition

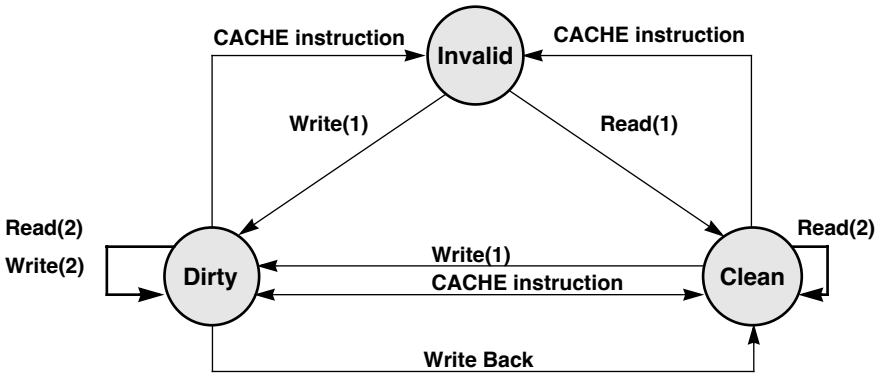


Figure 11-6 Data Cache State Diagram

11.5.2 Instruction Cache State Transition

The following diagram illustrates the instruction cache state transition sequence.

- Read(1) indicates a read operation from the main 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.

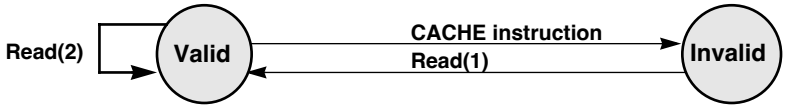


Figure 11-7 Instruction Cache State Diagram

11.6 Manipulation of the Caches by an External Agent

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

[MEMO]

System Interface

12

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

This chapter describes the System interface between the processor and the external agent.

The V_R4300 uses a subset of the System interface contained on the V_R4400 and V_R4200.

12.1 Terminology

The following terms are used in this chapter:

- An *external agent* is any device connected to the processor, over the System interface, that processes requests issued by the processor.
- A *system event* is an event that occurs within the processor and requires access to external resources. System events include: an instruction fetch that misses in the instruction cache; a load/store instruction that misses in the data cache; an uncached load or store instructions; an execution of cache instructions.
- *Sequence* refers to the series of requests that a processor generates to process a system event.
- *Protocol* refers to the cycle-by-cycle signal transitions that occur on the System interface pins, which issue external request, or a processor.
- *Syntax* refers to the definition of bit patterns on encoded buses, such as the command bus.
- *Block* indicates any data transfer of 8 bytes or longer across the System interface.
- *Single* indicates any data transfer of 7 bytes or shorter across the System interface.
- *Fetch* refers to the read of information from the instruction cache.
- *Load* refers to the read of information from the data cache.

12.2 System Interface Description

The processor uses the System interface to access external resources required for performing cache misses and uncached area processing.

12.2.1 Physical Addresses

Physical addresses are output to **SysAD(31:0)** in the address cycle. The address when the single read request and single write request are issued is determined by the data length as follows.

- If the data is a word (4 bytes), the low-order 2 bits of the address are 0.
- If the data is a halfword (2 bytes), the low-order 1 bit of the address is 0.
- If the data is 1, 3, 5, 6, or 7 bytes, the supplied address is a byte address (the 5-, 6-, or 7-byte data is divided into two single write requests).

When a doubleword (2 words), 4 words, or 8 words are transferred, a block request is issued. The block read request and block write request differ as follows in the physical address to be output.

Block Write Request

The physical address when the block write request is issued is always aligned with the first word address of the block (sequential ordering).

Block Read Request

- Instruction cache read request

  The block read request when a miss occurs in the instruction cache, the physical address is aligned with the 8-word data address (the low-order 5 bits are 0) including the requested word and output. Figure 12-1 shows the sequence in which data are transferred from the main memory when a block read request is issued to the instruction cache. When an instruction cache read request is issued, data is always read starting from W0 (sequential ordering).

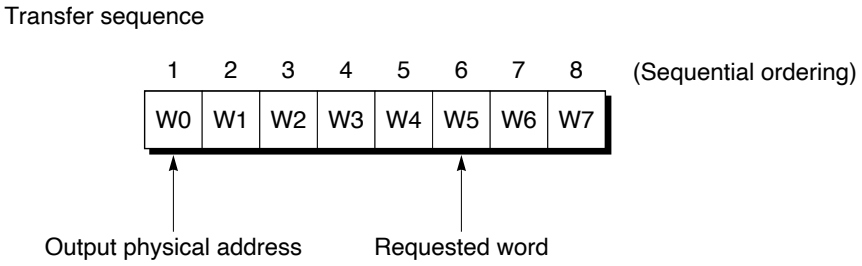


Figure 12-1 Data Sequence on Instruction Cache Read Request

- Data cache read request

  If a block read request is issued when a miss occurs in the data cache, the physical address is aligned with the doubleword address (the low-order 3 bits are 0) including the requested data and output. Figure 12-2 shows the data sequence in which data is transferred from the main memory when a block read request is issued to the data cache. When a data cache read request is issued, reading a doubleword including the necessary data is started in word units (W2 in this case) (refer to **Sub block ordering** in **12.12.2 Sequential and Subblock Ordering**).

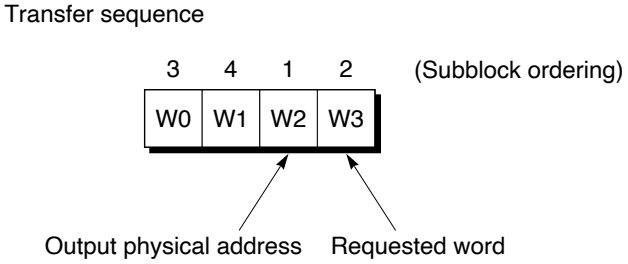


Figure 12-2 Data Sequence on Data Cache Read Request

12.2.2 Interface Buses

Figure 12-3 shows the primary communication buses for the System interface: a 32-bit address/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 device to issue an external request (refer to **12.4 Processor and External Requests**).

A request through the System interface consists of:

- an address
- a System interface command that specifies the nature of the request
- response data to read request, and write data to write request

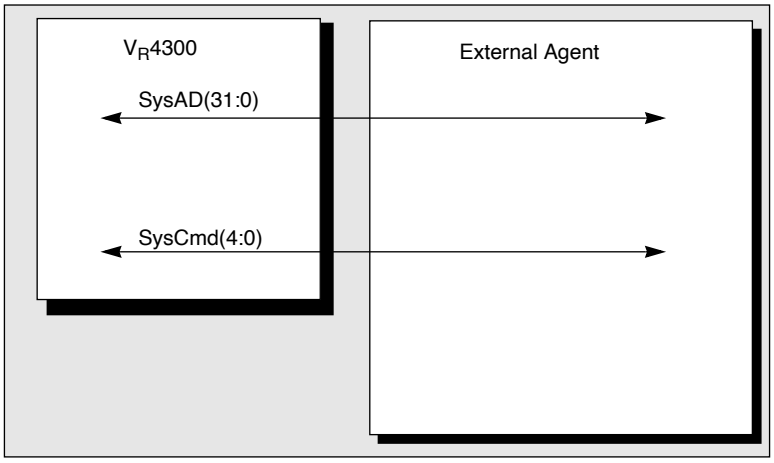


Figure 12-3 System Interface Buses

12.2.3 Address and Data Cycles

The **SysCmd (4:0)** bus identifies the contents of the **SysAD(31:0)** bus during any cycle in which it is valid. Cycles in which the **SysAD(31:0)** bus contains a valid address are called *address cycles*. Cycles in which the **SysAD(31:0)** bus contains valid data are called *data cycles*. The most significant bit of the **SysCmd(4:0)** 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 **$\overline{\text{EValid}}$** and **$\overline{\text{PValid}}$** signals (described in **12.2.2 Interface Buses**).

When the V_R4300 processor is driving the **SysAD(31:0)** and **SysCmd(4:0)** buses, the System interface is in *master state*. When the external agent is driving them, the System interface is in *slave state*.

- When the processor is master, it asserts the **$\overline{\text{PValid}}$** signal when the **SysAD(31:0)** and **SysCmd(4:0)** buses are valid.
- When the processor is slave, an external agent asserts the **$\overline{\text{EValid}}$** signal when the **SysAD(31:0)** and **SysCmd(4:0)** buses are valid.

SysCmd(4:0) indicate the following contents if the **$\overline{\text{PValid}}$** or **$\overline{\text{EValid}}$** signal is active.

- During address cycles [**SysCmd4** = 0], the remainder of the **SysCmd(4:0)** bus, **SysCmd(3:0)**, contains a *System interface command* (the encoding of System interface commands is detailed in **12.11 System Interface Commands and Data Identifiers**).
- During data cycles [**SysCmd4** = 1], the remainder of the **SysCmd(4:0)** bus, **SysCmd(3:0)**, contains a *data identifier command* (the encoding of data identifiers is detailed in **12.11 System Interface Commands and Data Identifiers**).

12.2.4 Issue Cycles

Processor Request

There are two types of processor issue cycles:

- processor read request
- processor write request

The issuance cycle of the processor read/write request is determined by the status of the **$\overline{\text{EOK}}$** signal. The issuance cycle is a cycle that becomes valid in the address cycle of each processor request. Only one issuance cycle exists for one processor request.

To define the issuance cycle of the address cycle, assert the **$\overline{\text{EOK}}$** signal active at the external agent side one cycle before the address cycle of the processor read/write request as shown in Figure 12-4.

To define the address cycle as the issuance cycle, do not deassert the **$\overline{\text{EOK}}$** signal inactive until the address cycle is started.

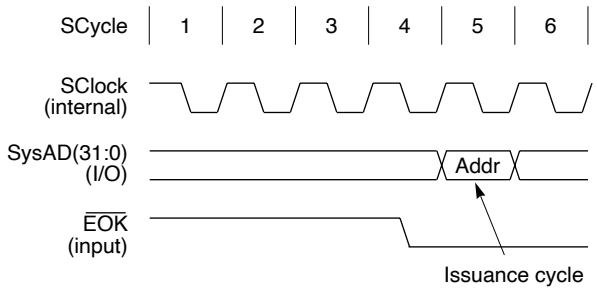


Figure 12-4 $\overline{EOK}$ Signal Status of Processor Request

The processor repeatedly outputs the address cycle until the address cycle of the processor request becomes the issuance cycle. With the V_R4300, therefore, the address cycle next to the cycle in which the **$\overline{\text{EOK}}$** signal has become active is the issuance cycle, and the address cycle is repeated up to that cycle. Figure 12-5 illustrates how the address cycle is extended by the **$\overline{\text{EOK}}$** signal.

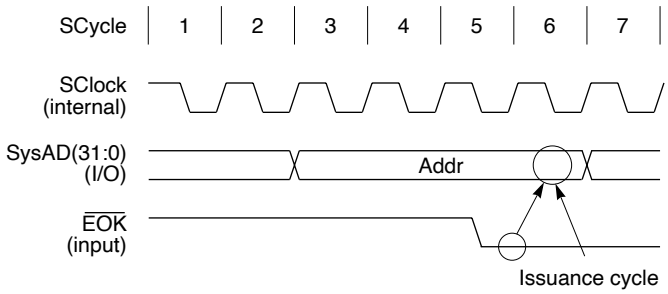


Figure 12-5 Address Cycle Extended by $\overline{EOK}$ Signal

Processor and External Requests

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

When an issuance of processor request and external request compete with each other, the processor either:

- completes the issuance of the processor request 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 completed.

12.2.5 Handshake Signals

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

$\overline{\text{EOK}}$ Signal

This signal is used by the external agent to indicate whether it can accept a new read or write transactions.

$\overline{\text{EReq}}$, $\overline{\text{PMaster}}$ and $\overline{\text{PReq}}$ Signals

These signals are used to transfer control of the **SysAD(31:0)** and **SysCmd(4:0)** buses. $\overline{\textbf{EReq}}$ signal is used by an external agent to indicate a need to control the interface. $\overline{\textbf{PMaster}}$ signal is deasserted by the processor when it transfers control of the System interface to the external agent. The $\overline{\textbf{PReq}}$ signal is used by the processor to request the external agent, which holds the right to control the system interface, for the right of control.

$\overline{\text{PValid}}$ and $\overline{\text{EValid}}$ Signals

The V_R4300 processor uses $\overline{\textbf{PValid}}$ signal, and the external agent uses $\overline{\textbf{EValid}}$ signal to indicate valid command/data on the **SysCmd(4:0)**/**SysAD(31:0)** buses.

12.3 System Interface Protocols

Figure 12-6 shows the register-to-register operation of the System interface. That is, output signals of the processor come directly from output registers and begin to change in synchronization with the rising edge of **SClock.**

Input signals to the processor are fed directly to input registers that latch these input signals with the rising edge of **SClock**.

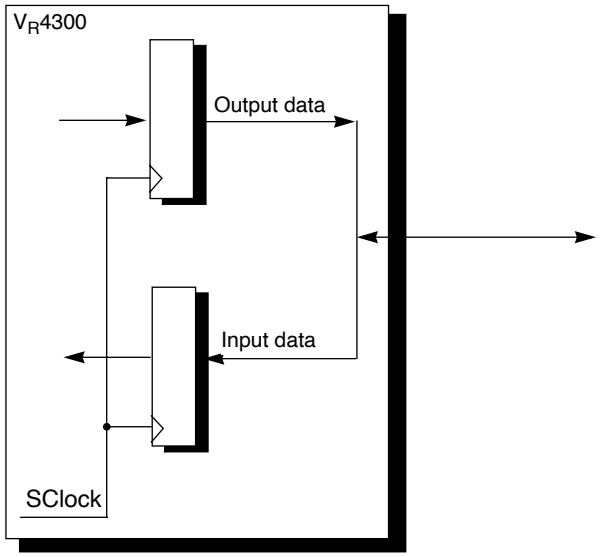


Figure 12-6 System Interface Register-to-Register Operation

12.3.1 Master and Slave States

When the V_R4300 processor is driving the **SysAD(31:0)** and **SysCmd(4:0)** buses, the System interface is in *master state*. When the external agent is driving these buses, the System interface is in *slave state*.

In master state, the processor asserts the **$\overline{\text{PValid}}$** signal whenever the **SysAD(31:0)** and **SysCmd(4:0)** buses are valid.

In slave state, the external agent asserts the **$\overline{\text{EValid}}$** signal whenever the **SysAD(31:0)** and **SysCmd(4:0)** buses are valid.

12.3.2 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 external arbitration, or 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 control (external arbitration).
- The processor issues a read request (uncompelled change to slave state).

The following sections describe these two cases.

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

1. An external agent transmits a request to issue an external request to the processor by asserting **$\overline{\text{EReq}}$** signal.
2. When the processor is ready to accept an external request, it releases the System interface from master to slave state by deasserting **$\overline{\text{PMaster}}$** signal.
3. The System interface returns to master state as soon as the issue of the external request is completed.

This process is described in **12.6.6 External Arbitration Protocol**.

12.3.4 Uncompelled Change to Slave State

An *uncompelled* change to slave state is the transition of the System interface from master state to slave state, performed by the processor itself when a processor read request is pending. **$\overline{\text{PMaster}}$** signal is deasserted automatically after a read request. An uncompelled change to slave state occurs either the first cycle after the issue cycle of a processor read request.

When the processor returns from the uncompelled transition differs depending on the cache status. The processor returns to the master status when the following external request (read response or other external request) is completed after the uncompelled transition to the slave status.

An external agent must confirm that the processor has performed an uncompelled change to slave state, and begin driving the **SysAD(31:0)** bus along with the **SysCmd(4:0)** 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 **$\overline{\text{EReq}}$** signal.

If **$\overline{\text{EReq}}$** is inactive, at the time the external request is completed, the System interface automatically returns to master state.

12.4 Processor and External Requests

There are two categories of requests: *processor requests* and *external requests*.

When a system event occurs, the processor issues a request through the system interface to access some external resource necessary 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 an internal resource of the processor issues an *external request*.

Processor requests include the following:

- read requests, which provide a read address to an external agent
- write requests, which provide an address and a single 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 response data as a read response, which may be returned at any time after the read request is completed.

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

The processor will not issue another request while a read request is pending (before receiving the response data after issuing the read request).

System events and requests are shown in Figure 12-7.

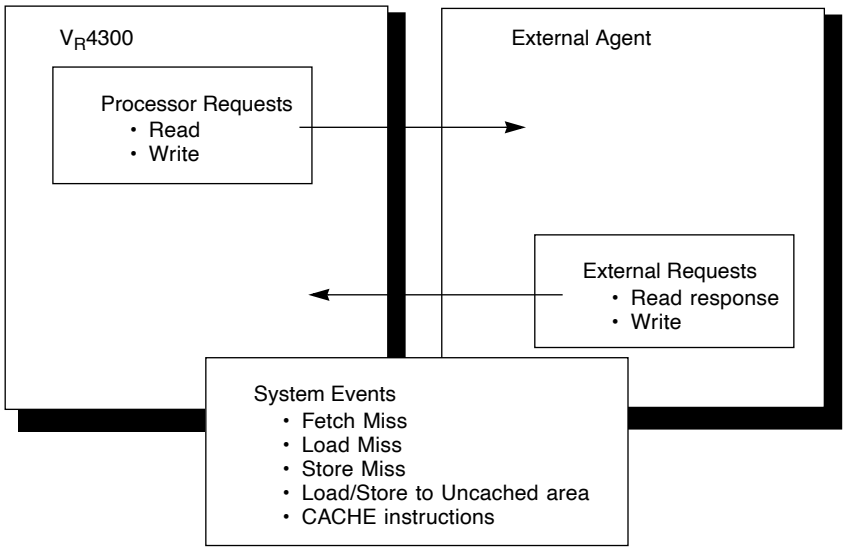


Figure 12-7 Requests and System Events

12.4.1 Processor Requests

A processor request is a request through the System interface, to access some external resource. Processor requests are either read or write requests.

Outline 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.

Request Issuance

The processor issues requests in a strict sequential order; 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 issues a write request only if there are no read requests pending.

Request Control

The processor has the input signal **$\overline{\text{EOK}}$** to allow an external agent to control the flow of processor requests.

The processor request cycle sequence is shown in Figure 12-8.

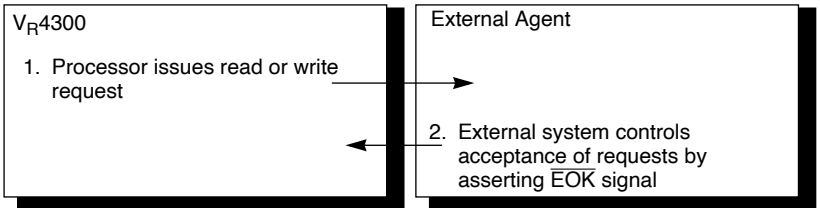


Figure 12-8 Processor Request Flow

12.4.2 Processor Read Request

When a processor issues a read request, the external agent must access the specified resource and return the requested data.

A processor read request can be split by the external agent's response 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.

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

Note that the data identifier associated with the response data can indicate that the response data is erroneous, causing the processor to generate a bus error exception.

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

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

12.4.3 Processor Write Request

When a processor issues a write request, the specified external resource is accessed and the data is written to it.

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

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

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

12.4.4 External Requests

External requests include read response and write requests.

Outline of 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.

Request Control

The processor controls the flow of external requests through the arbitration signals **$\overline{\text{EReq}}$** and **$\overline{\text{PMaster}}$**, as shown in Figure 12-9. The external agent must acquire mastership of the System interface before it issues an external request; the external agent acquires mastership of the System interface by asserting **$\overline{\text{EReq}}$** signal and then waiting for the processor to deassert **$\overline{\text{PMaster}}$** signal for one cycle.

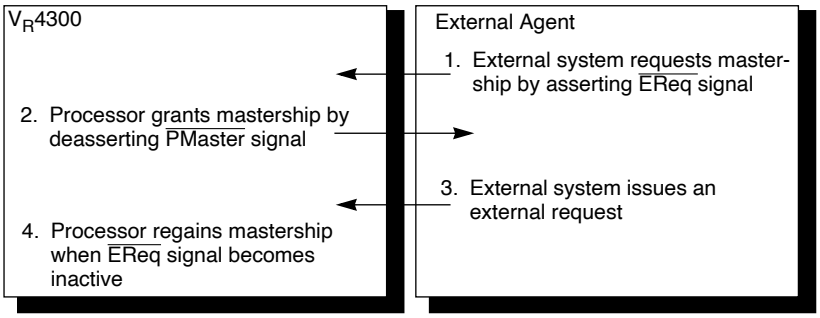


Figure 12-9 External Request Flow

Mastership of the System interface always returns to the processor when **$\overline{\text{EReq}}$** signal becomes inactive after an external request is issued. The processor does not accept a subsequent external request until it has completed the current request.

Request Issuance

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 **$\overline{\text{EReq}}$** signal indicating that it wishes to begin an external request. The processor releases mastership of the System interface by deasserting **$\overline{\text{PMaster}}$** signal. An external request can be accepted based on the criteria listed below.

- The processor completes any processor request in execution.
- While waiting for the assertion of **$\overline{\text{EOK}}$** signal to issue a processor read/write request, **$\overline{\text{EReq}}$** signal is input to the processor one or more cycles before **$\overline{\text{EOK}}$** signal 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).

12.4.5 External Write Request

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

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

12.4.6 Read Response

A *read response* returns data in response to a processor read request. While a read response is an external request, it has one characteristic that differentiates it from all other external requests—it does not perform System interface arbitration (requesting mastership of the System interface using **$\overline{\text{EReq}}$** signal.

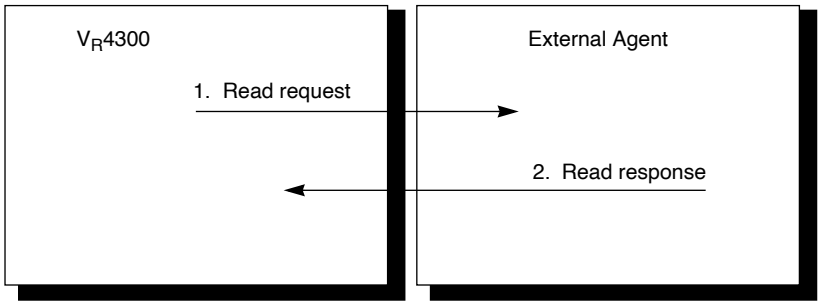


Figure 12-10 Read Response

12.5 Handling Requests

This section details the *sequence*, *protocol*, and *syntax* (Refer to **12.1 Terminology** for definitions of these terms) of both processor and external requests. The following system events are discussed here:

- fetch miss
- load miss
- store miss
- loads/stores to uncached area
- CACHE instructions

12.5.1 Fetch Miss

When the processor misses in the instruction cache on an instruction fetch, it issues a read request for the cache line acquisition. An external agent returns data as a read response.

12.5.2 Load Miss

When the processor misses in the data cache on a load, it issues a read request for the cache line acquisition. An external agent returns data as a read response.

If the cache data to be replaced is in the dirty state, this data is written to the memory. The above read operation must be completed before the data in the dirty state is written.

12.5.3 Store Miss

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

If the cache data to be replaced is in the dirty state, this data is written to the memory. The above read operation must be completed before the data in the dirty state is written.

When it is desirable to guarantee that cached data written by a store instruction is consistent with main memory contents, the corresponding cache line must be written back from the cache to the main memory using a CACHE instruction. CACHE instructions are described in **Chapter 16 CPU Instruction Set Details**.

12.5.4 Loads or Stores to Uncached Area

When the processor performs a load to uncached area, it issues a read request. An external agent returns a single/block transfer as a read response data.

When the processor performs a store to uncached area, it issues a write request and provides a single/block transfer of data to the external agent.

12.5.5 CACHE Instructions

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

12.6 Processor Request 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 12-1 lists the definitions and abbreviations for each of the buses that are used in the timing diagrams that follow.

Table 12-1 System Interface Requests

Scope	Abbreviation	Meaning
Global	Unsd	Unused
SysAD(31:0) bus	Addr	Physical address
	Data<n>	Data element number n of a block of data
SysCmd(4:0) 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

12.6.1 Processor Request Protocols

Processor request protocols described in this section include:

- read
- write

12.6.2 Processor Read Request Protocol

A processor read request is issued by outputting a read command on the **SysCmd(4:0)** bus and a read address on the **SysAD(31:0)** 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 request.

The processor makes an uncompelled change to slave state after the cycle of the read request by deasserting the $\overline{\textbf{PMaster}}$ signal. An external agent then returns the requested data through a read response.

Once the processor enters slave state (starting at cycle 5 in Figure 12-11), 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, indicate to **SysCmd(4:0)** bus that the returned data is erroneous as a read response. If the returned data is erroneous, the processor generates a bus error exception.

Figure 12-11 illustrates a processor read request, coupled with an uncompelled change to slave state, that occurs as the read request is issued. Figure 12-12 shows the processor read request delayed by the **$\overline{\text{EOK}}$** signal.

The following sequence describes the protocol for a processor read request (the numbered steps below correspond to Figures 12-11 and 12-12).

1. The processor is in the master status. It outputs a read command to **SysCmd(4:0)** and a read address to **SysAD(31:0)** to issue a read request. After the read request is issued, the processor enters the pending status. Only one read request can be pending at a time.
2. The processor asserts the **$\overline{\text{PValid}}$** signal to indicate that the current data of **SysCmd(4:0)** and **SysAD(31:0)** are valid.
3. The external agent asserts the **$\overline{\text{EOK}}$** signal for two consecutive cycles to enable issuance of a processor read request. If the **$\overline{\text{EOK}}$** signal is deasserted, the issuance cycle of the read request is delayed.
4. The processor deasserts the **$\overline{\text{PMaster}}$** signal at the first cycle after the read request is accepted, and shifts to the slave status unforcibly.
5. The processor releases **SysCmd(4:0)** and **SysAD(31:0)** at the same time as the **$\overline{\text{PMaster}}$** signal is deasserted.
6. An external agent can drive **SysCmd(4:0)** and **SysAD(31:0)** from the first cycle after the **$\overline{\text{PMaster}}$** signal is deasserted.

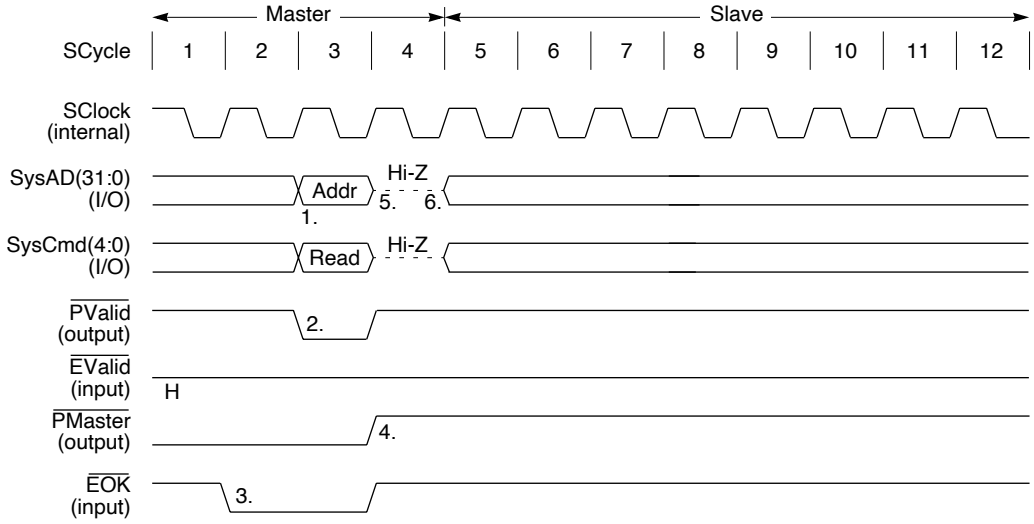


Figure 12-11 Unforcible Transition by Processor Read Request

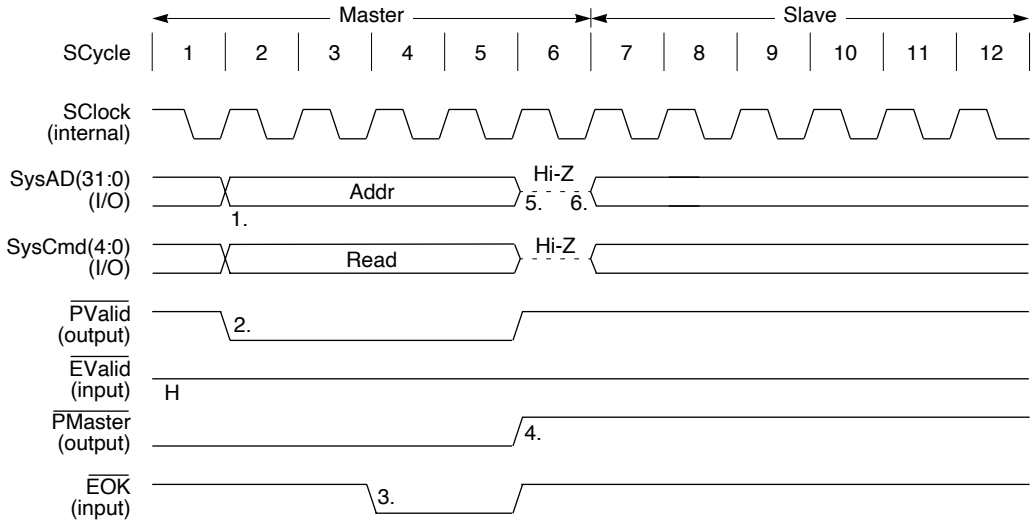


Figure 12-12 Delayed Processor Read Request

12.6.3 Processor Write Request Protocol

A processor write request is issued by outputting a write command on the **SysCmd(4:0)** bus and a write address on the **SysAD(31:0)** bus, and asserting **$\overline{\text{PValid}}$** signal.

After that, a data identifier is output to **SysCmd(4:0)**, write data is output to **SysAD(31:0)**, and the **$\overline{\text{PValid}}$** signal is asserted active to transfer during the cycles necessary for transferring the data. The transfer rate at this time is set by the EP bit of the *Config* register.

The data cycle differs depending on the size of the write request.

- 1 to 4 bytes: Single data cycle
- 5 to 7 bytes: Divided into two single write requests (one is 4 bytes long, and the other is 1 to 3 bytes long)
- 8 bytes or more: Block data cycle in 4-byte units

The last data is appended with a data identifier EOD (End of Data).

Figure 12-13 shows the processor block write request by write data pattern D, and Figure 12-14 shows the processor block write request by write data pattern Dxx.

The following sequence describes the protocol of the processor write request (the numbers correspond to the numbers in Figures 12-13 and 12-14).

1. The processor is in the master status. It outputs a write command to **SysCmd(4:0)** and a write address to **SysAD(31:0)** to issue a write request.
2. The processor asserts the **$\overline{\text{PValid}}$** signal to indicate that the current data of **SysCmd(4:0)** and **SysAD(31:0)** are valid.
3. The external agent asserts the **$\overline{\text{EOK}}$** signal for two consecutive cycles to enable issuance of a processor write request. If the **$\overline{\text{EOK}}$** signal is deasserted, the issuance cycle of the write request is delayed.
4. The processor outputs a data identifier to **SysCmd(4:0)** and write data to **SysAD(31:0)**.
5. The processor asserts the **$\overline{\text{PValid}}$** signal for the cycles necessary for data transfer, and transfer the data.
6. The last data is appended with data identifier EOD.

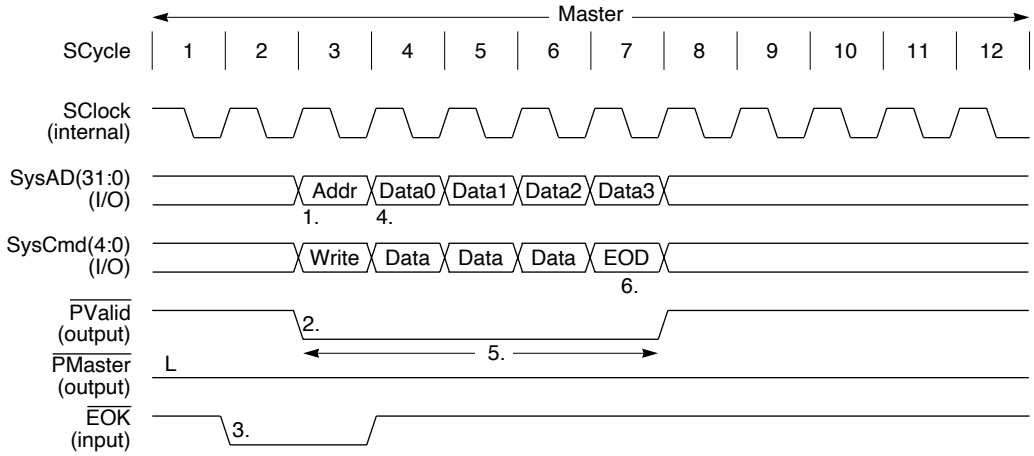


Figure 12-13 Processor Block Write Request (Write Data Pattern: D)

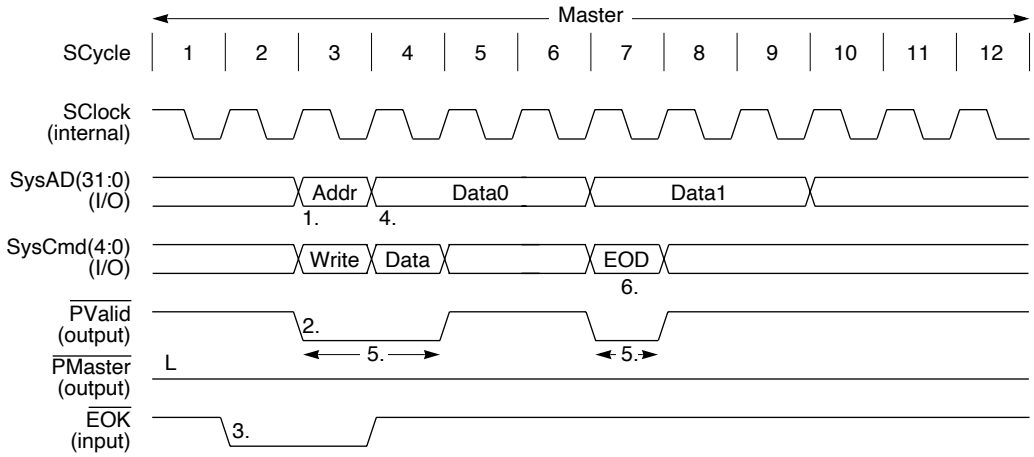


Figure 12-14 Processor Block Write Request (Write Data Pattern: Dxx)

12.6.4 Flow Control of Processor Request

The external agent uses the **$\overline{\text{EOK}}$** signal to control the flow of the processor read request. The processor repeats the current address cycle until the **$\overline{\text{EOK}}$** signal is asserted active. This address cycle continues for 1 cycle after the **$\overline{\text{EOK}}$** signal has been asserted, and then the issuance cycle ends. The **$\overline{\text{EOK}}$** signal must be asserted for at least two consecutive cycles.

Figures 12-15 and 12-16 show how to use the **$\overline{\text{EOK}}$** signal (the numbers in the description below correspond to the numbers in Figures 12-15 and 12-16.

1. Because the **$\overline{\text{EOK}}$** signal 1 cycle before is inactive, the processor request is delayed, and the address cycle does not end.
2. Because the **$\overline{\text{EOK}}$** signal 1 cycle before is active, the processor request is not delayed, and the address cycle ends.

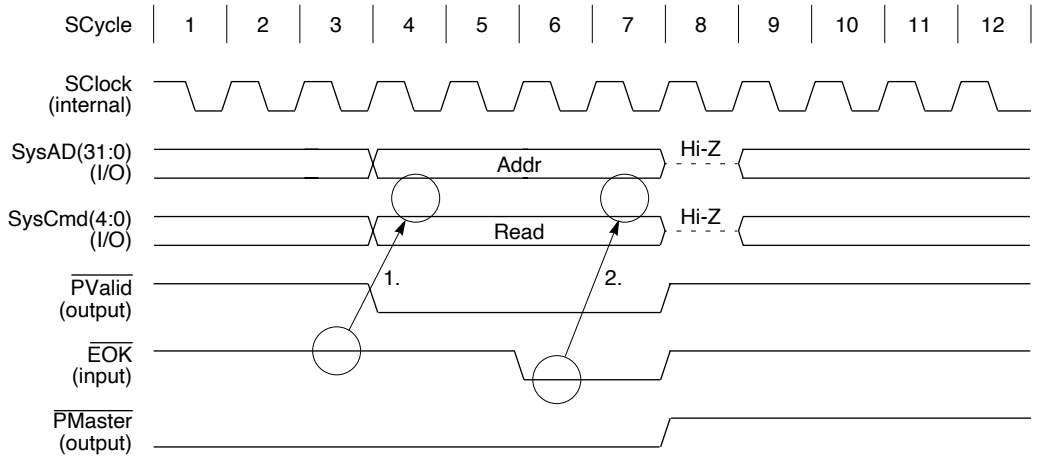


Figure 12-15 Delayed Processor Read Request

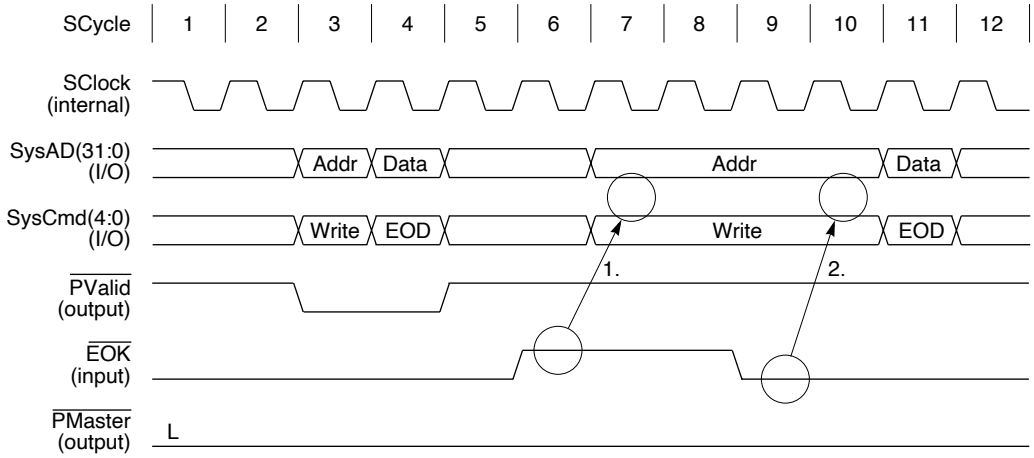


Figure 12-16 Delayed Second Processor Write Request

12.6.5 External Request Protocols

External requests can only be issued with the System interface in slave state. **EReq** signal must be asserted **EReq** signal to arbitrate (refer to **12.6.6 External Arbitration Protocol**) 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 mastership of the System interface to the processor, as described below.

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

- write
- read response

12.6.6 External Arbitration Protocol

Usually, the processor serves as the bus mastership. However, the processor relinquishes control of the bus and enters the slave status in the following cases.

- If the external agent issues a request and the system interface responds to that request
- After the processor has issued a read request

Arbitration to allow the processor to enter the slave status from the master status is realized by using the handshake signals (**$\overline{\text{EReq}}$**, **$\overline{\text{PReq}}$**, and **$\overline{\text{PMaster}}$**) of the system interface.

Status Transition On Read Response

While the processor read request is kept pending, the processor enters the slave status by deasserting the **$\overline{\text{PMaster}}$** signal inactive, and the external agent returns read response data.

If the **$\overline{\text{EReq}}$** signal is deasserted inactive, the processor remains in the slave status until the read response data is returned, and then returns to the master status by asserting the **$\overline{\text{PMaster}}$** signal active.

The external agent can remain in the master status as long as the **$\overline{\text{EReq}}$** signal remains active when the read response is returned.

Acquiring Bus Mastership by $\overline{\text{EReq}}$ Signal

If the processor is in the master status when the external agent has issued an external request, assert the **$\overline{\text{EReq}}$** signal active and wait until the processor deasserts the **$\overline{\text{PMaster}}$** signal inactive. If the processor deasserts the **$\overline{\text{PMaster}}$** signal inactive, the external agent acquires the bus mastership.

Once the external agent has entered the master status, it can remain in the master status as long as the **$\overline{\text{EReq}}$** signal is asserted active. When the **$\overline{\text{EReq}}$** signal is deasserted, the processor acquires the bus mastership two cycles later.

Figure 12-17 shows the arbitration protocol of the external request issued by the external agent.

The following sequence describes the arbitration protocol (the numbers in the sequence correspond to the numbers in Figure 12-17).

1. The external agent continues asserting the **$\overline{\text{EReq}}$** signal active to issue an external request.
2. When the processor is ready to process the external request, it deasserts the **$\overline{\text{PMaster}}$** signal inactive.
3. The processor sets **SysAD(31:0)** and **SysCmd(4:0)** in the high-impedance state.
4. The external agent should drive **SysAD(31:0)** and **SysCmd(4:0)** one cycle after the **$\overline{\text{PMaster}}$** signal has been deasserted inactive.
5. The external agent should deassert the **$\overline{\text{EReq}}$** signal inactive in the last cycle of the external request (2 cycles before the external agent enters the slave status), except when it executes another external request.
6. The external agent should set **SysAD(31:0)** and **SysCmd(4:0)** in the high-impedance state on completion of the external request.

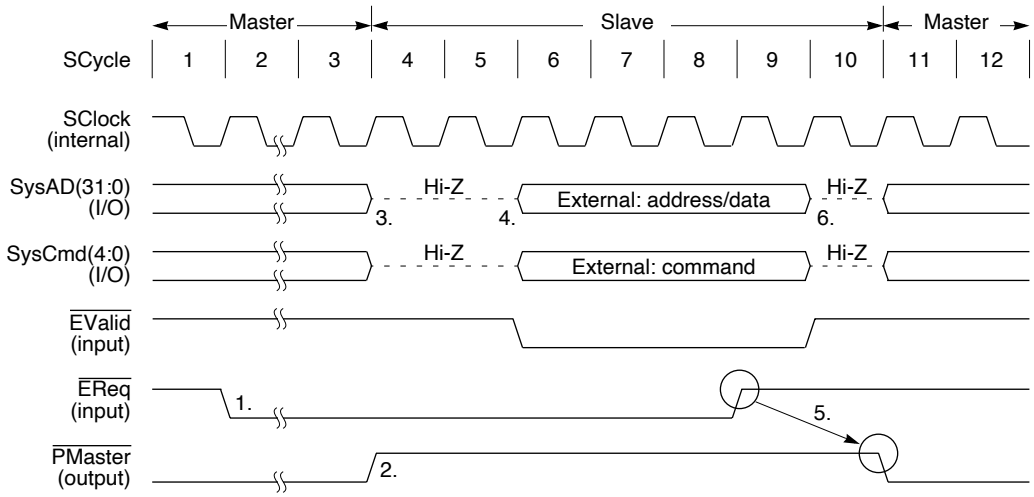


Figure 12-17 Arbitration of External Request

If the external agent has entered the master status by issuing the processor read request, the external agent must always return read request data. If the external agent has entered the master status by using the **$\overline{\text{EReq}}$** signal, any command and data can be issued in accordance with the arbitration process. This means that the processor always satisfies any request from the external agent.

Restoring Bus Mastership by $\overline{\text{PReq}}$ Signal

Once the external agent has entered the master status, the processor cannot stop the operation of the external agent. However, the processor can request bus mastership by asserting the **$\overline{\text{PReq}}$** signal. At this time, the external agent must deassert the **$\overline{\text{EReq}}$** signal inactive in response to the request by the processor, giving consideration to the priority of the mastership.

The processor asserts the **$\overline{\text{PMaster}}$** signal two cycles after the **$\overline{\text{EReq}}$** signal has deasserted to inform the external agent that the processor has regained the bus mastership.

Figure 12-18 illustrates how the processor requests the bus mastership and how the external agent releases the bus in response.

At reset (when the **$\overline{\text{Reset}}$** or **$\overline{\text{ColdReset}}$** signal is active), the processor enters the master status, and the external agent enters the slave status.

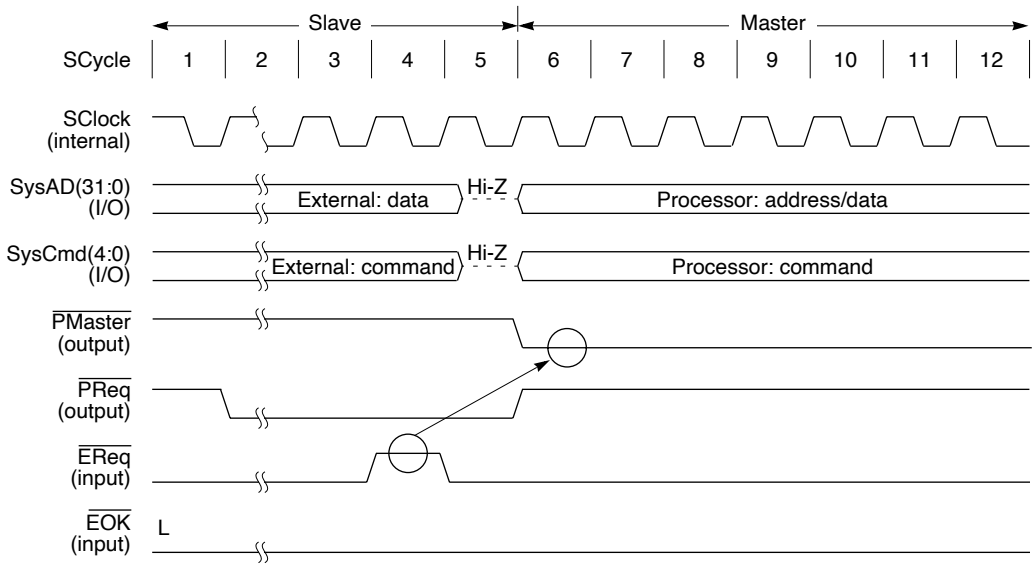


Figure 12-18 Bus Arbitration of Processor

12.6.7 External Write Request Protocol

External write requests are similar in operation to a processor single write except that the **$\overline{\text{EValid}}$** signal is asserted in place of the **$\overline{\text{PValid}}$** signal.

An external write request outputs a write command on the **SysCmd(4:0)** bus and a write address on the **SysAD(31:0)** bus when the processor is in slave state and asserting **$\overline{\text{EValid}}$** signal for one cycle. This is followed by outputting a data identifier on the **SysCmd(4:0)** bus and data on the **SysAD(31:0)** bus and asserting **$\overline{\text{EValid}}$** signal for one more cycle. The data identifier of the data cycle must contain an end of data cycle indication.

Keep the **$\overline{\text{EReq}}$** signal active while the external write request is issued.

After the data cycle is issued, the write request is completed and the external agent releases the **SysCmd(4:0)** and **SysAD(31:0)** buses and allows the system interface to return to master state.

An external write request with the processor generated in master state is illustrated in Figure 12-19.

Figure 12-22 shows an example in which the external agent issues an external write request following a read response. The external write request cannot be issued while read response data is transferred. It can be issued before data response or after the last data response.

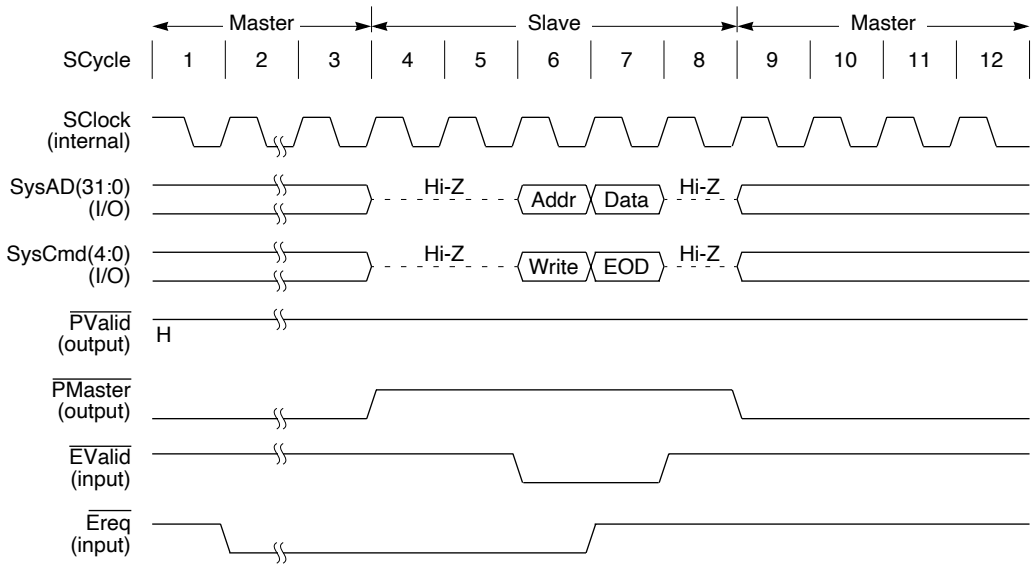


Figure 12-19 External Write Request Protocol

Only an interrupt processing can be done by the processor in the external write request.

12.6.8 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 size.

The **SysCmd(4:0)** and **SysAD(31:0)** buses are released after the last data cycle is issued. If the **$\overline{\text{ERq}}$** signal is inactive at this time, the processor returns to master state at the end of two cycles after the last data cycle.

The data identifier associated with a data cycle may indicate that data transferred during this cycle is erroneous; however, an external agent must return a specific data block whether or not the data is erroneous. If a read response includes one or more erroneous data cycles, the processor generates a bus error exception.

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

A processor single read request followed by a read response is illustrated in Figure 12-20. A read response for a processor block read with the processor already in slave state is illustrated in Figure 12-21.

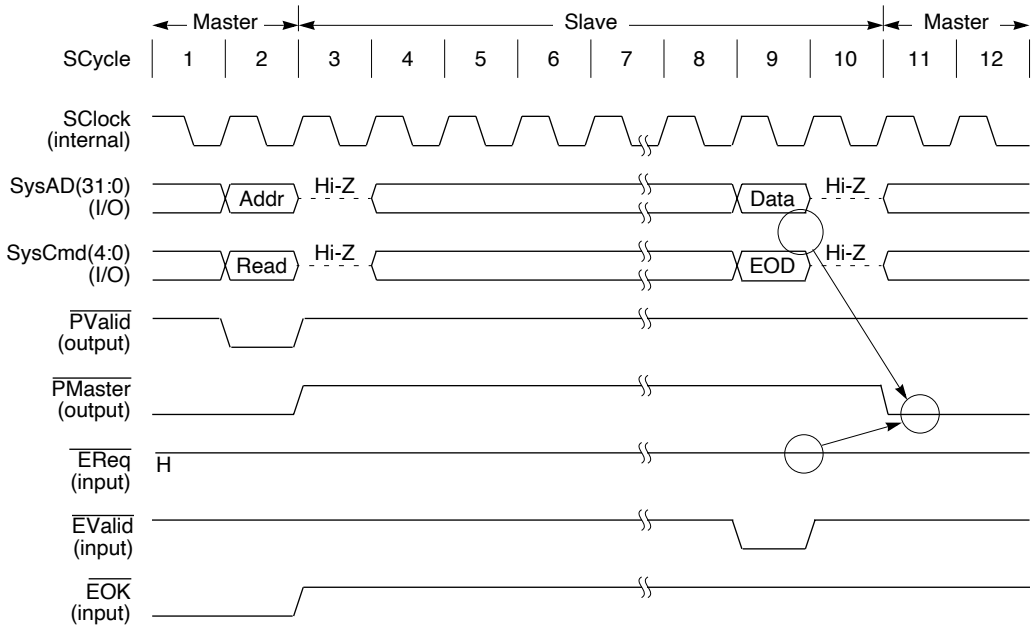


Figure 12-20 Read Request/Read Response Protocol

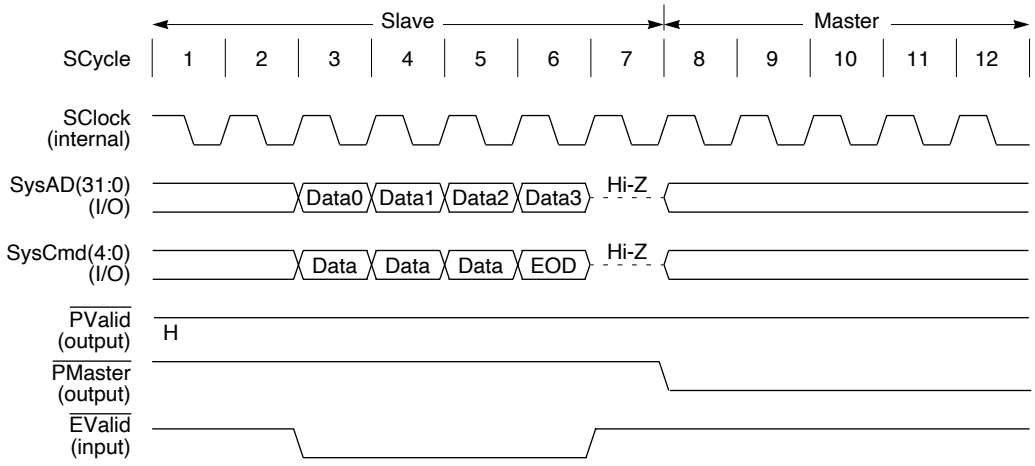


Figure 12-21 Block Read Response in Slave Status

Figure 12-22 shows the case where an external write request is issued following a read response to a processor single read request. The following sequence describes the protocol (the numbers in the following description correspond to the numbers in Figure 12-22).

1. The external agent returns response data to the processor single read request.
2. To issue an external request following the read response, assert the **$\overline{\text{EReq}}$** signal active in the cycle in which EOD is returned. In this case, the **$\overline{\text{PMaster}}$** signal remains inactive two cycles after EOD.
3. Because the external agent is in the master status, it can issue the external write request.
4. Deassert the **$\overline{\text{EReq}}$** signal inactive up to the data cycle of the external write request. In this case, the **$\overline{\text{PMaster}}$** signal is asserted active two cycles after EOD, and the bus mastership is returned to the processor.

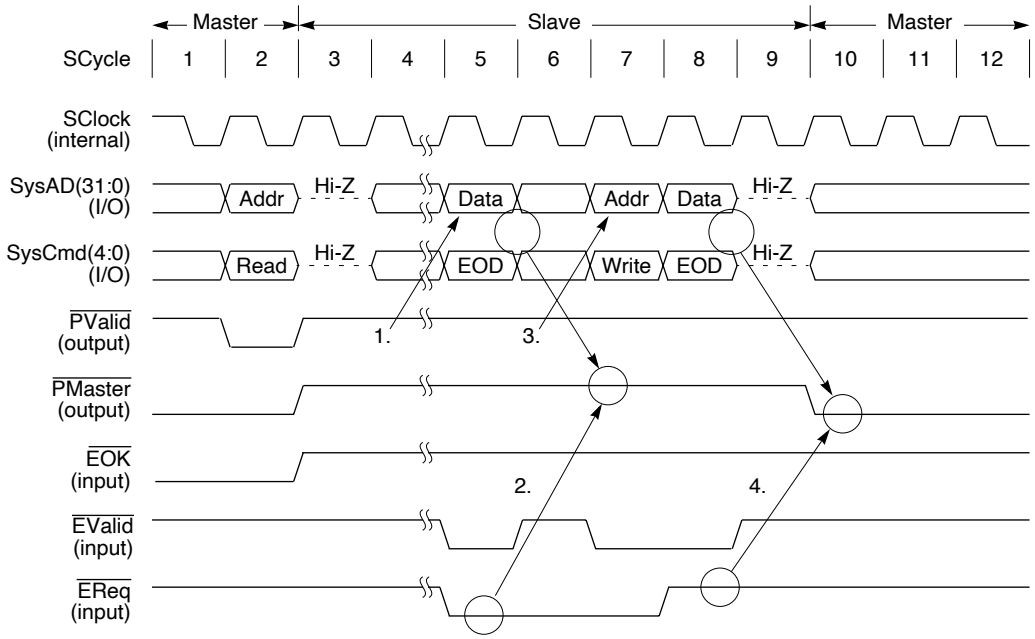


Figure 12-22 External Write Request Following Read Response

Figure 12-23 shows an example in which an external write request interrupts a read response to a processor single read request. Cycle 5 in the figure is the write data for the external write request in cycle 4, and cycle 7 is the read response data.

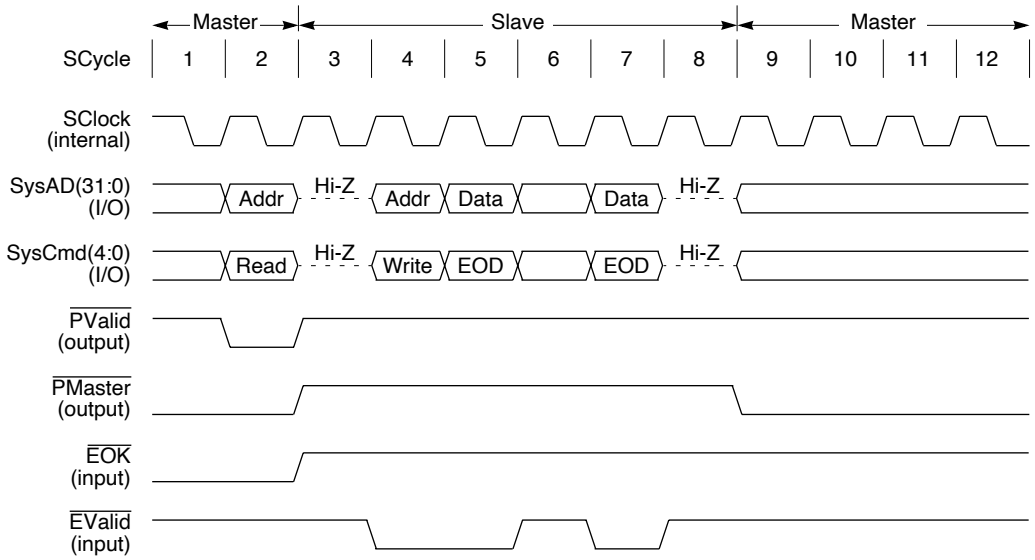


Figure 12-23 When External Write Request Takes Precedence While Processor Read Request is Pending

As shown in this figure, even if the external request interrupts the processor read request, the processor remains in the slave status until the read response data is returned.

12.7 Successive Processing of Request

12.7.1 Successive Processor Write Requests

The processor write requests may be successively operated as follows.

- In the case of data pattern “D”
  In this case, the processor write requests are processed without wait status as shown in Figure 12-24.
- In the case of data pattern “Dxx”
  In this case, the processing is separated by a wait status of two cycles as shown in Figure 12-25.

The processor write requests may be successively issued in the following four cases.

1. Successive single write requests
2. Successive block write requests
3. Block write request after single write request
4. Single write request after block write request

For the timing of the processor single write request, refer to **12.6.3 Processor Write Request Protocol**.

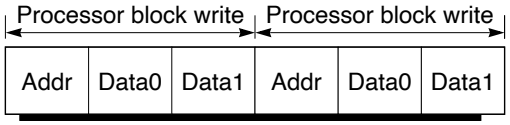


Figure 12-24 Successive Block Write Requests (Write Data Pattern: D)

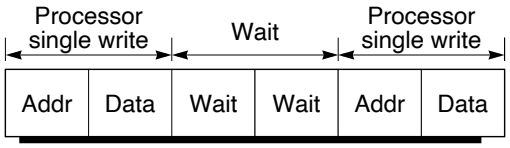


Figure 12-25 Successive Single Write Requests (Write Data Pattern: Dxx)

12.7.2 Processor Write Request Followed by Processor Read Request

Figure 12-26 shows the case where a processor read request follows a processor write request.

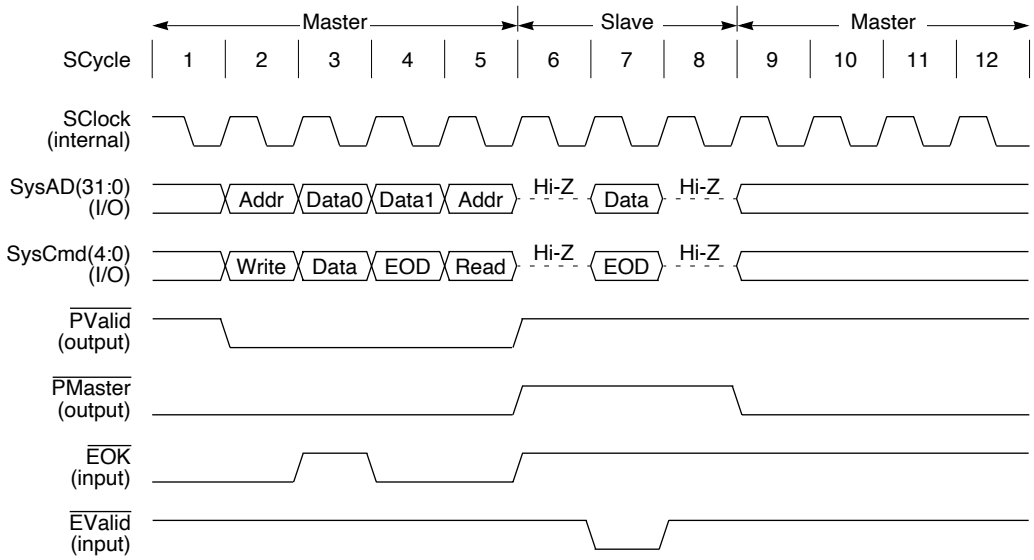


Figure 12-26 Processor Write Request Followed by Processor Read Request (Write Data Pattern: D)

12.7.3 Processor Read Request Followed by Processor Write Request

Figure 12-27 shows the case where a processor read request is followed by a processor write request.

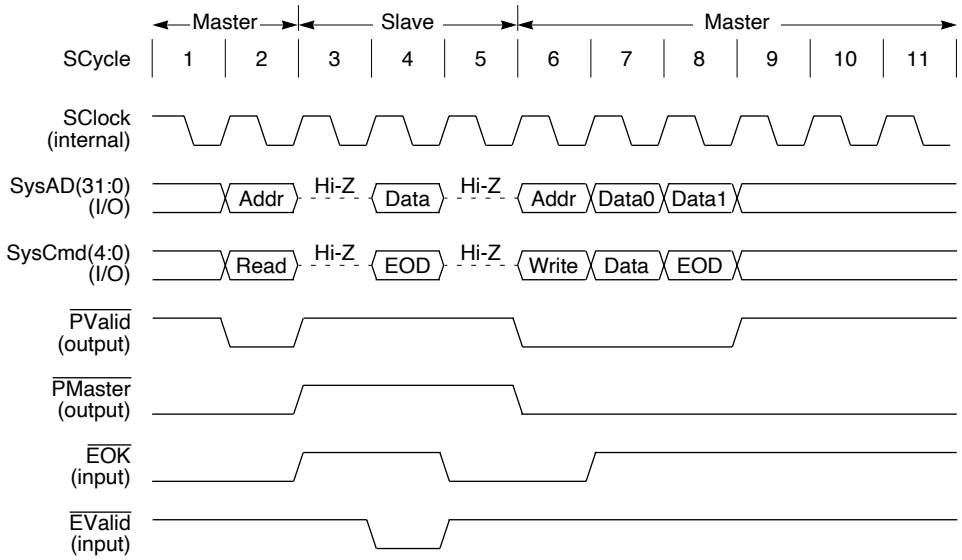


Figure 12-27 Processor Single Read Request Followed by Block Write Request (Write Data Pattern: D)

12.7.4 Processor Write Request Followed by External Write Request

Figure 12-28 shows the case where processor write requests are followed by an external write request.

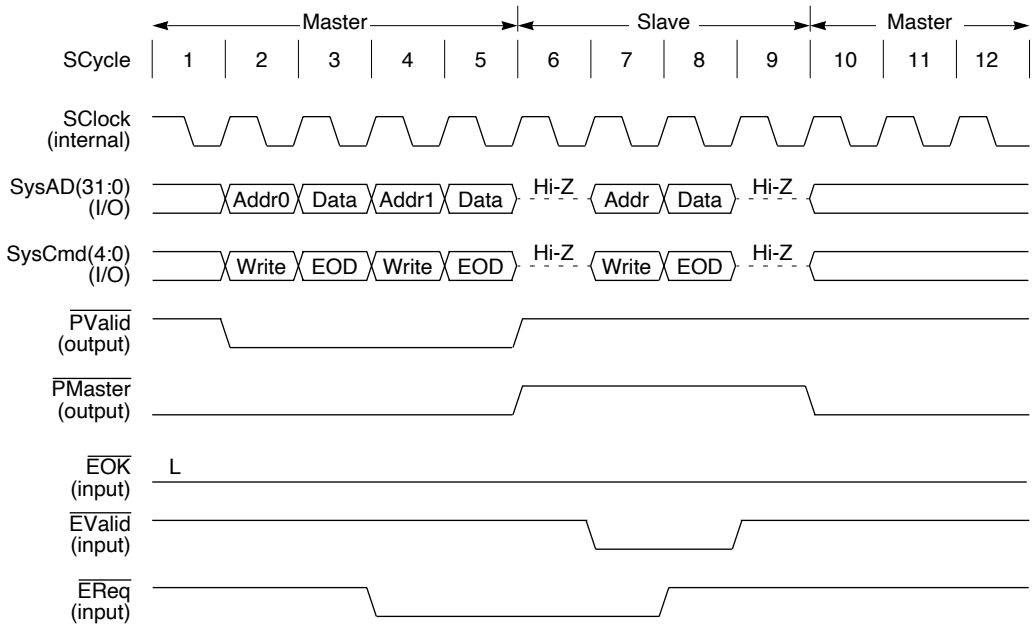


Figure 12-28 Successive Processor Write Requests Followed by External Write Request (Write Data Pattern: D)

12.8 Discarding and Re-Executing Commands

12.8.1 Re-Execution of Processor Commands

The external agent executes and controls the processor commands by using the **$\overline{\text{EOK}}$** signal. When the processor serves as the master, the processor cannot issue a command until the **$\overline{\text{EOK}}$** signal is active for at least two cycles.

If the **$\overline{\text{EOK}}$** signal is active for only one cycle before the processor issues a command and then becomes inactive in the next cycle in which the command is issued, this processor command is discarded. At this time, the external agent should ignore the discarded command.

If Write Command is Discarded

The processor issues write data and then the write command again. At this time, the external agent should ignore the write data following the discarded write command.

If Read Command is Discarded

The processor enters the slave status in the cycle following the address cycle of a read request. If the **$\overline{\text{EReq}}$** signal is inactive at this time, the processor returns to the master status again one cycle later, and reissues a read request.

12.8.2 Discarding and Re-Executing Write Command

Figure 12-29 illustrates how a processor single write request is discarded and re-executed. The following sequence describes the protocol (the numbers in the following description correspond to the numbers in Figure 12-29).

1. Because the **$\overline{\text{EOK}}$** signal is active one cycle before (cycle 2) the write request of Data0, this cycle is the issuance cycle.
2. Because the **$\overline{\text{EOK}}$** signal is active in the write request cycle of Data0 (cycle 3), the next cycle is a normal data cycle.
3. Because the **$\overline{\text{EOK}}$** signal is active in one cycle (cycle 4) before the write request of Data1, this cycle is the issuance cycle.
4. Because the **$\overline{\text{EOK}}$** signal is inactive in the write request cycle of Data1 (cycle 5), the data of the next cycle is discarded. At this time, data/command is output to **SysAD(31:0)** and **SysCmd(4:0)**, which should be ignored by the external agent.
5. Because the **$\overline{\text{EOK}}$** signal is inactive one cycle (cycle 6) before the write request of the second Data1, the write request is delayed.

6. Because the **$\overline{\text{EOK}}$** signal is active in one cycle (cycle 9) before the write request of the second Data1, this cycle is the issuance cycle.
7. Because the **$\overline{\text{EOK}}$** signal is active in the write request cycle (cycle 10) of the second Data1, the next cycle is a normal data cycle.

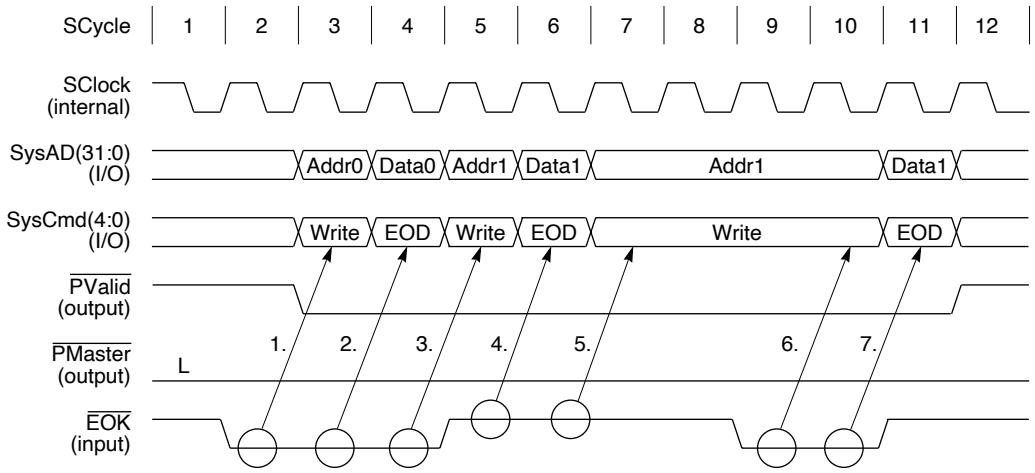


Figure 12-29 Discarding and Re-executing Processor Single Write Request

12.8.3 Discarding and Re-Executing Read Command

Figure 12-30 illustrates how a processor single read request is discarded and re-executed. The following sequence describes the protocol (the numbers in the following description correspond to the numbers in Figure 12-30).

1. Because the **$\overline{\text{EOK}}$** signal is low in cycle 5, the processor tries to issue an address (cycle 6).
2. If the **$\overline{\text{EOK}}$** signal is high at this point, the processor discards this read request and enters the slave status in the next cycle.
3. Because the **$\overline{\text{EReq}}$** signal is inactive, the processor returns to the master status again and reissues a read request. Because the **$\overline{\text{EOK}}$** signal is low in both the cycles 7 and 8, the issuance cycle of the read request is determined.
4. The external agent outputs data at the requested address.

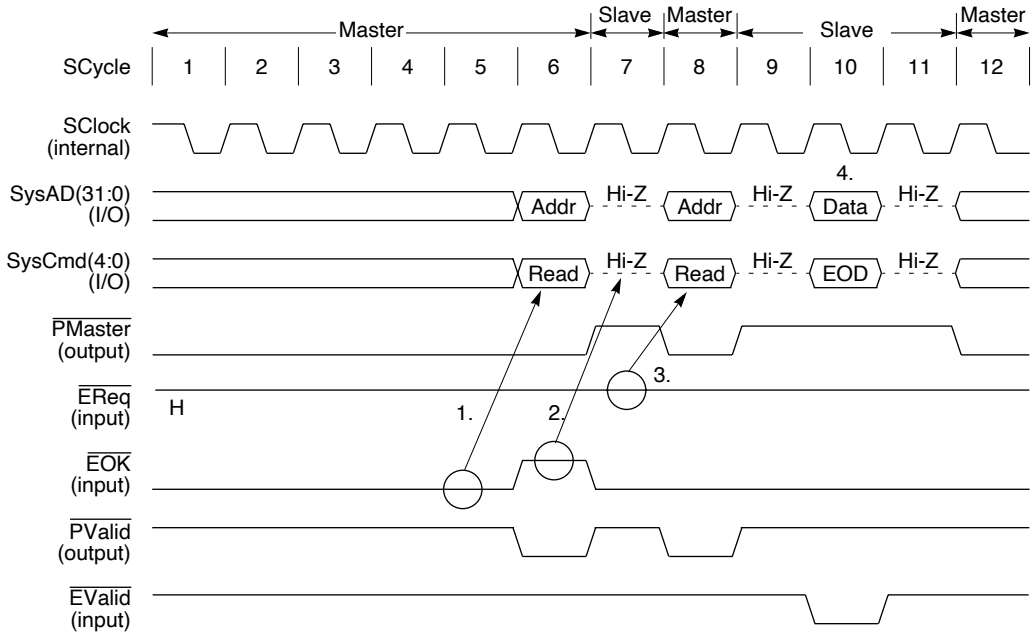


Figure 12-30 Discarding and Re-executing Processor Single Read Request

12.8.4 Executing and Discarding Command

When External Agent Requests Bus Mastership

The external agent requests the bus mastership by asserting the **$\overline{\text{EReq}}$** signal active. At this time, the external agent can acquires the bus mastership after it has accepted one processor read/write request only, or without accepting any request.

If the **$\overline{\text{EReq}}$** signal is asserted active while the external agent delays the processor request by deasserting **$\overline{\text{EOK}}$** signal inactive, the external agent can forcibly acquires the bus mastership.

When Processor Requests Bus Mastership

The processor requests the bus mastership by asserting the **$\overline{\text{PReq}}$** signal active. At this time, the external agent should transfer the bus mastership to the processor, giving consideration to the priority of the system. If the external agent keeps the **$\overline{\text{EReq}}$** signal inactive for more than one cycle, the bus is released.

The processor acquires the bus mastership by asserting the **$\overline{\text{PMaster}}$** signal active two cycles after the **$\overline{\text{EReq}}$** signal has become inactive. If the **$\overline{\text{EOK}}$** signal is active at this time, the processor can issue a request.

Figure 12-31 shows an example where the external agent has entered the slave status (the **$\overline{\text{EReq}}$** signal is inactive) from the master status, and then acquires the bus mastership again after accepting one processor request.

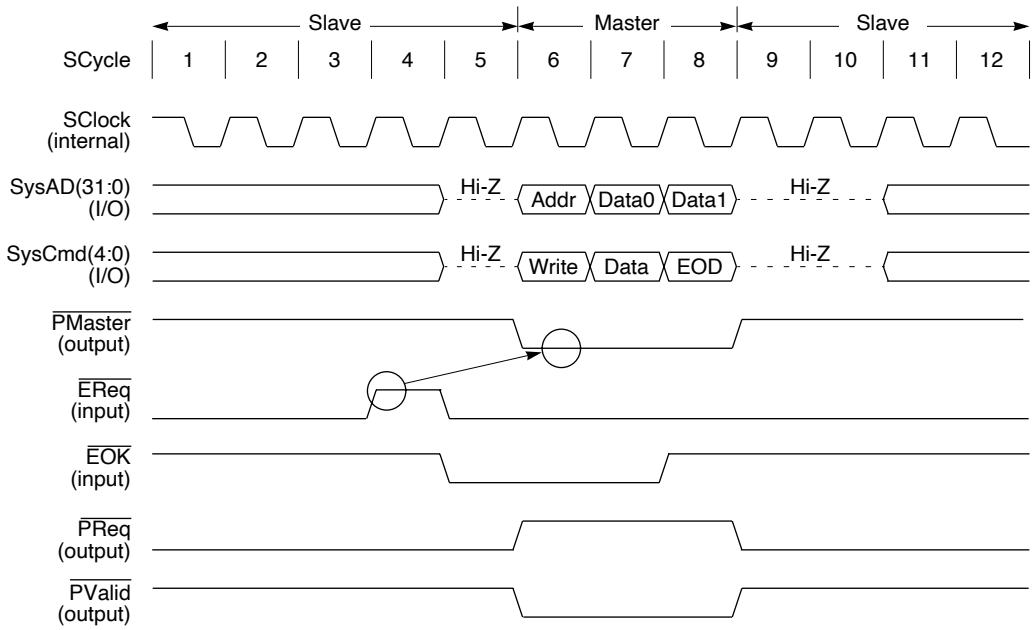


Figure 12-31 Discarding Bus Mastership by External Agent by Processor Request

12.9 Data Flow Control

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

Read Response

An external agent may transfer data to the processor at the maximum data rate of the System interface. The rate at which data is transferred to the processor can be controlled by the external agent, which asserts **$\overline{\text{EValid}}$** signal at the cycle which data is transferred. The processor accepts cycles as valid only when **$\overline{\text{EValid}}$** signal is asserted and the **SysCmd(4:0)** bus contains a data identifier; thereafter, the processor continues to accept data until it receives the data word tagged as the last one.

Data identifier EOD must be attached to the last data word. Without this, the System interface hangs up as a protocol error. In this case, because the protocol error state is identified with the **$\overline{\text{PReq}}$** signal at double the cycle of **SClock** oscillating in synchronization with the **MasterClock**, the processor should be reset and initialized.

Write Request

The rate at which the processor transfers data to an external agent is programmable through the *EP* bit of the *Config* register (setting at reset is D) signal. Data patterns are defined using the letters **D** and **x**, where **D** indicates a data cycle and **x** indicates an unused cycle. For example, a **Dxx** data pattern indicates a data rate of one word every three cycles.

The V_R4300 has two data transfer rates: **D** and **Dxx**. The processor continues outputting data output in the period of D immediately before, while the processor is in the master status and during the period of x.

A processor block write request with a **Dxx** data pattern (one word every three cycles) is shown in Figure 12-14.

12.9.1 Independent Transfer on SysAD(31:0) Bus

In general applications, the **SysAD(31:0)** bus is a point-to-point connection, running from the processor to a bidirectional register transceiver residing in an external agent. For these applications, the **SysAD(31:0)** bus has only two possible devices to connect, the processor or the external agent.

Certain applications may require connection of additional drivers and receivers to the **SysAD(31:0)** bus, to allow transfers over the **SysAD(31:0)** bus that the processor is not involved in. These are called *independent transfers*. To effect an independent transfer, the external agent must coordinate mastership of the **SysAD(31:0)** bus by using arbitration handshake signals (**$\overline{\text{EReq}}$**, **$\overline{\text{PMaster}}$** and **$\overline{\text{PReq}}$** signals).

An independent transfer on the **SysAD(31:0)** bus follows this procedure:

1. The external agent asserts **$\overline{\text{EReq}}$** signal, and requests mastership of the **SysAD(31:0)** bus, to issue an external request.
2. The processor deasserts **$\overline{\text{PMaster}}$** signal, and releases the System interface to slave state.
3. The external agent then allows the independent transfer to take place on the **SysAD(31:0)** bus, making sure that **$\overline{\text{EValid}}$** signal is not asserted during the transfer.
4. When the transfer is completed, the external agent deasserts **$\overline{\text{EReq}}$** signal to return the System interface to master state.

To connect multiple devices, separate enable signals for device to input/output are required to allow the non-processor chips to communicate.

12.9.2 System Endianness

The endianness of the system is set by the *BE* bit of the *Config* register: byte order is big endian when this bit is set to 1, and little endian when this bit is set to 0. This bit is set to 1 at cold reset. Set this bit first in the initial sequence with a little endian system.

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

12.10 System Interface Cycle Time

The processor specifies minimum and maximum cycle counts for the time required for various processor transactions and for the processor response time to external requests. Processor requests themselves are constrained by the System interface 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.

12.10.1 Release Latency Time

Release latency time is 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 time is therefore the number of cycles when **$\overline{\text{EReq}}$** signal becomes active until **$\overline{\text{PMaster}}$** signal becomes inactive.

There are two categories of release latency time:

- Category 1: when the **$\overline{\text{EReq}}$** signal is asserted by one cycle before the last cycle of a processor request.
- Category 2: when the **$\overline{\text{EReq}}$** signal is not asserted during a processor request, or is asserted during the last cycle of a processor request.

Table 12-2 shows the minimum and maximum release latency time for requests that fall into categories 1 and 2. Note that the maximum and minimum cycle counts are subject to change.

Table 12-2 Release Latency Time for External Requests

Category	Minimum PCycles	Maximum PCycles
1	4	6
2	4	24

12.11 System Interface Commands and Data Identifiers

System interface commands specify the types 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 transferred 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 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 commands and data identifiers are undefined.

12.11.1 Command and Data Identifier Syntax

System interface commands and data identifiers are encoded in 5 bits and are transferred on the **SysCmd(4:0)** 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(4:0)** 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, **SysCmd4** must be set to 0. For System interface data identifiers, **SysCmd4** must be set to 1.

Bit	Meaning
SysCmd4	Attributes. 0: Command (address) 1: Data identifier

12.11.2 System Interface Command Syntax

This section describes the **SysCmd(4:0)** bus encoding for System interface commands. Figure 12-32 shows a common encoding used for all System interface commands.

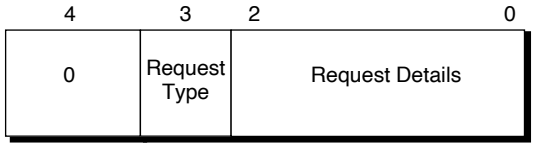


Figure 12-32 System Interface Command Syntax Bit Definition

SysCmd4 must be set to 0 for all System interface commands.

SysCmd3 specify the System interface request type which may be read or write.

Table 12-3 Encoding of SysCmd3 for System Interface Commands

Bit	Meaning
SysCmd3	Command. 0: Read Request 1: Write Request

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

12.11.3 Read Requests

For read requests, the encoding of the **SysCmd(2:0)** is as follows.

Figure 12-33 shows the format of a **SysCmd** read request.

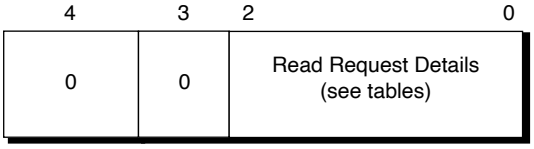


Figure 12-33 Read Request SysCmd(4:0) Bus Bit Definition

Tables 12-4 through 12-6 list the encodings of **SysCmd(2:0)** bit read attributes for read requests.

Table 12-4 Encoding of SysCmd2 for Read Requests

Bit	Meaning
SysCmd2	Read Attributes. 0: Single Read 1: Block Read

Table 12-5 Encoding of SysCmd(1:0) for Block Read Requests

Bit	Meaning
SysCmd(1:0)	Read Block Size. 0: 2 words 1: 4 words (D-cache only) 2: 8 words (I-cache only) 3: Reserved

Table 12-6 Encoding of SysCmd(1:0) for Single Read Requests

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

12.11.4 Write Requests

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

Figure 12-34 shows the format of a **SysCmd** write request.

Table 12-7 lists the write attributes encoded in bits **SysCmd2**. Table 12-8 lists the block write replacement attributes encoded in bits **SysCmd(1:0)**. Table 12-9 lists the single write request encoded in bits **SysCmd(1:0)**.

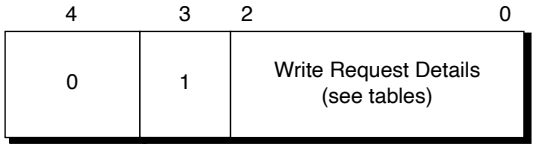


Figure 12-34 Write Request SysCmd(4:0) Bus Bit Definition

Table 12-7 Encoding of SysCmd2 for Write Requests

Bit	Meaning
SysCmd2	Write Attributes. 0: Single Write 1: Block Write

Table 12-8 Encoding of SysCmd(1:0) for Block Write Requests

Bit	Meaning
SysCmd(1:0)	Write Block Size. 0: 2 words 1: 4 words (for D-cache only) 2: 8 words (for I-cache only) (for test) 3: Reserved

Table 12-9 Encoding of SysCmd(1:0) for Single Write Requests

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

12.11.5 System Interface Data Identifier Syntax

This section defines the encoding of the **SysCmd(4:0)** bus for System interface data identifiers. Figure 12-35 shows a common encoding used for all System interface data identifiers.

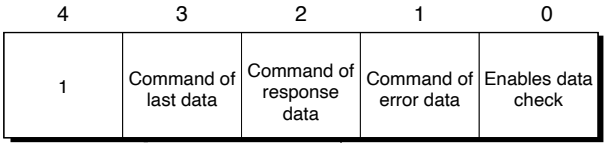


Figure 12-35 Data Identifier SysCmd(4:0) Bus Bit Definition

SysCmd4 must be set to 1 for all System interface data identifiers.

12.11.6 Data Identifier Bit Definitions

Bit definitions of **SysCmd(3:0)** are described next.

SysCmd3 marks the last data element.

SysCmd2 indicates whether or not the data is response data. Response data is data returned in response to a read request.

SysCmd1 indicates whether or not the data element is error free. Erroneous data contains an uncorrectable error and is returned to the processor, resulting a bus error exception. Because the V_R4300 does not have a parity check function, the processor does not transfer data by setting the error bit to 1.

SysCmd0 enables data check (reserved function).
Because the V_R4300 does not have a data check function, the processor outputs 1 (data check disable) when it transfers data. When the external agent transfers data, the processor ignores this bit. But set this bit to 1 to disable checking.

Table 12-10 lists the encodings of **SysCmd(3:0)** for processor data identifiers.
Table 12-11 lists the encodings of **SysCmd(3:0)** for external data identifiers.

Table 12-10 Processor Data Identifier Encoding of SysCmd(3:0)

Bit	Meaning
SysCmd3	Last Data Element Indication. 0: Last data element, or data element on single transfer 1: Not the last data element
SysCmd2	Reserved
SysCmd1	Reserved: Error Data Indication. The processor outputs 0 (error free).
SysCmd0	Reserved: Data check enabled Processor outputs 1 (data check disabled).

Table 12-11 External Data Identifier Encoding of SysCmd(3:0)

Bit	Meaning
SysCmd3	Last Data Element Indication. 0: Last data element or data element on single transfer 1: Not the last data element
SysCmd2	Response Data Indication. 0: Data is response data 1: Data is not response data
SysCmd1	Error Data Indication. 0: Data is error free 1: Data is erroneous
SysCmd0	Reserved: Data Checking Enable. Processor ignores this bit. (external agent transfers 1)

12.12 System Interface Addresses

System interface addresses are full 32-bit physical addresses output to the **SysAD(31:0)** bus during address cycles.

12.12.1 Addressing Conventions

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

- Addresses associated with block requests are aligned to requested doubleword 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.

12.12.2 Sequential and Subblock Ordering

Sequential Ordering

An instruction cache read request returns data in sequential order, starting with the first word (DW0) of the 8-word block, no matter which word is requested.

Subblock Ordering

When a read request is issued to the data cache, the low-order word of the doubleword that includes the word required by the CPU is first returned, and then the high-order word, the low-order word of the remaining doubleword, and the high-order word of it is returned in that order (for details, refer to **12.2.1 Physical Addresses**).

[MEMO]

JTAG Interface

13

The V_R4300 processor is provided with a boundary-scan interface that is compatible with Joint Test Action Group (JTAG) specifications, conforming to the industry-standard JTAG protocol (IEEE Standard 1149.1/D6).

This chapter describes the functions related to JTAG interface.

13.1 Principles of Boundary Scanning

With the evolution of integrated circuits (ICs), surface-mounted devices, double-sided component mounting on printed-circuit boards (PCBs), and via hole technology, in-circuit tests connected to boards and chips have become more and more difficult to perform. The greater complexity of ICs has also meant that testing all the circuits in a chip have become much larger in size of the test pattern and more difficult to write.

One solution to this difficulty has been the development of testing method using *boundary-scan* circuits. A boundary-scan circuit is shift register organization of a series of connected cells placed between each pin of the chip and the internal circuitry of the IC, as shown in Figure 13-1. In normal operation these boundary-scan cells are bypassed; in the test mode, however, the scan cells are directed by the test program to pass data along the shift register path and perform various diagnostic tests. To accomplish this, the tests use the four signals described in the next section: **JTDI, JTDO**, **JTMS**, and **JTCK**.

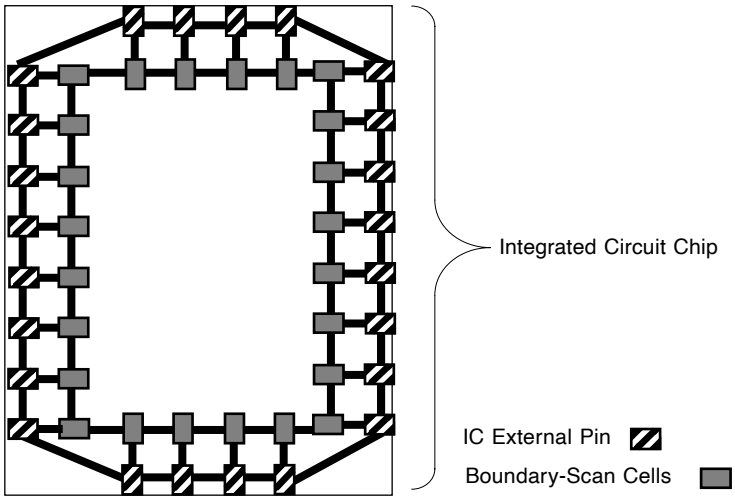


Figure 13-1 JTAG Boundary-Scan Cells

13.2 Signal Summary

The JTAG interface signals used are listed below.

JTDI	JTAG serial data input
JTDO	JTAG serial data output
JTMS	JTAG test mode select
JTCK	JTAG serial clock input

Caution When the JTAG interface is not used, keep the JTCK signal low.

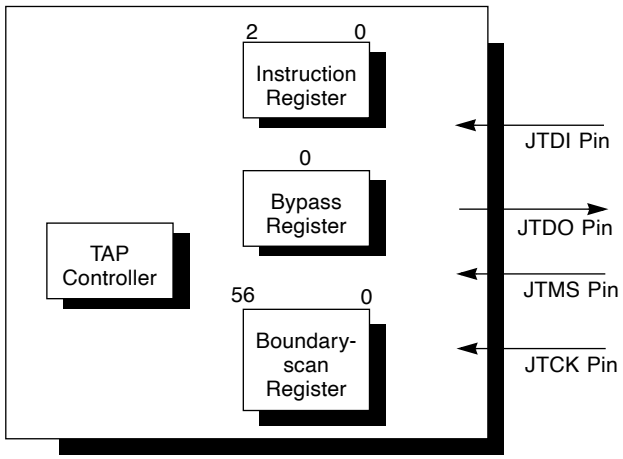


Figure 13-2 JTAG Interface Signals and Registers

The JTAG boundary-scan mechanism (referred to as *JTAG mechanism* in this chapter) allows testing of the connections between the processor, the printed circuit board to which it is attached, and the other device on the board.

The JTAG mechanism does not provide any capability for testing the processor itself.

13.3 JTAG Controller and Registers

The processor contains the following registers and JTAG controller:

- *Instruction* register
- *Boundary-scan* register
- *Bypass* register
- Test Access Port (TAP) controller

The processor executes the standard JTAG EXTEST operation associated with External Test function testing.

The basic operation of JTAG is for the TAP controller state machine to monitor the JTMS input signal, as shown in Table 13-1. When it starts, the TAP controller determines the test function to be implemented. This includes either loading an instruction register (IR), or beginning a serial data scan through a data register (DR). As the data is scanned in, the state of the JTMS pin transmits each new data word, and indicates the end of the data stream. The data register to be selected is determined by the contents of the *Instruction* register.

13.3.1 Instruction Register

The JTAG *Instruction* register includes three shift register-organization cells; this register is used to select the test to be performed and the test data register to be accessed. As listed in Table 13-1, the register value setting selects either the *Boundary-scan* register or the *Bypass* register.

Table 13-1 JTAG Instruction Register Bit Encoding

MSB. . . . . LSB	Data Register
0 0 0	Boundary-scan register (external test only)
0 1 1	Setting prohibited
Others	Bypass register

The *Instruction* register has two stages: shift register, and parallel output latch. Refer to **13.3.7 Controller States** for detail. Figure 13-3 shows the format of the *Instruction* register.

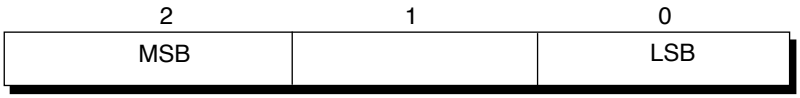


Figure 13-3 Instruction Register

13.3.2 Bypass Register

The *Bypass* register is 1 bit wide. When the TAP controller is in the Shift-DR (Bypass) state, the data on the **JTDI** pin is shifted into the *Bypass* register, and the data on *Bypass* register output shifts to the **JTDO** output pin.

Actually the *Bypass* register is a short-circuit which allows bypassing of board-level devices, in the boundary-scan chain, which do not require a specific test. The logical location of the *Bypass* register in the boundary-scan chain is shown in Figure 13-4. Use of the *Bypass* register speeds up access to boundary-scan registers in those ICs that remain active in the board-level test data path.

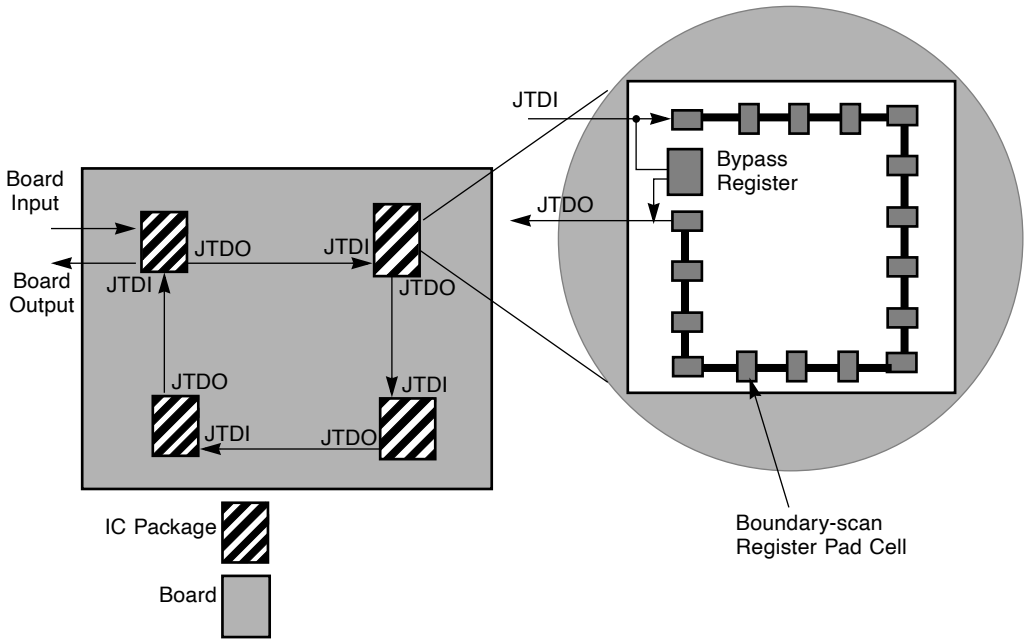


Figure 13-4 Bypass Register Operation

13.3.3 Boundary-Scan Register

The *Boundary-scan* register retains states all of the input and output pins of the V_R4300 processor, except for some clock and phase lock loop signals. The external pins of the V_R4300 can be configured to drive any arbitrary pattern depending on scanning contents into the *Boundary-scan* register from the Shift-DR state. Incoming data to the processor is examined by shifting while in the Capture-DR state with the *Boundary-scan* register enabled.

The *Boundary-scan* register is a single bus comprised of 58-bit shift registers, each bit of which is connected to all input and output pads one by one on the V_R4300 processor. Figure 13-5 shows the most-significant bit of the *Boundary-scan* register; this one bit controls the output enable signals on the various bidirectional buses.

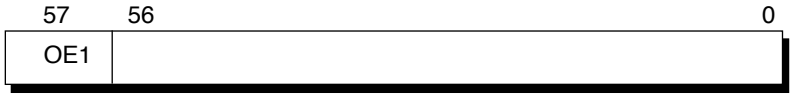


Figure 13-5 Output Enable Bit of Boundary-Scan Register

OE1 (jSysADEn) is the JTAG output enable bit for all outputs of the processor. Output is enabled when this bit is set to 1 (default state).

The remaining 57 bits correspond to 57 signal pads. Outputs are enabled when this bit is set to 1.

Table 13-2 lists the scan order of these scan bits.

13.3.4 Test Access Port (TAP)

The Test Access Port (TAP) consists of the four signal pins: **JTDI**, **JTDO**, **JTMS**, and **JTCK**. These pins control the test to be executed.

As Figure 13-6 shows, data is serially scanned into one of the three registers (*Instruction* register, *Bypass* register, or the *Boundary-scan* register) from the **JTDI** pin, or it is scanned from one of these three registers onto the **JTDO** pin.

Data is input to the **JTDI** pin from the least-significant bit (LSB) of the selected register, whereas the most-significant bit (MSB) of the selected register appears on the **JTDO** pin output.

The **JTMS** signal controls the state transitions of the main TAP controller state machine.

The **JTCK** signal is a dedicated test clock that allows serial JTAG data to be shifted synchronously, independent of any chip-specific or system clock.

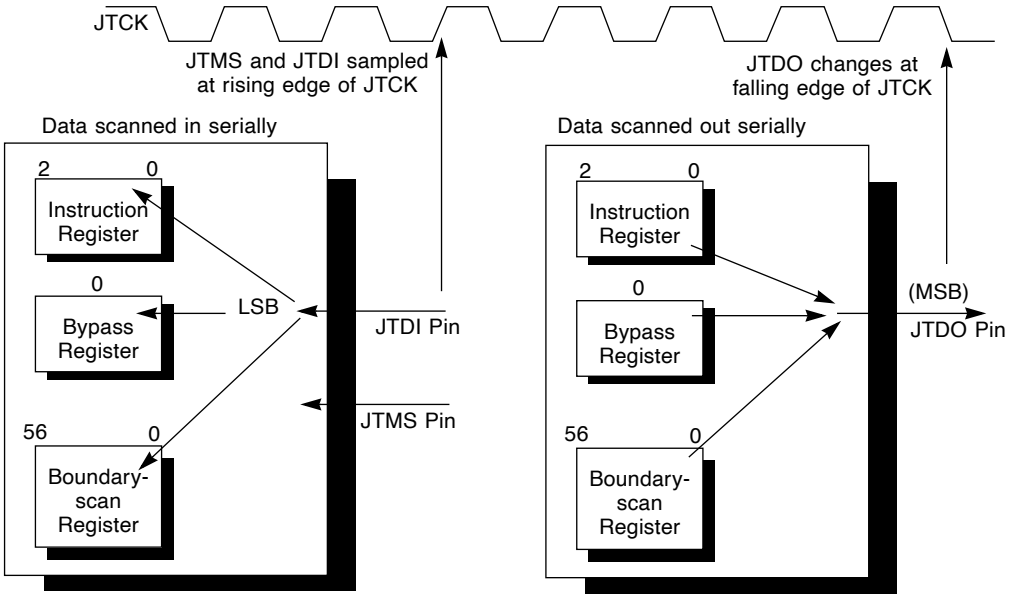


Figure 13-6 JTAG Test Access Port

The **JTDI** and **JTMS** signals are sampled in synchronization with the rising edge of the **JTCK** signal. State on the **JTDO** signal changes in synchronization with the falling edge of the **JTCK** signal.

13.3.5 TAP Controller

The processor incorporates a 16-state TAP controller conforming to the IEEE JTAG standard.

13.3.6 Controller Reset

The TAP controller can be reset by one of the following:

- assert the **$\overline{\text{ColdReset}}$** signal
- keep the **JTMS** signal asserted and input five rising edges of **JTCK** signal

In either case, keeping **JTMS** signal asserted maintains the Reset state.

13.3.7 Controller States

The TAP controller has four states: Reset, Capture, Shift, and Update. They can be further classified as Shift-R state or Capture-DR state, depending on whether the type of signal is instruction or data.

Reset State (TAP Controller)

The value 0x7 is loaded into the parallel output latch, selecting the *Bypass* register as default. The most-significant bits of the *Boundary-scan* register is cleared to 0, disabling the outputs.

Capture IR State

The value 0x4 is loaded into the shift register stage.

Capture DR (Boundary Scan) State

The data currently on the processor input and I/O pins is latched into the *Boundary-scan* register. In this state, the *Boundary-scan* register bits corresponding to output pins are undefined and cannot be checked during the scan out processing.

Shift IR State

Data is loaded serially into the shift register stage of the *Instruction* register from the **JTDI** input pin, and the MSB of the *Instruction* register's shift register stage is shifted out to the **JTDO** pin.

Shift DR (Boundary Scan) State

Data is serially shifted into the *Boundary-scan* register from the **JTDI** pin, and the contents of the *Boundary-scan* register are serially shifted onto the **JTDO** pin.

Update IR State

The current data in the shift register stage is loaded into the parallel output latch.

Update DR (Boundary Scan) State

Data in the *Boundary-scan* register is latched into the register parallel output latch. Bits corresponding to output pins, and those I/O pins whose outputs are enabled by the MSB (OE1) of the *Boundary-scan* register, are loaded onto the processor pins.

Table 13-2 shows the boundary scan order of the processor signals.

Table 13-2 JTAG Scan Order

No.	Signal Name	No.	Signal Name	No.	Signal Name	No.	Signal Name
1	SysAD4	16	SysAD26	31	SysAD23	46	SysAD14
2	SysAD3	17	$\overline{\text{PMaster}}$	32	$\overline{\text{Int3}}$	47	SysAD13
3	SysAD2	18	SysAD25	33	SysAD22	48	SysAD12
4	SysAD1	19	$\overline{\text{EReq}}$	34	SysAD21	49	SysAD11
5	SysAD0	20	SysCmd0	35	SysAD20	50	SysAD10
6	$\overline{\text{PReq}}$	21	SysCmd1	36	RFU (Input: always 1)	51	$\overline{\text{Int0}}$
7	SysAD31	22	$\overline{\text{Reset}}$	37	RFU (Input: always 1)	52	SysAD9
8	$\overline{\text{PValid}}$	23	$\overline{\text{EValid}}$	38	TClock	53	SysAD8
9	SysAD30	24	SysCmd2	39	SyncOut	54	SysAD7
10	$\overline{\text{EOK}}$	25	SysCmd3	40	SysAD19	55	SysAD6
11	SysAD29	26	$\overline{\text{ColdReset}}$	41	SysAD18	56	SysAD5
12	SysAD28	27	SysCmd4	42	SysAD17	57	$\overline{\text{Int1}}$
13	SysAD27	28	DivMode1	43	$\overline{\text{Int4}}$	58	jSysADEn
14	$\overline{\text{Int2}}$	29	SysAD24	44	SysAD16		
15	$\overline{\text{NMI}}$	30	DivMode0	45	SysAD15		

13.4 Notes on Implementation

This section describes points to be noted of JTAG boundary-scan operation that are specific to the processor.

- The **MasterClock**, **SyncIn**, and **SyncOut** signal pads do not support JTAG.
- The update function occurs on the falling edge of JTCK signal after the TAP controller enters the Update-DR state. This conforms to the IEEE standard.

  The V_R4200 generates the update function at the next rising edge. In other words, it is 1/2JTCK cycle late as compared with the V_R4300.

Interrupts

14

Four types of interrupt are available on the V_R4300. These are:

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

These are described in this chapter.

14.1 Non-Maskable Interrupt

The non-maskable interrupt request is accepted by asserting the **$\overline{\text{NMI}}$** signal (low), forcing the processor to branch to the Reset Exception vector. **$\overline{\text{NMI}}$** signal is latched into an internal register in synchronization with the rising edge of **SClock** signal, as shown in Figure 14-1. The **$\overline{\text{NMI}}$** signal is edge-triggered, and NMI request is acknowledged when the **$\overline{\text{NMI}}$** signal is kept low for more than one cycle. This signal must be high after an exception occurs. An NMI request can also be set by an external write request through the **SysAD(31:0)** bus. On the data cycle, **SysAD6** acts as the NMI request bit (1:requested) and **SysAD22** acts as the write enable bit (1:enable) for **SysAD6**.

NMI only takes effect when the processor pipeline is running. Thus NMI can be used to recover the processor from a software hang up (for example, in an infinite loop) but cannot be used to recover the processor from a hardware hang up (for example, no read response from an external device). NMI cannot cause drive contention on the **SysAD(31:0)** bus and no reset of external agents is required.

This interrupt cannot be masked.

Figure 14-1 shows the internal processing of the **$\overline{\text{NMI}}$** signal. The low-level signal input to **$\overline{\text{NMI}}$** pin is latched into an internal register in synchronization with the rising edge of **SClock**. Bit 6 of the *internal* register is then ORed with the inverted value of latched **$\overline{\text{NMI}}$** signal to transfer internally as the non-maskable interrupt request.

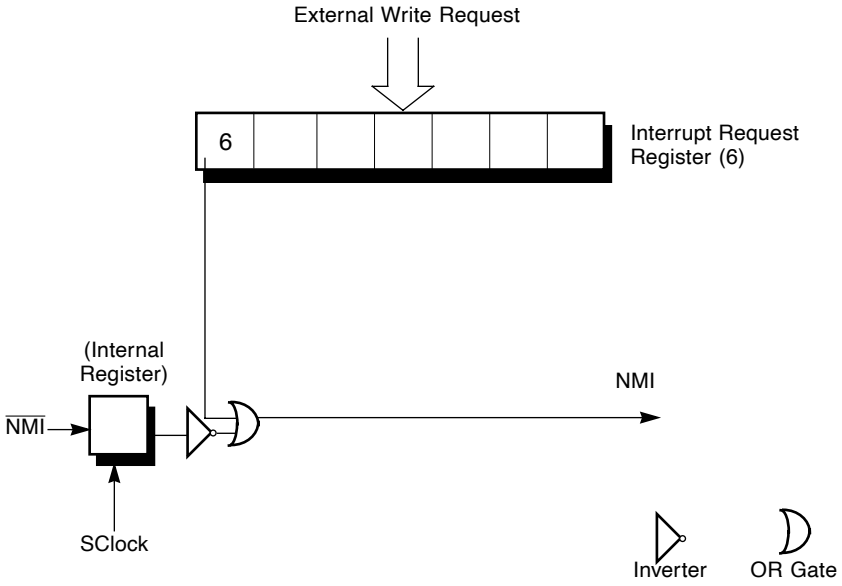


Figure 14-1 $\overline{NMI}$ Signal

14.2 External Normal Interrupts

These interrupt requests are accepted by asserting **$\overline{\text{Int(4:0)}}$** signal (low). **Int(4:0)** signals are level-triggered, and these signals must be kept low until an external interrupt exception is generated. After an external interrupt exception occurs, **$\overline{\text{Int(4:0)}}$** signal must be high before the processor returns to its normal routine, or before multiple interrupts are enabled. This interrupt request can be set by an external write request through the **SysAD(31:0)** bus. During the data cycle, **SysAD(4:0)** acts as the external interrupt request bit (1:requested) and **SysAD(20:16)** acts as the write enable bit (1:enable) for **SysAD(4:0)**.

After an external interrupt exception occurs, an external write request must be issued to clear the corresponding bit of the interrupt register to 0 before the processor returns to its normal routine, or before multiple interrupts are enabled.

These interrupt requests can be masked with the *IM(6:2)*, *IE*, *EXL*, and *ERL* fields of the *Status* register.

14.3 Software Interrupts

These interrupt requests are accepted by setting bit 1 or 0 of the interrupt pending, *IP*, field in the *Cause* register to 1. These bits can be written by software, but there is no hardware mechanism to set or clear these bits.

After a software interrupt exception occurs, the corresponding bit of the *IP* field in the *Cause* register must be cleared to 0 before the processor returns to its normal routine, or before multiple interrupts are enabled.

These interrupt requests are maskable with the *IM(1:0)*, *IE*, *EXL*, and *ERL* fields of the *Status* register.

14.4 Timer Interrupt

These interrupt requests use bit 7 of the IP (interrupt pending) field in the *Cause* register. The timer interrupt is automatically set and accepted whenever the value of the *Count* register equals the value of the *Compare* register.

To clear this interrupt request, either clear the *IP7* bit of the *Cause* register, or change the contents of the *Compare* register.

This interrupt request is maskable through the *IM7* bit and *IE*, *EXL* and *ERL* fields of the *Status* register.

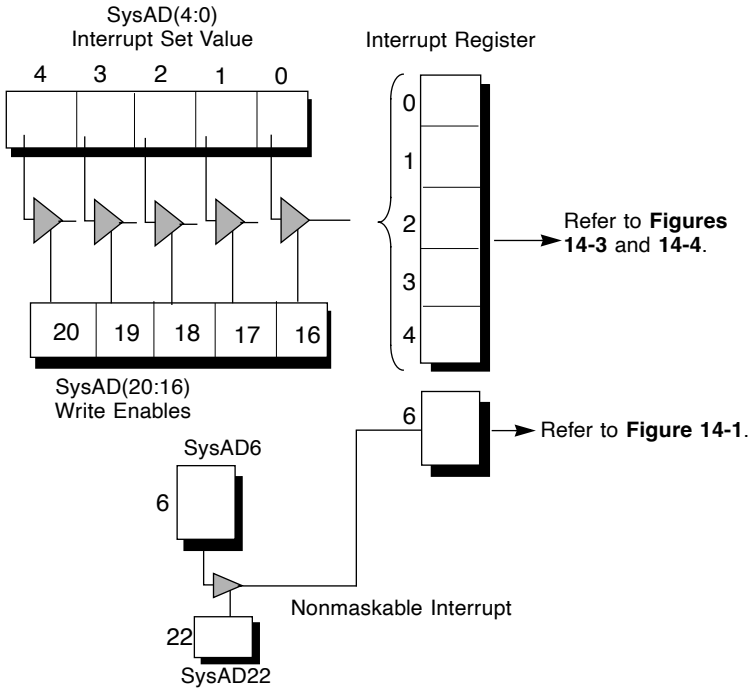
14.5 Generation of Interrupt Request Signal

When an external agent issues an external write request, it is written to the *Interrupt* register. This register can be used in an external write cycle, but not in an external read cycle.

When data is written to the *Interrupt* register, the processor ignores the address issued by the external agent.

This register cannot be read or written by software unlike the *CP0* register.

In the data cycle, bits **SysAD20** through **SysAD16** are used as individual write enable bits corresponding to the 5 bits of the *Interrupt* register. The values **SysAD4** through **SysAD0** are written to the bits of the *Interrupt* register. Therefore, the bits 0 through 4 of the *Interrupt* register can be set or cleared by issuing an external write request only once. Figure 14-2 illustrates this along with the NMI described earlier.



Bit	Meaning	Setting
SysAD(4:0)	External interrupt request Int (4:0)	1 : requested 0 : no request (for each bit)
SysAD(20:16)	Write enable bits for SysAD(4:0)	1 : enable 0 : disable (for each bit)
SysAD6	NMI	1 : requested 0 : no request
SysAD22	Write enable bit for SysAD6	1 : enable 0 : disable

Figure 14-2 Interrupt Register Bits and Enables Bits

14.5.1 Detection of Hardware Interrupts

Figure 14-3 shows how the V_R4300 hardware interrupt causes are detected through the *Cause* register.

- The timer interrupt signal, *IP7*, is directly detected as bit 15 of the *Cause* register.
- The other hardware interrupt signals are directly detected since bits 4:0 of the *Interrupt* register are ORed one by one with each signal of the interrupt pins **$\overline{\text{Int(4:0)}}$** and the result is input to bits 14:10 of the *Cause* register.

IP(1:0) of the *Cause* register are related to software interrupts. (Refer to **Chapter 6 Exception Processing** for detail.) There is no hardware mechanism for setting or clearing the software interrupts.

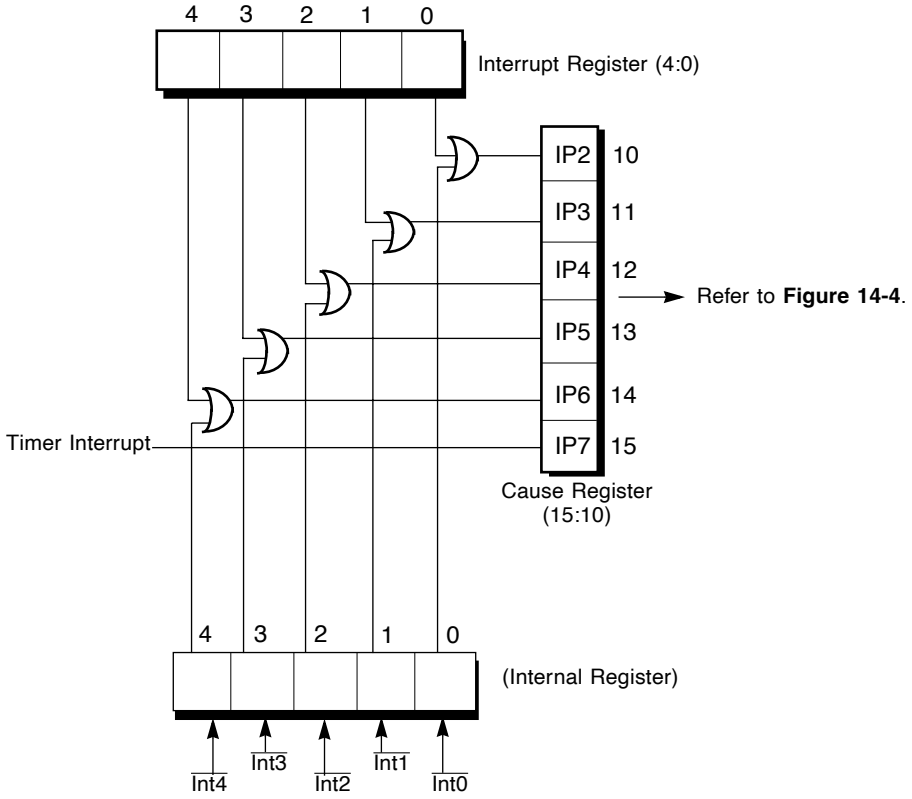
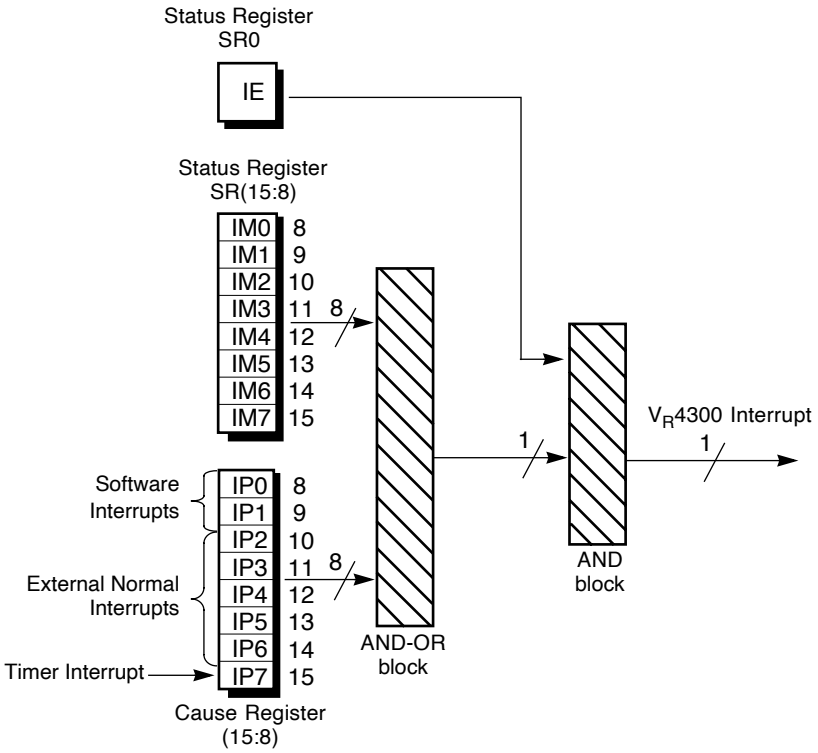


Figure 14-3 Hardware Interrupt Request Signals

14.5.2 Masking of Interrupt Request Signals

Figure 14-4 shows the masking of the V_R4300 interrupt request signals.

- *Cause* register bits 15:8 (IP7-IP0) are AND-ORed with *Status* register interrupt mask bits 15:8 (IM7-IM0) to mask individual interrupt signals.
- *Status* register bit 0 is a global Interrupt Enable (IE) bit. The output of this bit is ANDed with the output of the AND-OR logic block to produce the V_R4300 interrupt signal as shown in Figure 14-4. The *EXL* bit in the *Status* register also enables these interrupts.



Bit	Meaning	Setting
IE	Enable all interrupts	1 : enable 0 : disable
IM(7:0)	Mask interrupts	1 : enable 0 : disable (for each bit)
IP(7:0)	Interrupt requests	1 : request pending 0 : no pending (for each bit)

Figure 14-4 Masking of Interrupt Requests

[MEMO]

Power Management

15

One of the objectives of the design of the V_R4300 processor is to minimize power consumption in order to make the processor suitable for use in battery operated systems, as well as in environments where low power consumption and heat dissipation are desirable.

To accomplish this, the V_R4300 has power management features which bring a dynamic reduction of power consumption, described in this chapter.

15.1 Features

★ The V_R4300 has three processor-level operation modes: normal, low power (100 MHz model of the V_R4300 and the V_R4305 only), and power off.

These modes allow processor power consumption to be managed by system logic.

Generally a notebook system has many different levels of power management. It is the responsibility of system logic to switch the processor between the three available modes in order to reflect the power management state of the system.

15.1.1 Normal Power Mode

The normal pipeline clock (**PClock**) is generated based on the input clock (**MasterClock**). The ratio of the frequency of **PClock** to that of **MasterClock** is set by the **DivMode(1:0)*** pins. For the details of setting, refer to **2.2.2 Clock/ Control Interface Signals**.

The frequency of the system interface clock (**SClock**) is the same as that of **MasterClock**.

The processor operates in the normal mode as default condition. The processor enters the default status after reset.

* In V_R4300 and V_R4305. In V_R4310, DivMode(2:0).

15.1.2 Low Power Mode

★ The low power mode is supported only in the 100 MHz model of the V_R4300 and the V_R4305.

The processor operates in the low power mode when the RP bit of the *Status* register is set. In this mode, the processor once stalls the pipeline, entering the quiescent status. In this status, the store buffer becomes empty, and all cache misses are processed.

The frequency of **PClock** drops to the 1/4 of the normal level. The speeds of **SClock** and **TClock** also drop to the 1/4 of the normal level.

★ **Example** When DivMode (1:0) = 10 in 100 MHz model of the V_R4300

	MasterClock	PClock	SClock, TClock
Normal mode	50 MHz	100 MHz	50 MHz
Low power mode	50 MHz	25 MHz	12.5 MHz

The low power mode can reduce the power consumption of the processor to about 25% of the normal level. When setting or clearing the *RP* bit, guarantee the normal operation of the system by software.

Also keep in mind the following points.

1. The functions of circuits such as the DRAM refresh counter change if the operating frequency changes. Consequently, first write new values to the registers of the external agent that are directly affected by changes in the frequency.
2. Make sure that the operation of the system interface is inactive. For example, execute an instruction that reads the non-cache area, and vacate the write/ buffer after execution of the instruction. After that, the *RP* bit can be set or cleared.
3. Make sure that eight instructions before and after the MTC0 instruction that sets or clears the *RP* bit do not cause an exception such as cache miss or TLB miss exception.

15.1.3 Power Off Mode

In the power off mode, power supply to the processor is entirely cut off and operation of the processor stops completely.

Before entering power off mode, the state of the processor is written to non-volatile memory. When the processor returns to the normal mode, all registers are restored to their previous state.

In order to support power off mode, all internal state information necessary for restoring the processor from the state of power off is read and write accessible. Prior to power off, this information must be saved into non-volatile memory connected externally.

It is the system's responsibility to power off the chip when the system is in idle state. At this time the Load Link **LL** bit is not required to be saved since it is automatically cleared by the cache start-up.

Cache content is not retained, and therefore the cache should be invalidated during the power-on routine and written back to the memory during the power-off routine. The V_R4300 chip supports the CACHE instructions and TLB operation instructions which invalidate all caches and TLB contents.

[MEMO]

CPU Instruction Set Details

16

This chapter provides a detailed description of the function of each V_R4300 CPU instruction in both 32- and 64-bit modes. The instructions are listed in alphabetical order.

For details of the FPU instruction set, refer to **Chapter 17 FPU Instruction Set Details**.

16.1 Instruction Notation Conventions

In this chapter, all variable subfields in an instruction format (such as *rs, rt, immediate*, etc.) are shown in lowercase characters. Instruction names (such as *ADD*, *SUB*, *etc*.) are shown in upper case characters. For the sake of clarity, sometimes an alias is used for a subfield in the specific instructions. For example, we use *rs = base* for load and store instructions. Such an alias is always lower case characters, since it also refers to a subfield.

The actual encoding for all the mnemonics are located in **16.7 CPU Instruction Opcode Bit Encoding**, and the bit encoding also accompanies each instruction description.

In the instruction descriptions, the *Operation* section describes the operation performed by each instruction using a high-level language notation. The V_R4300 can operate in either 32- or 64-bit mode. Differences in operations in each mode are shown in operation section. Special symbols used in the notation are described in Table 16-1.

Table 16-1 CPU Instruction Operation Notations

Symbol	Meaning
←	Substitution
||	Bit string concatenation.
x^y	Repetition of bit string *x* with a *y*-bit string. *x* is always a single-bit value.
$x_{y...z}$	Selection of bits *y* through *z* for 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 remainder.
/	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 Purpose 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 purpose register *x*.
CCR[*z,x*]	Coprocessor unit *z*, control register *x*.
COC[*z*]	Coprocessor unit *z*, condition signal.
BigEndianMem	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 modes.
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 is set to 1 only when the *RE* bit is set in User mode.
BigEndianCPU	The endianness for load and store instructions (0 → Little, 1 → Big). In User mode, this endianness is reversed by setting *RE* bit. Thus, BigEndianCPU is calculated as BigEndianMem XOR ReverseEndian.
LLbit	Bit showing synchronized state of instructions. Set by *LL* instruction, cleared by *ERET* instruction and read by *SC* instruction.
T+*i*:	Indicates the time steps between operations. Each statement within a time step are defined to be executed in sequential order (instruction execution order may be changed by conditional branch and loop). Operations which are marked *T+i:* are executed at instruction cycle *i* from the start of execution of the instruction. Thus, an instruction which starts at time *j* executes operations marked T+*i:* at time of *i + j*th cycle. The order is not defined for instructions executed at the same time or operations.

Instruction Notation Examples

The following are examples of the instruction notations:

Example #1:

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

Sixteen zero bits are concatenated with a low-order immediate value (normally 16 bits), and the 32-bit string is substituted to CPU General Purpose Register *rt*.

Example #2:

$$(immediate_{15})^{16} \text{ || } immediate_{15...0}$$

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

16.2 Load and Store Instructions

In the V_R4300, the instruction immediately following a load instruction may use the loaded register contents. In such cases, the hardware *interlocks by* **1PCycle** only, so scheduling load delay slots is desirable to improve performance, although not required as a functional code.

Two special instructions are provided in the V_R4300 implementation of the MIPS ISA, Load Link and Conditional Store Instructions. These instructions are used in carefully coded sequences to execute one of several synchronization primitives, including test-and-set, bit-level locks, semaphores, and sequencers/event counter, etc. This synchronization is essential in multi-processor systems. This functionality is included in the V_R4300 primarily for reasons to keep compatibility with the V_R4000 and V_R4200.

In the load and store instruction descriptions, the functions listed below are used to simplify the handling of virtual addresses and physical memory.

Table 16-2 Load and Store Instruction Common Functions

Function	Meaning
AddressTranslation	Uses TLB to search a physical address from a virtual address. If TLB does not have the requested contents of conversion, this function fails, and TLB non-coincidence exception occurs.
LoadMemory	Searches the cache and main memory to search for the contents of the specified data length stored in a specified physical address. If the specified data length is less than a word, the contents of a data position taking the endian mode and reverse endian mode of the processor into consideration are loaded. The low-order 3 bits and access type field of the address determine the data position in a data word. The data is loaded to the cache if the cache is enabled.
StoreMemory	Searches the cache, write buffer, and main memory to store the contents of a specified data length to a specified physical address. If the specified data length is less than a word, the contents of a data position taking the endian mode and reverse endian mode of the processor into consideration are stored. The low-order 3 bits and access type field of the address determine the data position in a data word.

The *Access Type* field indicates the size of the data to be loaded or stored. Regardless of access type or byte order (endianness), the address specifies the byte which has the smallest byte address in the field accessed. For a big-endian system, this is the leftmost byte and contains the sign for a 2's complement value; for a little-endian system, this is the rightmost byte.

Table 16-3 Access Type Specifications for Load/Store Instructions

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

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

16.3 Jump and Branch Instructions

All jump and branch instructions have structural delay of exactly one instruction. That is, the instruction immediately following a jump or branch instruction (that is, occupying the delay slot) is executed while the target instruction is being fetched from the cache. A jump or branch instruction cannot be used in a delay slot; however, if they are used, the error is not detected and the results of such an operation are undefined.

If an exception or interrupt prevents the completion of the instruction during it is in a delay slot, the hardware sets a virtual address to the *EPC* register at the point of the jump or branch instruction that precedes it. When processing exceptions or interrupts is completed and the program is restored, both the jump or branch instruction and the instruction in the delay slot are reexecuted.

Because jump and branch instructions may be reexecuted after exception or interrupt processing, register *31* (the register in which the link address is stored) should not be used as a source register in jump and link/branch and link instructions.

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

16.4 Coprocessor Instructions

Coprocessors are alternate execution units, which have register files separate from the CPU. The MIPS architecture provides four coprocessor units and these coprocessors have two register spaces, each space containing thirty-two 32-bit registers.

- The first space, coprocessor general purpose registers, is directly loaded from and stored into the main memory, and their contents can be transferred between the coprocessor and processor.
- The second space, coprocessor control registers, can only have their contents transferred between the coprocessor and the processor. Coprocessor instructions may alter registers in either space.

16.5 System Control Coprocessor (CP0) Instructions

There are some 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 exchange control codes 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 coprocessor transfer instructions are the only valid way for writing to and reading from the CP0 registers.

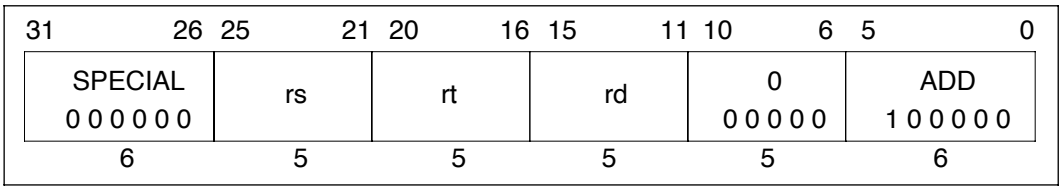
Some CP0 instructions are defined to directly read, write, and probe TLB entries and to change the operating modes in preparation for restoring to User mode or interrupt-enabled states.

16.6 CPU Instructions

This section describes in detail each function of CPU instructions in 32- or 64-bit mode.

Possible exceptions, which may occur are caused by instruction execution, and are explained at the end of the description for each instruction. Refer to **Chapter 6 Exception Processing** for details of exceptions and their processing.

ADD Add ADD



Format:

ADD rd, rs, rt

Description:

The contents of general purpose register *rs* and the contents of general purpose register *rt* are added to store the result in general purpose register *rd*. In 64-bit mode, the operands must be sign-extended, 32-bit values.

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

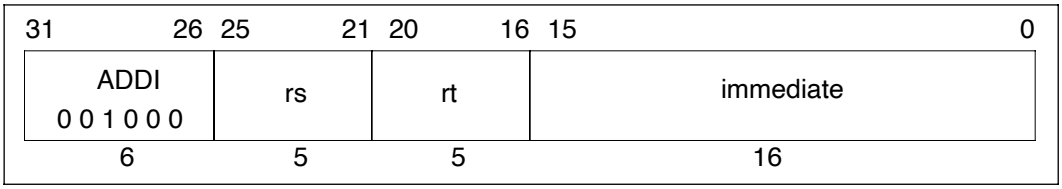
Operation:

```
32    T:   GPR[rd] ←GPR[rs] + GPR[rt]

64    T:   temp ← GPR[rs] + GPR[rt]
           GPR[rd] ← (temp31)32 || temp31...0
```

Exceptions:

Integer overflow exception

ADDI Add Immediate ADDI



Format:

ADDI rt, rs, immediate

Description:

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

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

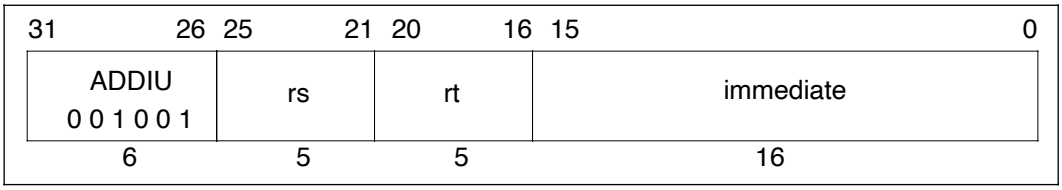
Operation:

```
32    T:    GPR [rt] ← GPR[rs] +(immediate15)16 || immediate15...0

64    T:    temp ← GPR[rs] + (immediate15)48 || immediate15...0
            GPR[rt] ← (temp31)32 || temp31...0
```

Exceptions:

Integer overflow exception

ADDIU Add Immediate Unsigned ADDIU



Format:

ADDIU rt, rs, immediate

Description:

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

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

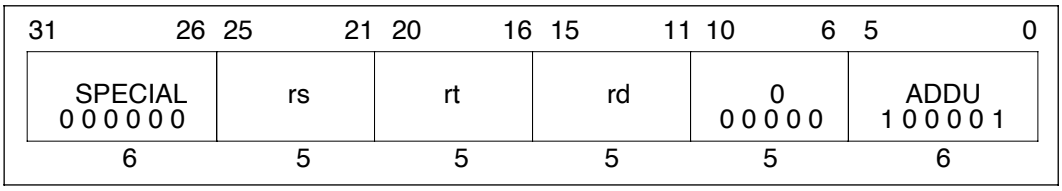
Operation:

```
32    T:    GPR [rt] ← GPR[rs] + (immediate15)16 || immediate15...0

64    T:    temp ← GPR[rs] + (immediate15)48 || immediate15...0
            GPR[rt] ← (temp31)32 || temp31...0
```

Exceptions:

None

ADDU Add Unsigned ADDU



Format:

ADDU rd, rs, rt

Description:

The contents of general purpose register *rs* and the contents of general purpose register *rt* are added to store the result in general purpose register *rd*. No integer overflow exception occurs under any circumstance. In 64-bit mode, the operands must be sign-extended, 32-bit values.

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

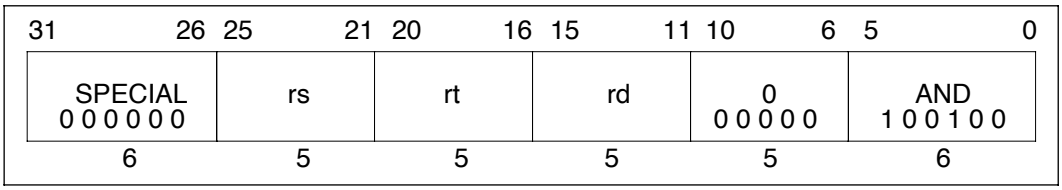
Operation:

```
32    T:    GPR[rd] ←GPR[rs] + GPR[rt]

64    T:    temp ← GPR[rs] + GPR[rt]
            GPR[rd] ← (temp31)32 || temp31...0
```

Exceptions:

None

AND And AND



Format:

AND rd, rs, rt

Description:

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

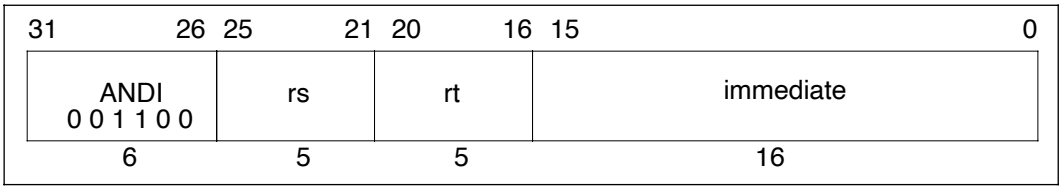
Operation:

```
32    T:    GPR[rd] ← GPR[rs] and GPR[rt]

64    T:    GPR[rd] ← GPR[rs] and GPR[rt]
```

Exceptions:

None

ANDI And Immediate ANDI



Format:

ANDI rt, rs, immediate

Description:

The 16-bit *immediate* is zero-extended and combined with the contents of general purpose register *rs* in a bit-wise logical AND operation. The result is stored in general purpose register *rt*.

Operation:

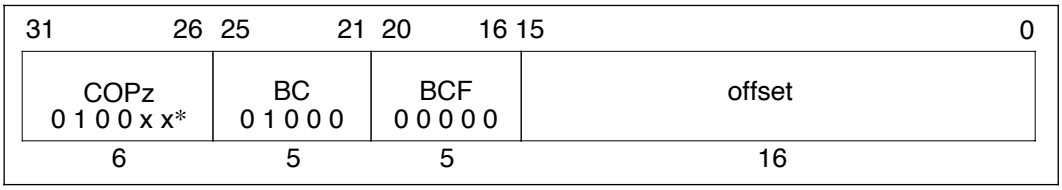
32 T: $GPR[rt] \leftarrow 0^{16} \,||\, (immediate \text{ and } GPR[rs]_{15...0})$

64 T: $GPR[rt] \leftarrow 0^{48} \,||\, (immediate \text{ and } GPR[rs]_{15...0})$

Exceptions:

None

BCzF Branch On Coprocessor z False BCzF



Format:

BCzF offset

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted two bits left and sign-extended. If CPz's condition signal (CpCond), as sampled during the previous instruction execution, is false, then the program branches to the branch address with a delay of one instruction.

Because the condition signal is sampled during the previous instruction execution, there must be at least one instruction between this instruction and a coprocessor instruction that changes the condition signal.

Operation:

```
32    T–1: condition ← not COC[z]
      T:   target ← (offset15)^14 || offset || 0^2
      T+1: if condition then
                        PC ← PC + target
           endif

64    T–1: condition ← not COC[z]
      T:   target ← (offset15)^46 || offset || 0^2
      T+1: if condition then
                        PC ← PC + target
           endif
```

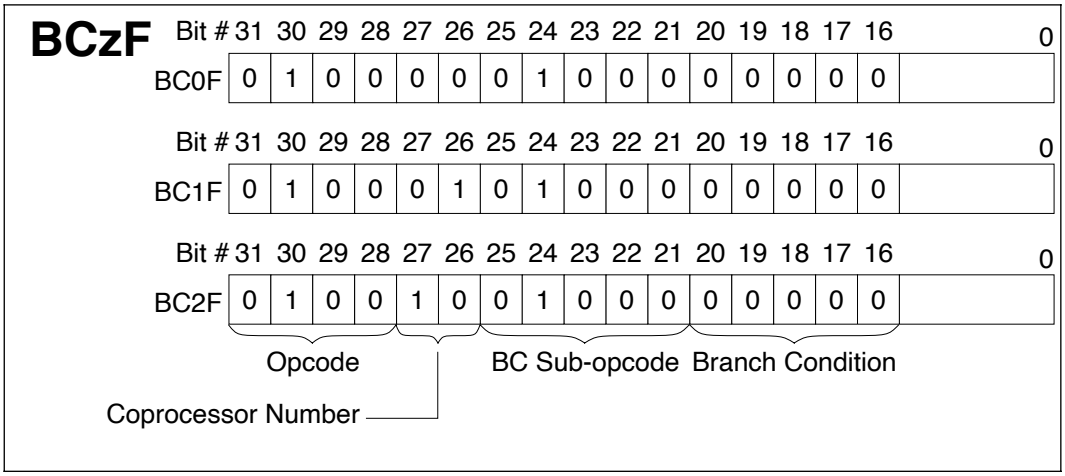
* Refer to the table **Opcode Bit Encoding** on the next page, or **16.7 CPU Instruction Opcode Bit Encoding**.

BCzF Branch On Coprocessor z False (continued) BCzF

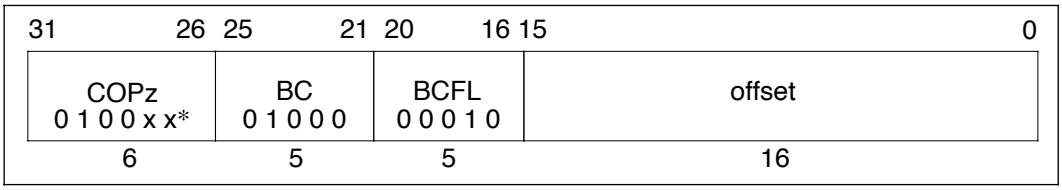
Exceptions:

Coprocessor unusable exception

Opcode Bit Encoding:



BCzFL — Branch On Coprocessor z False Likely — BCzFL



Format:

BCzFL offset

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted two bits left and sign-extended. If the CPz's condition signal (CpCond), as sampled during the previous instruction execution, is false, the program branches to the branch address with a delay of one instruction.

If it does not branch, the instruction in the branch delay slot is discarded.

Because the condition signal is sampled during the previous instruction execution, there must be at least one instruction between this instruction and a coprocessor instruction that changes the condition signal.

* Refer to the table **Opcode Bit Encoding** on the next page, or **16.7 CPU Instruction Opcode Bit Encoding**.

BCzFL — Branch On Coprocessor z False Likely (continued) — BCzFL

Operation:

```
32    T–1: condition ← not  COC[z]
      T:   target ← (offset15)^14 || offset || 0^2
      T+1: if condition then
                        PC ← PC + target
           else
                        NullifyCurrentInstruction
           endif
64    T–1: condition ← not  COC[z]
      T:   target ← (offset15)^46 || offset || 0^2
      T+1: if condition then
                        PC ← PC + target
           else
                        NullifyCurrentInstruction
           endif
```

Exceptions:

Coprocessor unusable exception

Opcode Bit Encoding:

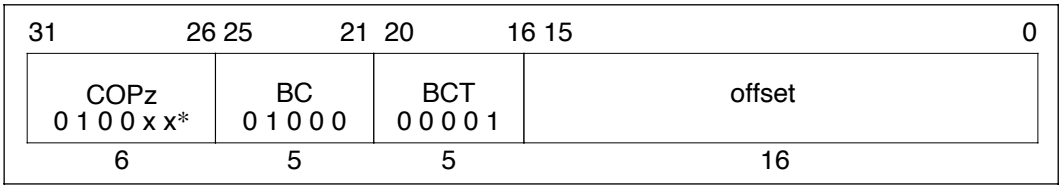
BCzFL

Bit #	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	0
BC0FL	0	1	0	0	0	0	0	1	0	0	0	0	0	0	1	0	
BC1FL	0	1	0	0	0	1	0	1	0	0	0	0	0	0	1	0	
BC2FL	0	1	0	0	1	0	0	1	0	0	0	0	0	0	1	0	

Opcode — BC Sub-opcode — Branch Condition

Coprocessor Number

BCzT Branch On Coprocessor z True BCzT



Format:

BCzT offset

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted two bits left and sign-extended. If the CPz's condition signal (CpCond) sampled during the previous instruction execution is true, then the program branches to the branch address with a delay of one instruction.

Because the condition signal is sampled during the previous instruction execution, there must be at least one instruction between this instruction and a coprocessor instruction that changes the condition signal.

Operation:

```
32    T–1: condition ← COC[z]
      T:   target ← (offset15)^14 || offset || 0^2
      T+1: if condition then
                PC ← PC + target
           endif
64    T–1: condition ← COC[z]
      T:   target ← (offset15)^46 || offset || 0^2
      T+1: if condition then
                PC ← PC + target
           endif
```

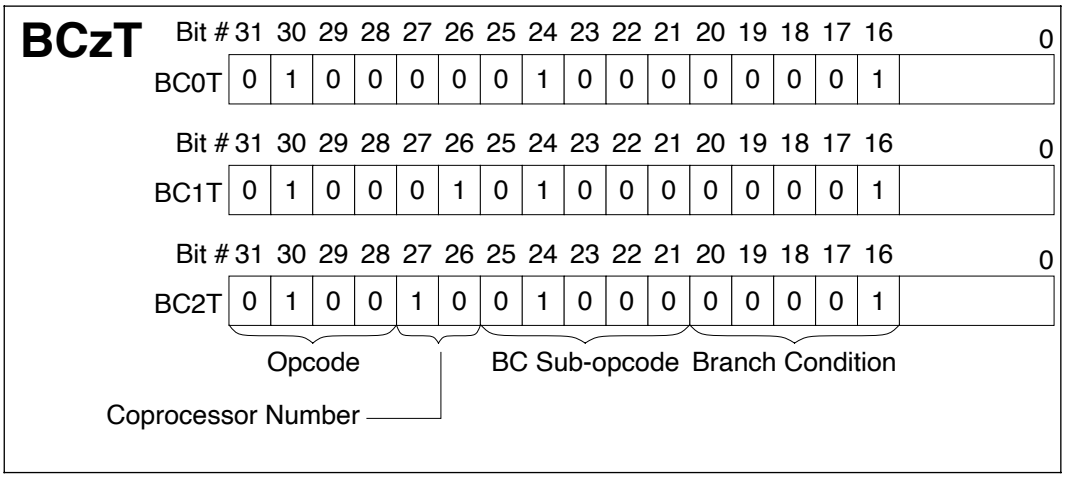
* Refer to the table **Opcode Bit Encoding** on the next page, or **16.7 CPU Instruction Opcode Bit Encoding**.

BCzT Branch On Coprocessor z True (continued) BCzT

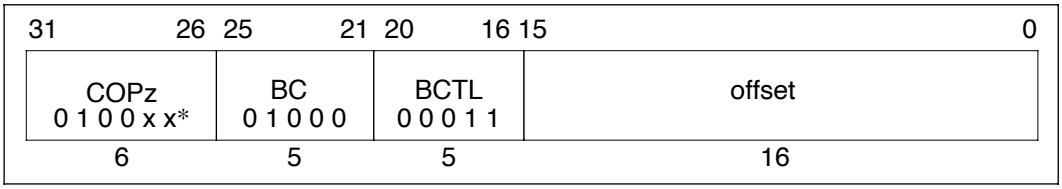
Exceptions:

Coprocessor unusable exception

Opcode Bit Encoding:



BCzTL Branch On Coprocessor z True Likely BCzTL



Format:

BCzTL offset

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted two bits left and sign-extended. If the CPz's condition signal (CpCond), as sampled during the previous instruction execution, is true, the program branches to the branch address with a delay of one instruction.

If it does not branch, the instruction in the branch delay slot is discarded.

Because the condition signal is sampled during the previous instruction execution, there must be at least one instruction between this instruction and a coprocessor instruction that changes the condition signal.

Operation:

```
32    T–1: condition ← COC[z]
      T:   target ← (offset15)^14 || offset || 0^2
      T+1: if condition then
                PC ← PC + target
           else
                NullifyCurrentInstruction
           endif
64    T–1: condition ← COC[z]
      T:   target ← (offset15)^46 || offset || 0^2
      T+1: if condition then
                PC ← PC + target
           else
                NullifyCurrentInstruction
           endif
```

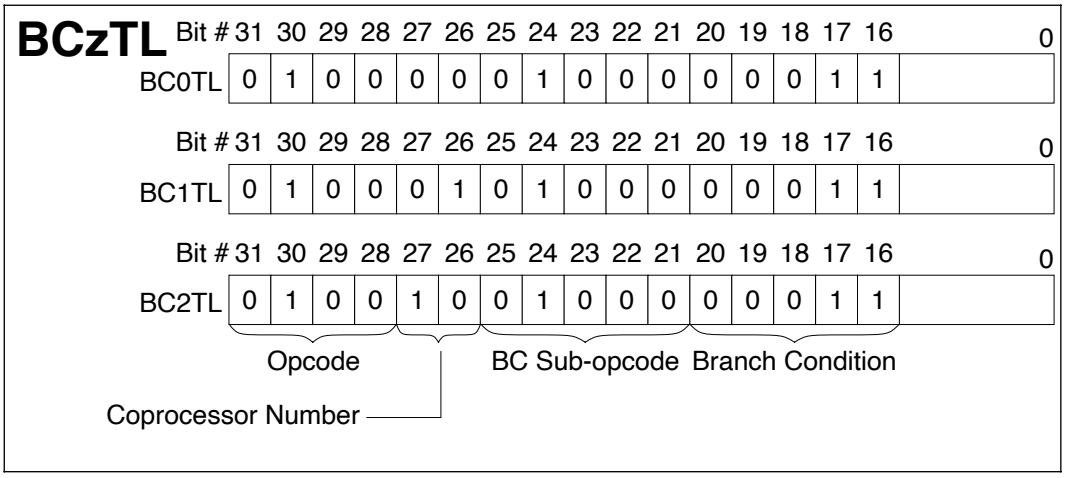
* Refer to the table **Opcode Bit Encoding** on the next page, or **16.7 CPU Instruction Opcode Bit Encoding**.

BCzTL Branch On Coprocessor z True Likely (continued) BCzTL

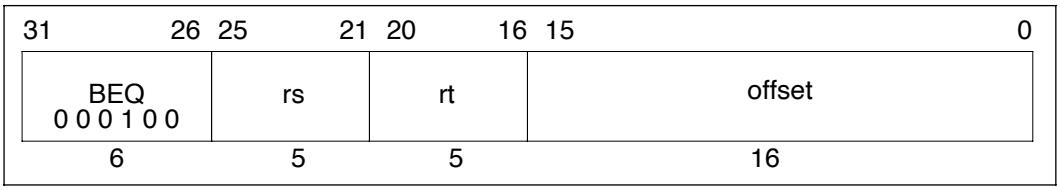
Exceptions:

Coprocessor unusable exception

Opcode Bit Encoding:



BEQ Branch On Equal BEQ



Format:

BEQ rs, rt, offset

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted two bits left and sign-extended. The contents of general purpose register *rs* and the contents of general purpose register *rt* are compared. If the two registers are equal, then the program branches to the branch address with a delay of one instruction.

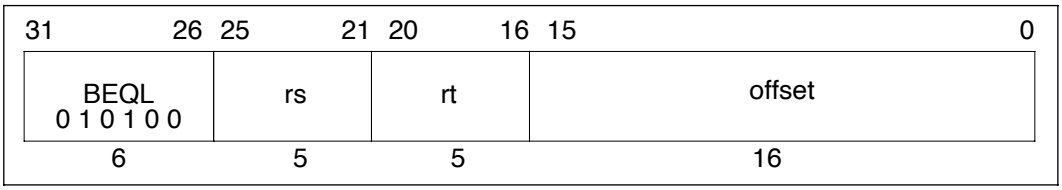
Operation:

```
32    T:    target ← (offset15)^14 || offset || 0^2
            condition ← (GPR[rs] = GPR[rt])
      T+1:  if condition then
                        PC ← PC + target
            endif
64    T:    target ← (offset15)^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



Format:

BEQL rs, rt, offset

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted two bits left and sign-extended. The contents of general purpose register *rs* and the contents of general purpose register *rt* are compared. If the two registers are equal, the program branches to the branch address with a delay of one instruction.

If it does not branch, the instruction in the branch delay slot is discarded.

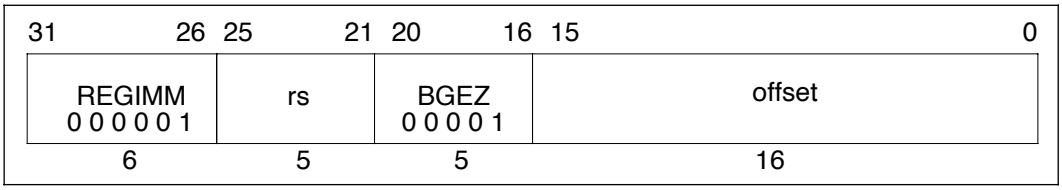
Operation:

```
32    T:    target ← (offset15)^14 || offset || 0^2
            condition ← (GPR[rs] = GPR[rt])
      T+1:  if condition then
                        PC ← PC + target
            else
                        NullifyCurrentInstruction
            endif
64    T:    target ← (offset15)^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



Format:

BGEZ rs, offset

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted two bits left and sign-extended. If the contents of general purpose register *rs* are equal to or larger than 0, then the program branches to the branch address with a delay of one instruction.

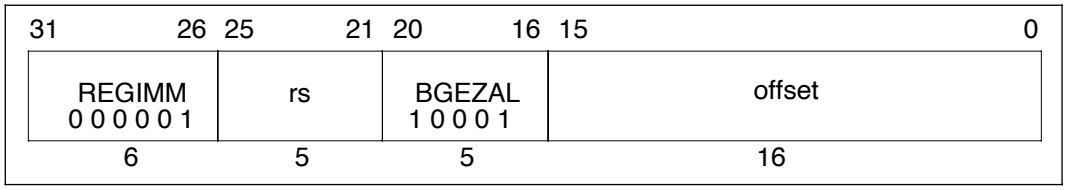
Operation:

```
32   T:    target ← (offset15)^14 || offset || 0^2
           condition ← (GPR[rs]31 = 0)
      T+1: if condition then
                        PC ← PC + target
           endif
64   T:    target ← (offset15)^46 || offset || 0^2
           condition ← (GPR[rs]63 = 0)
      T+1: if condition then
                        PC ← PC + target
           endif
```

Exceptions:

None

BGEZAL Branch On Greater Than Or Equal To Zero And Link BGEZAL



Format:

BGEZAL rs, offset

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset,* shifted two bits left and sign-extended. Unconditionally, the address of the instruction next to the delay slot is stored in the link register, *r31*. If the contents of general purpose register *rs* are equal to or larger than 0, then the program branches to the branch address, with a delay of one instruction.

Generally, general purpose register *r31* should not be specified as general purpose register *rs*, because the contents of *rs* are destroyed by storing link address, and then it may not be reexecutable. An attempt to execute this instruction does not cause exception, however.

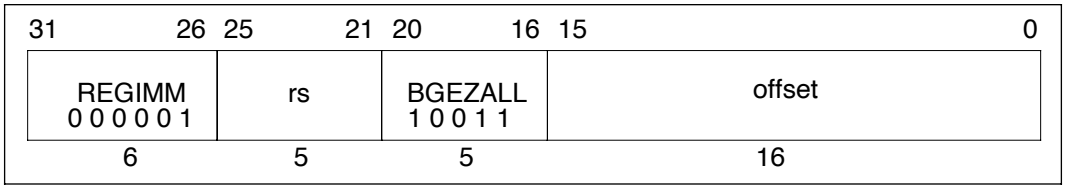
Operation:

```
32    T:    target ← (offset15)^14 || offset || 0^2
            condition ← (GPR[rs]31 = 0)
            GPR[31] ← PC + 8
      T+1:  if condition then
                  PC ← PC + target
            endif
64    T:    target ← (offset15)^46 || offset || 0^2
            condition ← (GPR[rs]63 = 0)
            GPR[31] ← PC + 8
      T+1:  if condition then
                  PC ← PC + target
            endif
```

Exceptions:

None

BGEZALL Branch On Greater Than Or Equal To Zero And Link Likely BGEZALL



Format:

BGEZALL rs, offset

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset,* shifted two bits left and sign-extended. Unconditionally, the address of the instruction next to the delay slot is stored in the link register, *r31*. If the contents of general purpose register *rs* are equal to or larger than 0, then the program branches to the branch address, with a delay of one instruction. When it does not branch, instruction in the delay slot are discarded. Generally, general purpose register *r31* should not be specified as general purpose register *rs*, because the contents of *rs* are destroyed by storing link address, and then it may not be reexecutable. An attempt to execute this instruction does not cause any exception, however.

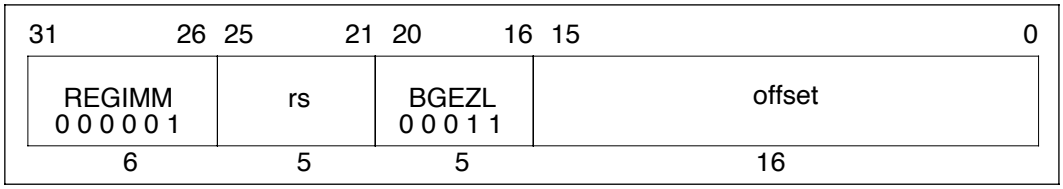
Operation:

```
32    T:    target ← (offset15)^14 || offset || 0^2
            condition ← (GPR[rs]31 = 0)
            GPR[31] ← PC + 8
      T+1:  if condition then
                  PC ← PC + target
            else NullifyCurrentInstruction
            endif
64    T:    target ← (offset15)^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

BGEZL — Branch On Greater Than Or Equal To Zero Likely — BGEZL



Format:

BGEZL rs, offset

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted two bits left and sign-extended. If the contents of general purpose register *rs* are equal to or larger than 0, then the program branches to the branch address, with a delay of one instruction.

If it does not branch, the instruction in the branch delay slot is discarded.

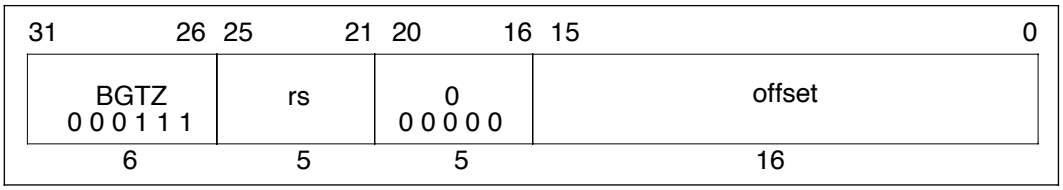
Operation:

```
32    T:    target ← (offset15)^14 || offset || 0^2
            condition ← (GPR[rs]31 = 0)
      T+1:  if condition then
                  PC ← PC + target
            else
                  NullifyCurrentInstruction
            endif
64    T:    target ← (offset15)^46 || offset || 0^2
            condition ← (GPR[rs]63 = 0)
      T+1:  if condition then
                  PC ← PC + target
            else
                  NullifyCurrentInstruction
            endif
```

Exceptions:

None

BGTZ Branch On Greater Than Zero BGTZ



Format:

BGTZ rs, offset

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted two bits left and sign-extended. The contents of general purpose register *rs* are larger than zero, then the program branches to the branch address, with a delay of one instruction.

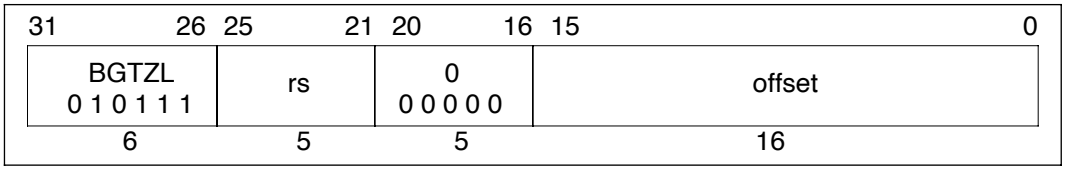
Operation:

```
32    T:    target ← (offset15)^14 || offset || 0^2
            condition ← (GPR[rs]31 = 0) and (GPR[rs] ≠ 0^32)
      T+1:  if condition then
                  PC ← PC + target
            endif
64    T:    target ← (offset15)^46 || offset || 0^2
            condition ← (GPR[rs]63 = 0) and (GPR[rs] ≠ 0^64)
      T+1:  if condition then
                  PC ← PC + target
            endif
```

Exceptions:

None

BGTZL Branch On Greater Than Zero Likely BGTZL



Format:

BGTZL rs, offset

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted two bits left and sign-extended. The contents of general purpose register *rs* are larger than 0, then the program branches to the branch address, with a delay of one instruction.

If it does not branch, the instruction in the branch delay slot is discarded.

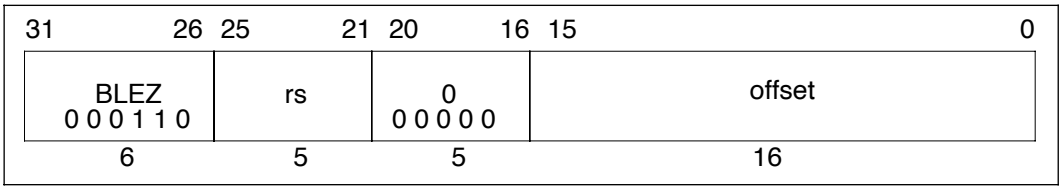
Operation:

```
32    T:    target ← (offset15)^14 || offset || 0^2
            condition ← (GPR[rs]31 = 0) and (GPR[rs] ≠ 0^32)
      T+1:  if condition then
                  PC ← PC + target
            else
                  NullifyCurrentInstruction
            endif
64    T:    target ← (offset15)^46 || offset || 0^2
            condition ← (GPR[rs]63 = 0) and (GPR[rs] ≠ 0^64)
      T+1:  if condition then
                  PC ← PC + target
            else
                  NullifyCurrentInstruction
            endif
```

Exceptions:

None

BLEZ — Branch On Less Than Or Equal To Zero — BLEZ



Format:

BLEZ rs, offset

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted two bits left and sign-extended. If the contents of general purpose register *rs* are equal to 0 or smaller than 0, then the program branches to the branch address, with a delay of one instruction.

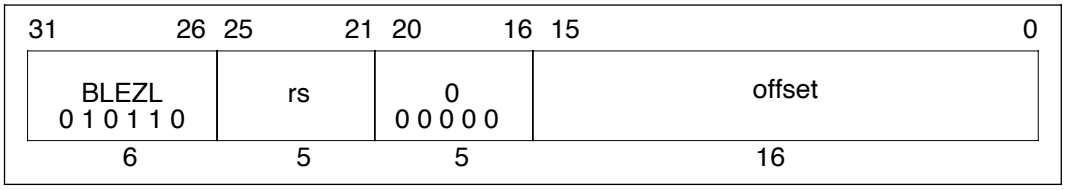
Operation:

```
32    T:    target ← (offset15)^14 || offset || 0^2
            condition ← (GPR[rs]31 = 1) or (GPR[rs] = 0^32)
      T+1:  if condition then
                  PC ← PC + target
            endif
64    T:    target ← (offset15)^46 || offset || 0^2
            condition ← (GPR[rs]63 = 1) and (GPR[rs] = 0^64)
      T+1:  if condition then
                  PC ← PC + target
            endif
```

Exceptions:

None

BLEZL Branch On Less Than Or Equal To Zero Likely BLEZL



Format:

BLEZL rs, offset

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted two bits left and sign-extended. The contents of general purpose register *rs* is equal to or smaller than zero, then the program branches to the branch address, with a delay of one instruction.

If it does not branch, the instruction in the branch delay slot is discarded.

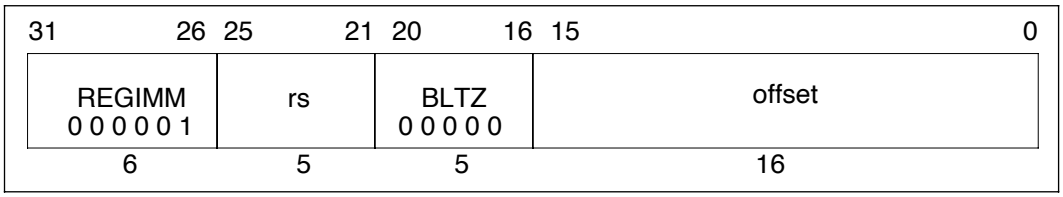
Operation:

```
32      T:    target ← (offset15)^14 || offset || 0^2
              condition ← (GPR[rs]31 = 1) or  (GPR[rs] = 0^32)
        T+1:  if condition then
                    PC ← PC + target
              else
                    NullifyCurrentInstruction
              endif
64      T:    target ← (offset15)^46 || offset || 0^2
              condition ← (GPR[rs]63 = 1) and  (GPR[rs] = 0^64)
        T+1:  if condition then
                    PC ← PC + target
              else
                    NullifyCurrentInstruction
              endif
```

Exceptions:

None

BLTZ Branch On Less Than Zero BLTZ



Format:

BLTZ rs, offset

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted two bits left and sign-extended. If the contents of general purpose register *rs* are smaller than 0, then the program branches to the branch address, with a delay of one instruction.

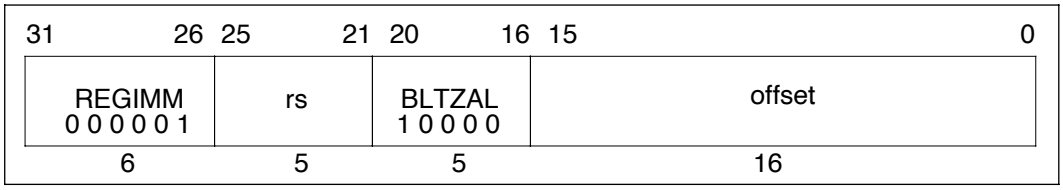
Operation:

```
32    T:    target ← (offset15)^14 || offset || 0^2
            condition ← (GPR[rs]31 = 1)
      T+1:  if condition then
                  PC ← PC + target
            endif
64    T:    target ← (offset15)^46 || offset || 0^2
            condition ← (GPR[rs]63 = 1)
      T+1:  if condition then
                  PC ← PC + target
            endif
```

Exceptions:

None

BLTZAL Branch On Less Than Zero And Link BLTZAL



Format:

BLTZAL rs, offset

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset,* shifted two bits left and sign-extended. Unconditionally, the address of the instruction next to the delay slot is stored in the link register, *r31*. If the contents of general purpose register *rs* are smaller than 0, then the program branches to the branch address, with a delay of one instruction.

Generally, general purpose register *r31* should not be specified as general purpose register *rs*, because the contents of *rs* are destroyed by storing link address, and then it is not reexecutable. An attempt to execute this instruction does not generate exceptions, however.

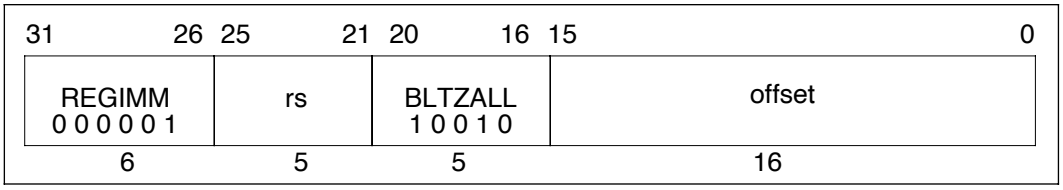
Operation:

```
32    T:    target ← (offset15)^14 || offset || 0^2
            condition ← (GPR[rs]31 = 1)
            GPR[31] ← PC + 8
      T+1:  if condition then
                  PC ← PC + target
            endif
64    T:    target ← (offset15)^46 || offset || 0^2
            condition ← (GPR[rs]63 = 1)
            GPR[31] ← PC + 8
      T+1:  if condition then
                  PC ← PC + target
            endif
```

Exceptions:

None

BLTZALL Branch On Less Than Zero And Link Likely BLTZALL



Format:

BLTZALL rs, offset

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted two bits left and sign-extended. Unconditionally, the instruction next to the delay slot is stored in the link register, *r31*. If the contents of general purpose register *rs* is smaller than 0, then the program branches to the branch address, with a delay of one instruction.
If it does not branch, the instruction in the branch delay slot is discarded.
Generally, general purpose register *r31* should not be specified as general purpose register *rs*, because the contents of *rs* are destroyed by storing link address, and then it is not reexecutable. An attempt to execute this instruction does not cause exception, however.

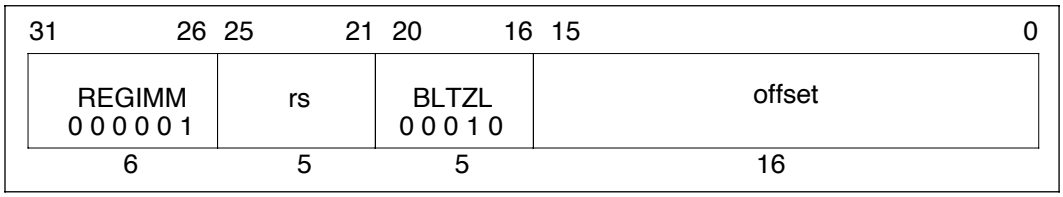
Operation:

```
32    T:    target ← (offset15)^14 || offset || 0^2
            condition ← (GPR[rs]31 = 1)
            GPR[31] ← PC + 8
      T+1:  if condition then
                  PC ← PC + target
            else NullifyCurrentInstruction
            endif
64    T:    target ← (offset15)^46 || offset || 0^2
            condition ← (GPR[rs]63 = 1)
            GPR[31] ← PC + 8
      T+1:  if condition then
                  PC ← PC + target
            else NullifyCurrentInstruction
            endif
```

Exceptions:

None

BLTZL Branch On Less Than Zero Likely BLTZL



Format:

BLTZL rs, offset

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted two bits left and sign-extended. Unconditionally, the instruction next to the delay slot is stored in the link register, *r31*. If the contents of general purpose register *rs are* smaller than 0, then the program branches to the branch address, with a delay of one instruction.

If it does not branch, the instruction in the branch delay slot is discarded.

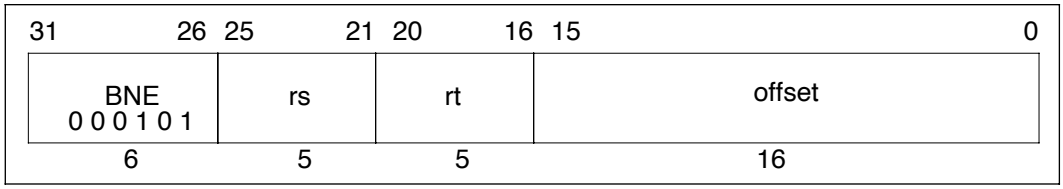
Operation:

```
32    T:    target ← (offset15)^14 || offset || 0^2
            condition ← (GPR[rs]31 = 1)
      T+1:  if condition then
                  PC ← PC + target
            else
                  NullifyCurrentInstruction
            endif
64    T:    target ← (offset15)^46 || offset || 0^2
            condition ← (GPR[rs]63 = 1)
      T+1:  if condition then
                  PC ← PC + target
            else
                  NullifyCurrentInstruction
            endif
```

Exceptions:

None

BNE Branch On Not Equal BNE



Format:

BNE rs, rt, offset

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset,* shifted two bits left and sign-extended. The contents of general purpose register *rs* and the contents of general purpose register *rt* are compared. If the two registers are not equal, then the program branches to the branch address, with a delay of one instruction.

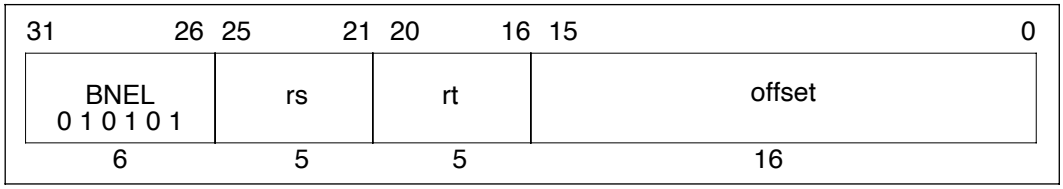
Operation:

```
32   T:    target ← (offset15)^14 || offset || 0^2
           condition ← (GPR[rs] ≠ GPR[rt])
      T+1: if condition then
                PC ← PC + target
           endif
64   T:    target ← (offset15)^46 || offset || 0^2
           condition ← (GPR[rs] ≠ GPR[rt])
      T+1: if condition then
                PC ← PC + target
           endif
```

Exceptions:

None

BNEL Branch On Not Equal Likely BNEL



Format:

BNEL rs, rt, offset

Description:

A branch address is calculated from the sum of the address of the instruction in the delay slot and the 16-bit *offset,* shifted two bits left and sign-extended. The contents of general purpose register *rs* and the contents of general purpose register *rt* are compared. If the two registers are not equal, then the program branches to the branch address, with a delay of one instruction.

If it does not branch, the instruction in the branch delay slot is discarded.

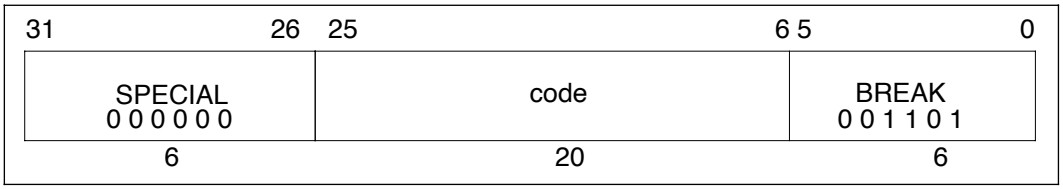
Operation:

```
32      T:    target ← (offset15)^14 || offset || 0^2
              condition ← (GPR[rs] ≠ GPR[rt])
        T+1:  if condition then
                    PC ← PC + target
              else
                    NullifyCurrentInstruction
              endif
64      T:    target ← (offset15)^46 || offset || 0^2
              condition ← (GPR[rs] ≠ GPR[rt])
        T+1:  if condition then
                    PC ← PC + target
              else
                    NullifyCurrentInstruction
              endif
```

Exceptions:

None

BREAK Breakpoint BREAK



Format:

BREAK

Description:

A breakpoint exception occurs after execution of this instruction, transferring control to the exception handler.

The code area is available for use to transfer parameters to the exception handler, the parameter is retrieved by the exception handler only by loading the contents of the memory word containing the instruction as data.

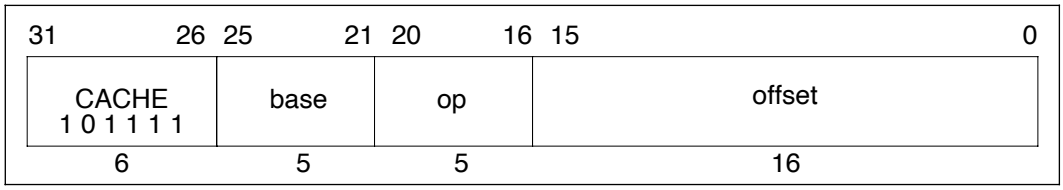
Operation:

```
32, 64      T:    BreakpointException
```

Exceptions:

Breakpoint exception

CACHE Cache Operation CACHE



Format:

CACHE op, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general purpose 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 op specifies a cache operation contents for the specified address.

CP0 is not usable if the CP0 enable bit CU_0 in the *Status* register in the User or Supervisor mode is cleared, and a coprocessor unusable exception occurs after execution of this instruction. The execution of this instruction on any cache/ operation combination not listed below, or on a secondary cache which is not supplied to V_R4300, is undefined. The execution of this instruction in uncached area is also undefined.

The Index operation uses a part of the virtual address to specify a cache block. For example a cache of $2^{CACHEBITS}$ bytes with $2^{LINEBITS}$ bytes per tag, $vAddr_{CACHEBITS \ldots LINEBITS}$ specifies the block.

The Hit operation accesses the cache as normal data references, and performs the specified cache operation only if the cache contains valid data of the specified physical address (a hit). If data is not in the cache (a miss), the cache operation is not executed.

CACHE Cache Operation (continued) CACHE

Write back from a cache goes to the main memory. The address in the main memory to be written is the address in the cache tag and not the physical address translated by using TLB.

The TLB miss exception and TLB invalid exception may occur when any cache operation is performed. The Index* operation executed to the address in the unmapped area is used to prevent occurrence of the TLB exception. The Index operation never generates the TLB change exception. Bits 16 and 17 of the instruction code indicate the cache subject to the operation as follows.

Code	Symbol	Cache
0	I	instruction cache
1	D	data cache
2	–	reserved
3	–	reserved

* Although a physical address is used to index the cache, it does not have to coincide with the cache tag.

Bits 20:18 of this instruction specify the contents of the cache operation. For details, refer to the following pages.

CACHE Cache Operation (continued) CACHE

$op_{4...2}$	Caches	Cache Operation	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 of the data cache block at the Invalidate index specified by the virtual address. If the state is not Invalid, then write back the block to main memory. The address to write is taken from the cache tag. Set cache state of 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 T*agLo* register of the CP0.
2	I, D	Index_Store_Tag	Write the contents of the *Lo* register of the CP0 register to the tag for the cache block at the specified index.
3	D	Create_Dirty_Exclusive	This operation is used to load as little data as possible from main memory when writing new data into the entire cache block where the coherency is kept. If the cache block does not contain the specified address, and the block is dirty, write it back to main memory. In all cases, set the cache block tag to the specified physical address, set the cache state to dirty.
4	I, D	Hit_Invalidate	If the cache block contains the specified address, set the cache block state invalid.
5	D	Hit_Write_Back_Invalidate	If the cache block contains the specified address, write back the data if it is dirty, and set the cache block state invalid.
5	I	Fill	Fill the instruction cache block with the data from main memory.
6	D	Hit_Write_Back	If the cache block contains the specified address and the cache state is in the dirty state, write back the data to main memory.
6	I	Hit_Write_Back	If the cache block contains the specified address, write back the data unconditionally.

CACHE — Cache Operation (continued) — CACHE

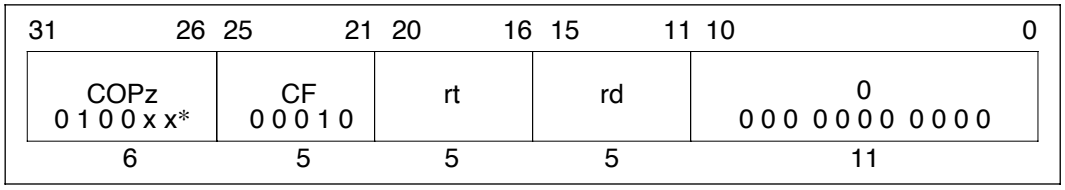
Operation:

```
32, 64   T:   vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
              (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
              CacheOp (op, vAddr, pAddr)
```

Exceptions:

Coprocessor unusable exception
TLB invalid exception
TLB miss exception
Bus error exception
Address error exception

CFCz Move Control From Coprocessor z CFCz



Format:

CFCz rt, rd

Description:

The contents of coprocessor control register *rd* of CPz are loaded to general purpose register *rt*.

This instruction is not valid for CP0.

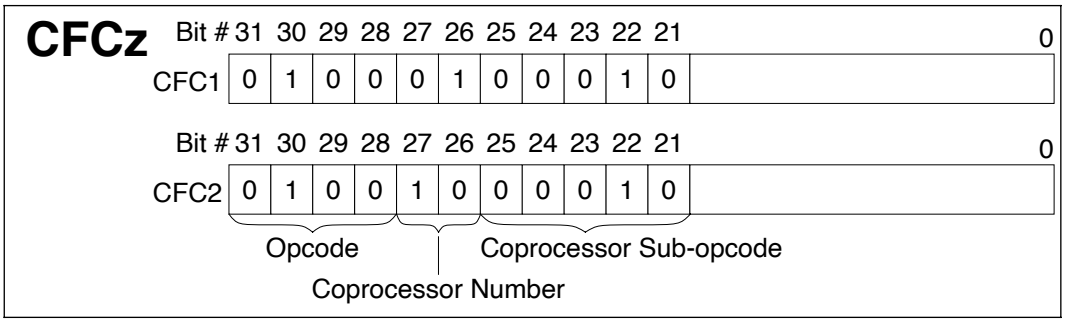
Operation:

```
32      T:    data ← CCR[z, rd]
        T+1:  GPR[rt] ← data

64      T:    data ← (CCR[z, rd]31)32 || CCR[z, rd]
        T+1:  GPR[rt] ← data
```

Exceptions:

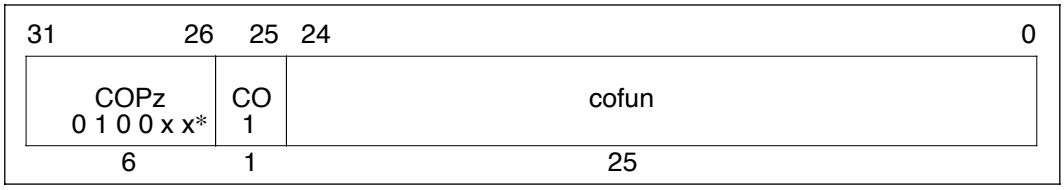
Coprocessor unusable exception

Opcode Bit Encoding:



* Refer to **16.7 CPU Instruction Opcode Bit Encoding**.

COPz Coprocessor z Operation COPz



Format:

COPz cofun

Description:

A coprocessor operation is performed. The operation may specify and reference internal coprocessor registers, and may change the state of the coprocessor condition line, but does not modify state within the processor or the cache/main memory. For details of coprocessor operations, refer to **Chapter 17 FPU Instruction Set Details**.

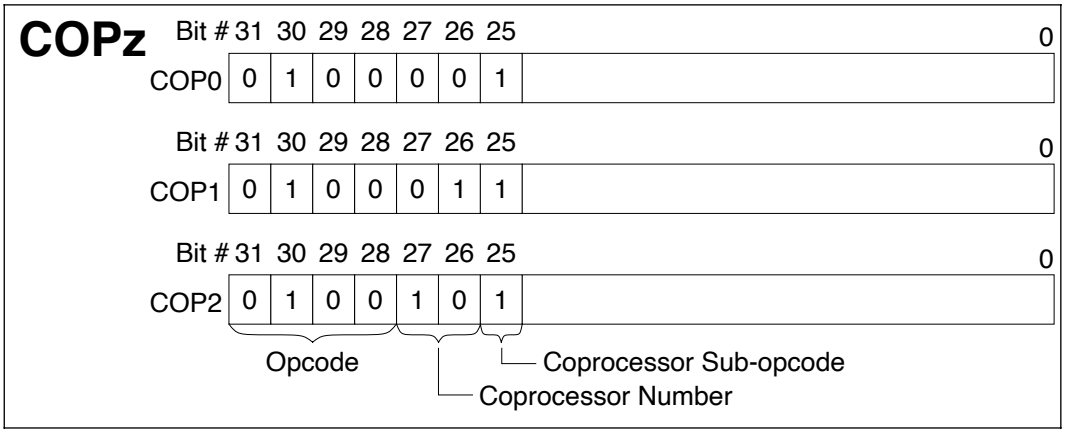
Operation:

```
32, 64      T:      CoprocessorOperation (z, cofun)
```

Exceptions:

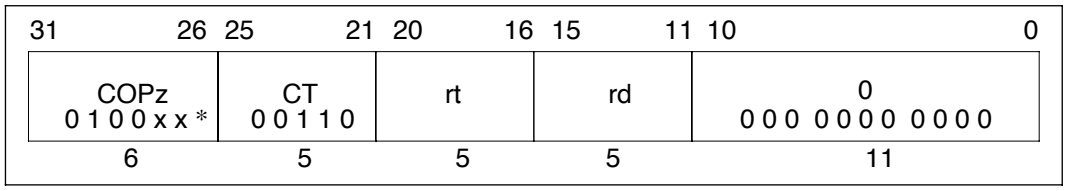
Coprocessor unusable exception
Floating-point exception (CP1 only)

Opcode Bit Encoding:



* Refer to **16.7 CPU Instruction Opcode Bit Encoding**.

CTCz Move Control To Coprocessor z CTCz



Format:

CTCz rt, rd

Description:

The contents of general purpose register *rt* are loaded into coprocessor control register *rd* of CPz. This instruction is not valid for CP0.

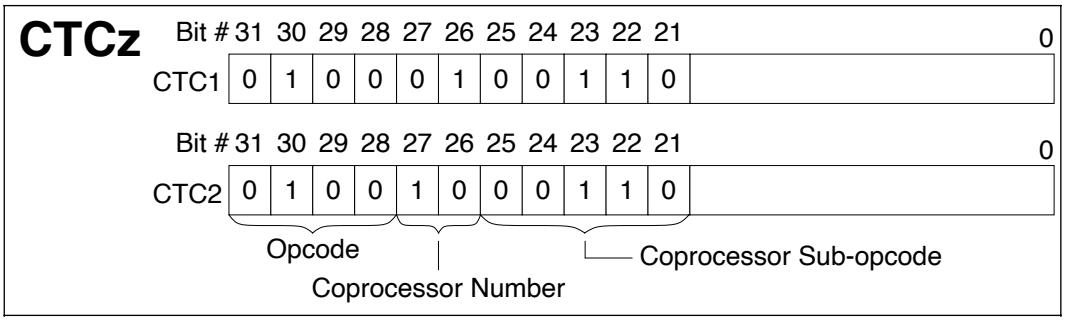
Operation:

```
32,64    T:      data ← GPR[rt]
         T + 1:  CCR[z, rd]  ← data
```

Exceptions:

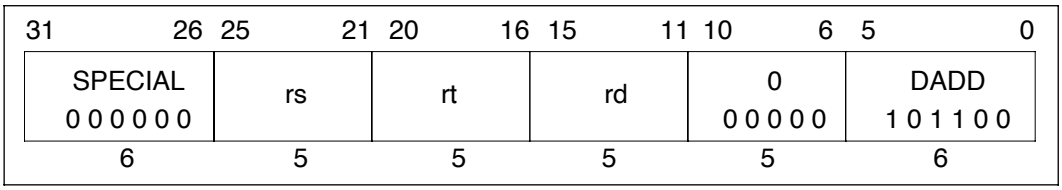
Coprocessor unusable exception

Opcode Bit Encoding:



* Refer to **16.7 CPU Instruction Opcode Bit Encoding**.

DADD Doubleword Add DADD



Format:

DADD rd, rs, rt

Description:

The contents of general purpose register *rs* and the contents of general purpose register *rt* are added, and the result is stored in general purpose register *rd*. An integer overflow exception occurs if the carries out of bits 62 and 63 differ (2's complement overflow). The contents of the destination register *rd* are not modified when an integer overflow exception occurs.

This operation is only defined for the V_R4300 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.

Operation:

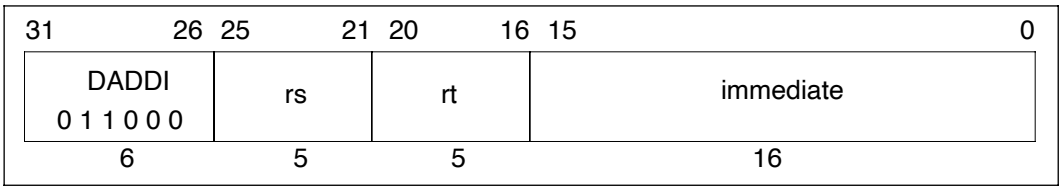
```
64      T:    GPR[rd] ←GPR[rs] + GPR[rt]
```

Remark Same operation in the 32-bit Kernel mode.

Exceptions:

Integer overflow exception
Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

DADDI Doubleword Add Immediate DADDI



Format:

DADDI rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and added to the contents of general purpose register *rs*, and the result is stored in general purpose register *rt*. An integer overflow exception occurs if carries out of bits 62 and 63 differ (2's complement overflow). The contents of the destination register *rt* are not modified when an integer overflow exception occurs.

This operation is only defined for the V_R4300 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.

Operation:

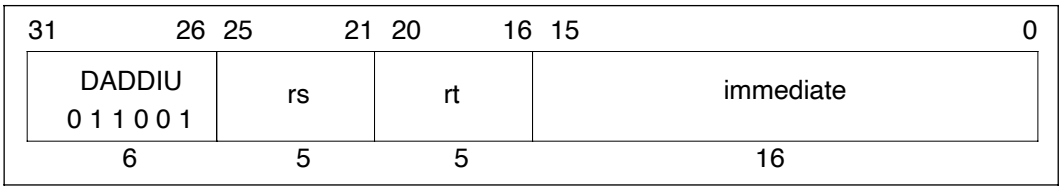
```
64      T:      GPR [rt] ← GPR[rs] + (immediate15)48 || immediate15...0
```

Remark Same operation in the 32-bit Kernel mode.

Exceptions:

Integer overflow exception
Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

DADDIU Doubleword Add Immediate Unsigned DADDIU



Format:

DADDIU rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and added to the contents of general purpose register *rs*, and the result is stored in general purpose register *rt*.

This operation is only defined for the V_R4300 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.

The only difference between this instruction and the DADDI instruction is that DADDIU instruction never causes an integer overflow exception.

Operation:

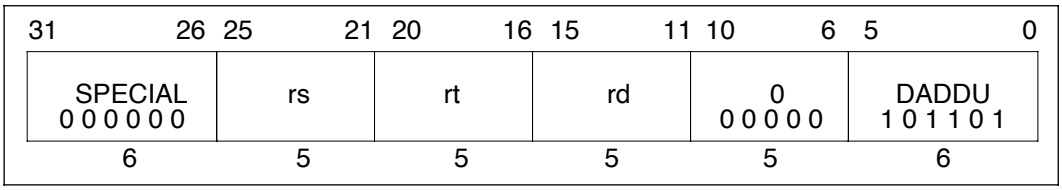
```
64    T:    GPR [rt] ← GPR[rs] + (immediate15)^48 || immediate15...0
```

Remark Same operation in the 32-bit Kernel mode.

Exceptions:

Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

DADDU Doubleword Add Unsigned DADDU



Format:

DADDU rd, rs, rt

Description:

The contents of general purpose register *rs* and the contents of general purpose register *rt* are added, and the result is stored in general purpose register *rd*.

This operation is only defined for the V_R4300 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.

The only difference between this instruction and the DADD instruction is that DADDU instruction never causes an integer overflow exception.

Operation:

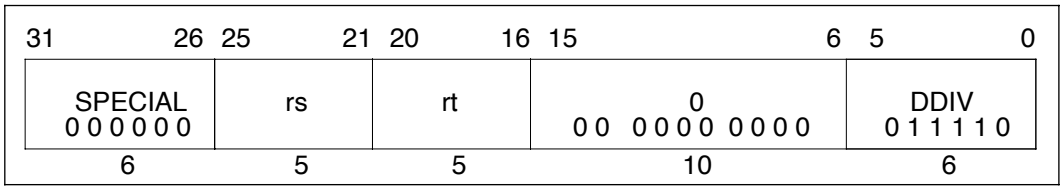
```
64      T:    GPR[rd] ←GPR[rs] + GPR[rt]
```

Remark Same operation in the 32-bit Kernel mode.

Exceptions:

Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

DDIV Doubleword Divide DDIV



Format:

DDIV rs, rt

Description:

The contents of general purpose register *rs* are divided by the contents of general purpose register *rt*, treating both operands as signed integers. An integer overflow exception never occurs, and the result of this operation is undefined when the divisor is zero.

This instruction is usually executed after 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. To obtain the correct result, insert two or more additional instructions between the MFHI or MFLO and DDIV instruction.

This operation is only defined for the V_R4300 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.

Operation:

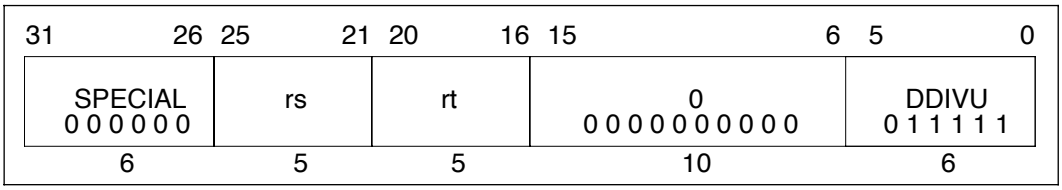
```
64      T–2:    LO      ← undefined
                HI      ← undefined
        T–1:    LO      ← undefined
                HI      ← undefined
        T:      LO      ← GPR[rs] div GPR[rt]
                HI      ← GPR[rs] mod GPR[rt]
```

Remark Same operation in the 32-bit Kernel mode.

Exceptions:

Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

DDIVU Doubleword Divide Unsigned DDIVU



Format:

DDIVU rs, rt

Description:

The contents of general purpose register *rs* are divided by the contents of general purpose register *rt*, treating both operands as unsigned integers. An integer overflow exception never occurs, and the result of this operation is undefined when the divisor is zero.

This instruction is executed after the instructions to check for a zero division.

When the operation completes, the quotient (doubleword) is stored into special register *LO*, and the remainder (doubleword) is stored into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. To obtain the correct result, insert two or more instructions in between the MFHI or MFLO and DDIVU instructions.

This operation is only defined for the V_R4300 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.

Operation:

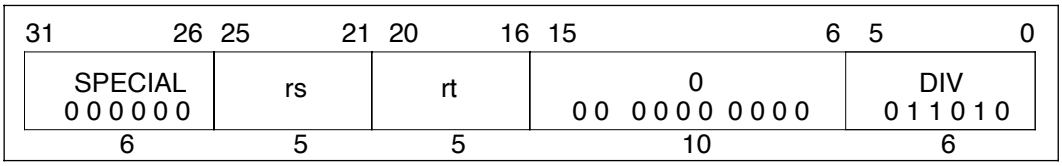
```
64      T–2:    LO      ← undefined
                HI      ← undefined
        T–1:    LO      ← undefined
                HI      ← undefined
        T:      LO      ← (0 || GPR[rs]) div (0 || GPR[rt])
                HI      ← (0 || GPR[rs]) mod  (0 || GPR[rt])
```

Remark Same operation in the 32-bit Kernel mode.

Exceptions:

Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

DIV — Divide — DIV



Format:

DIV rs, rt

Description:

The contents of general purpose register *rs* are divided by the contents of general purpose register *rt,* treating both operands as unsigned integers. An overflow exception never occurs, and the result of this operation is undefined when the divisor is zero. In 64-bit mode, the result must be sign-extended, 32-bit values.

This instruction is usually executed after the instructions to check for a zero division and for overflow.

When the operation completes, the quotient (doubleword) is stored into special register *LO*, and the remainder (doubleword) is stored into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. To obtain the correct result, insert two or more additional instructions in between the MFHI or MFLO and DIV instructions.

DIV — Divide (continued) — DIV

Operation:

32	T–2:	LO	← undefined
		HI	← undefined
	T–1:	LO	← undefined
		HI	← undefined
	T:	LO	← GPR[rs] div GPR[rt]
		HI	← GPR[rs] mod GPR[rt]
64	T–2:	LO	← undefined
		HI	← undefined
	T–1:	LO	← undefined
		HI	← undefined
	T:	q	← $GPR[rs]_{31...0}$ div $GPR[rt]_{31...0}$
		r	← $GPR[rs]_{31...0}$ mod $GPR[rt]_{31...0}$
		LO	← $(q_{31})^{32}$ ll $q_{31...0}$
		HI	← $(r_{31})^{32}$ ll $r_{31...0}$

Exceptions:

None

DIVU Divide Unsigned DIVU

Format:

DIVU rs, rt

Description:

The contents of general purpose register *rs* are divided by the contents of general purpose register *rt*, treating both operands as unsigned integers. An integer overflow exception never occurs, and the result of this operation is undefined when the divisor is zero. In 64-bit mode, the result must be sign-extended, 32-bit values.

This instruction is executed after the instructions to check for a zero division.

When the operation completes, the quotient (doubleword) is stored into special register *LO*, and the remainder (doubleword) is stored into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results of those instructions are undefined. To obtain the correct result, insert two or more additional instructions in between the MFHI or MFLO and DIVU instructions.

DIVU Divide Unsigned (continued) DIVU

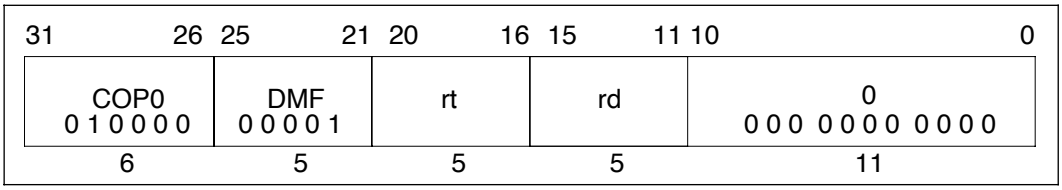
Operation:

32	T–2:	LO	← undefined
		HI	← undefined
	T–1:	LO	← undefined
		HI	← undefined
	T:	LO	← (0 || GPR[rs]) div (0 || GPR[rt])
		HI	← (0 || GPR[rs]) mod (0 || GPR[rt])
64	T–2:	LO	← undefined
		HI	← undefined
	T–1:	LO	← undefined
		HI	← undefined
	T:	q	← (0 || $GPR[rs]_{31...0}$) div (0 || $GPR[rt]_{31...0}$)
		r	← (0 || $GPR[rs]_{31...0}$) mod (0 || $GPR[rt]_{31...0}$)
		LO	← $(q_{31})^{32}$ || $q_{31...0}$
		HI	← $(r_{31})^{32}$ || $r_{31...0}$

Exceptions:

None

DMFC0 Doubleword Move From System Control Coprocessor DMFC0



Format:

DMFC0 rt, rd

Description:

The contents of coprocessor register *rd* of the CP0 are loaded into general purpose register *rt*.

This operation is defined for the V_R4300 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. The contents of the source coprocessor register *rd* are written to the 64-bit destination general purpose register *rt*. The operation of DMFC0 instruction on a 32-bit register of the CP0 is undefined.

Operation:

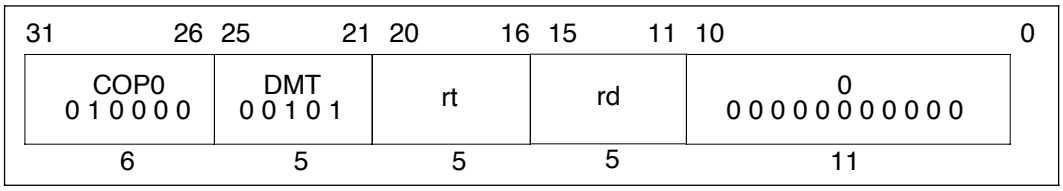
```
64      T:    data ←CPR[0,rd]
        T+1:  GPR[rt] ← data
```

Remark Same operation in the 32-bit Kernel mode.

Exceptions:

Coprocessor unusable exception	(V_R4300 in 64-/32-bit User mode and Supervisor mode if CP0 is disabled)
Reserved instruction exception	(V_R4300 in 32-bit User or Supervisor mode)

DMTC0 Doubleword Move To System Control Coprocessor DMTC0



Format:

DMTC0 rt, rd

Description:

The contents of general purpose register *rt* are loaded into coprocessor register *rd* of the CP0.

This operation is defined for the V_R4300 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.

The contents of the source general purpose register *rd* are written to the 64-bit destination coprocessor register *rt*. The operation of DMTC0 instruction on a 32-bit register of the CP0 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:

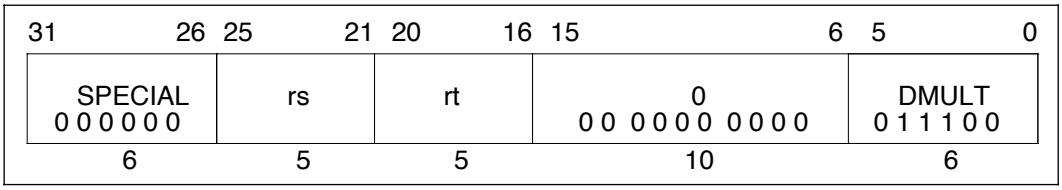
```
64      T:    data ← GPR[rt]
        T+1:  CPR[0, rd] ← data
```

Remark Same operation in the 32-bit Kernel mode.

Exceptions:

Coprocessor unusable exception	(V_R4300 in 64-/32-bit User and Supervisor mode if CP0 is disabled)
Reserved instruction exception	(V_R4300 in 32-bit User or Supervisor mode)

DMULT Doubleword Multiply DMULT



Format:

DMULT rs, rt

Description:

The contents of general purpose registers *rs* and *rt* are multiplied, treating both operands as signed integers. An integer overflow exception never occurs.

When the operation completes, the low-order doubleword is stored into special register *LO*, and the high-order doubleword is stored into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results of these instructions are undefined. To obtain the correct result, insert two or more other instructions in between the MFHI or MFLO and DMULT instructions.

This operation is only defined for the V_R4300 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.

Operation:

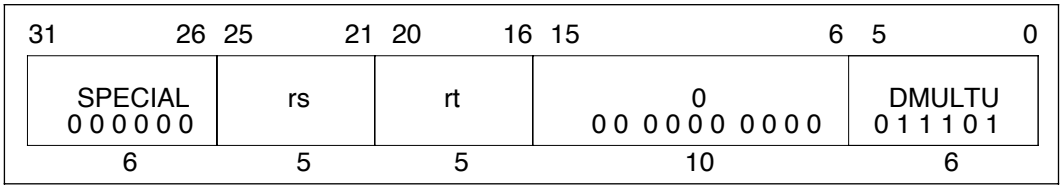
```
64      T–2:  LO      ← undefined
              HI      ← undefined
        T–1:  LO      ← undefined
              HI      ← undefined
        T:    t       ← GPR[rs] * GPR[rt]
              LO      ← t63...0
              H I     ← t127...64
```

Remark Same operation in the 32-bit Kernel mode.

Exceptions:

Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

DMULTU Doubleword Multiply Unsigned DMULTU



Format:

DMULTU rs, rt

Description:

The contents of general purpose register *rs* and the contents of general purpose register *rt* are multiplied, treating both operands as unsigned integers. An overflow exception never occurs.

When the operation completes, the low-order doubleword is stored into special register *LO*, and the high-order doubleword is stored into special register *HI*.

If either of the two preceding instructions is MFHI or MFLO, the results of these instructions are undefined. To obtain the correct result, insert two or more other instructions in between the MFHI or MFLO and DMULTU instructions.

This operation is defined for the V_R4300 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.

Operation:

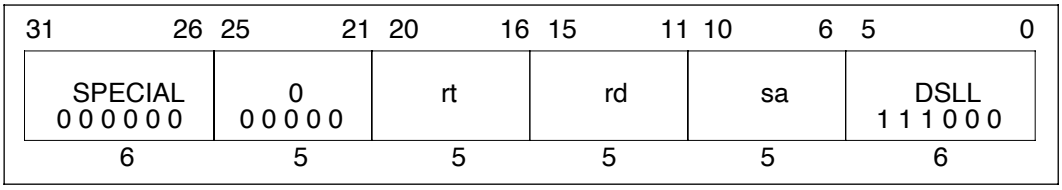
```
64      T–2:    LO ← undefined
                HI ← undefined
        T–1:    LO ← undefined
                HI ← undefined
        T:      t ← (0 || GPR[rs]) * (0 || GPR[rt])
                LO ← t63...0
                HI ←t127...64
```

Remark Same operation in the 32-bit Kernel mode.

Exceptions:

Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

DSLL Doubleword Shift Left Logical DSLL



Format:

DSLL rd, rt, sa

Description:

The contents of general purpose register *rt* are shifted left by *sa* bits, inserting zeros into the low-order bits. The result is stored in general purpose register *rd*.

This operation is defined for the V_R4300 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.

Operation:

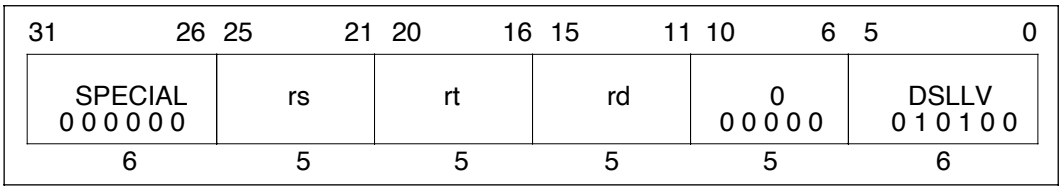
```
64      T:   s ← 0 || sa
             GPR[rd] ← GPR[rt](63–s)...0 || 0^s
```

Remark Same operation in the 32-bit Kernel mode.

Exceptions:

Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

DSLLV Doubleword Shift Left Logical Variable DSLLV



Format:

DSLLV rd, rt, rs

Description:

The contents of general purpose register *rt* are shifted left by the number of bits specified by the low-order six bits contained in general purpose register *rs*, inserting zeros into the low-order bits. The result is stored in general purpose register *rd*.

This operation is defined for the V_R4300 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.

Operation:

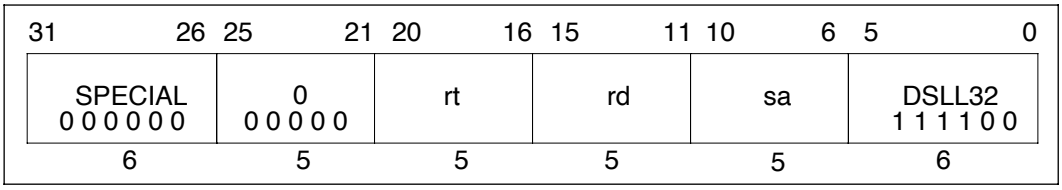
```
64          T:   s ← GPR[rs]5...0
                 GPR[rd]← GPR[rt](63–s)...0 || 0^s
```

Remark Same operation in the 32-bit Kernel mode.

Exceptions:

Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

DSLL32 Doubleword Shift Left Logical + 32 DSLL32



Format:

DSLL32 rd, rt, sa

Description:

The contents of general purpose register *rt* are shifted left by *32+sa* bits, inserting zeros into the low-order bits. The result is stored in general purpose register *rd*.

This operation is defined for the V_R4300 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.

Operation:

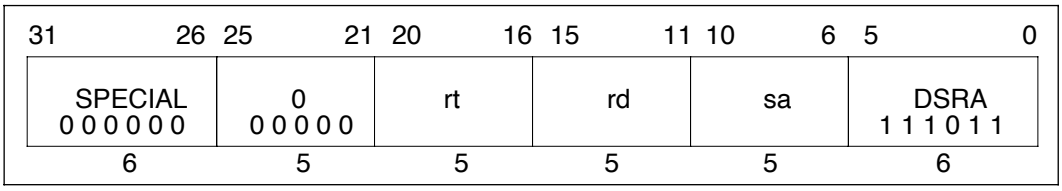
```
64          T:   s ← 1 || sa
                 GPR[rd]← GPR[rt](63–s)...0 || 0^s
```

Remark Same operation in the 32-bit Kernel mode.

Exceptions:

Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

DSRA Doubleword Shift Right Arithmetic DSRA



Format:

DSRA rd, rt, sa

Description:

The contents of general purpose register *rt* are shifted right by *sa* bits, sign-extending the high-order bits. The result is stored in general purpose register *rd*.

This operation is defined for the V_R4300 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.

Operation:

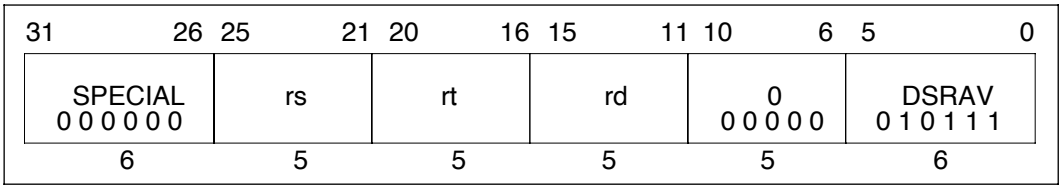
```
64          T:   s ← 0 || sa
                 GPR[rd] ← (GPR[rt]63)^s || GPR[rt] 63...s
```

Remark Same operation in the 32-bit Kernel mode.

Exceptions:

Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

DSRAV Doubleword Shift Right Arithmetic Variable DSRAV



Format:

DSRAV rd, rt, rs

Description:

The contents of general purpose register *rt* are shifted right by the number of bits specified by the low-order six bits of general purpose register *rs*, sign-extending the high-order bits. The result is stored in general purpose register *rd*.

This operation is defined for the V_R4300 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.

Operation:

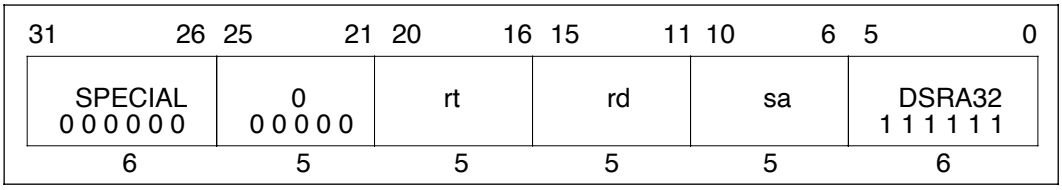
```
64      T:   s ← GPR[rs]5...0
             GPR[rd] ← (GPR[rt]63)^s || GPR[rt]63...s
```

Remark Same operation in the 32-bit Kernel mode.

Exceptions:

Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

DSRA32 Doubleword Shift Right Arithmetic + 32 DSRA32



Format:

DSRA32 rd, rt, sa

Description:

The contents of general purpose register *rt* are shifted right by *32+sa* bits, sign-extending the high-order bits. The result is stored in general purpose register *rd*.

This operation is defined for the V_R4300 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.

Operation:

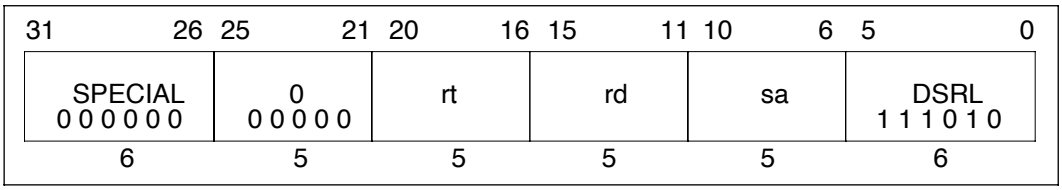
```
64      T:   s ←1 || sa
             GPR[rd] ← (GPR[rt]63)^s || GPR[rt] 63...s
```

Remark Same operation in the 32-bit Kernel mode.

Exceptions:

Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

DSRL — Doubleword Shift Right Logical — DSRL



Format:

DSRL rd, rt, sa

Description:

The contents of general purpose register *rt* are shifted right by *sa* bits, inserting zeros into the high-order bits. The result is stored in general purpose register *rd*.

This operation is defined for the V_R4300 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.

Operation:

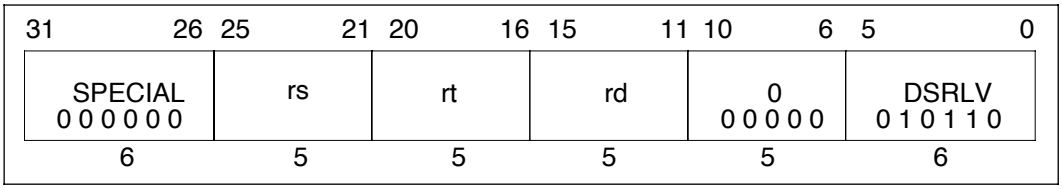
```
64      T:   s ← 0 || sa
             GPR[rd] ← 0^s || GPR[rt]63...s
```

Remark Same operation in the 32-bit Kernel mode.

Exceptions:

Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

DSRLV Doubleword Shift Right Logical Variable DSRLV



Format:

DSRLV rd, rt, rs

Description:

The contents of general purpose register *rt* are shifted right by the number of bits specified by the low-order six bits of general purpose register *rs*, inserting zeros into the high-order bits. The result is stored in general purpose register *rd*.

This operation is defined for the V_R4300 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.

Operation:

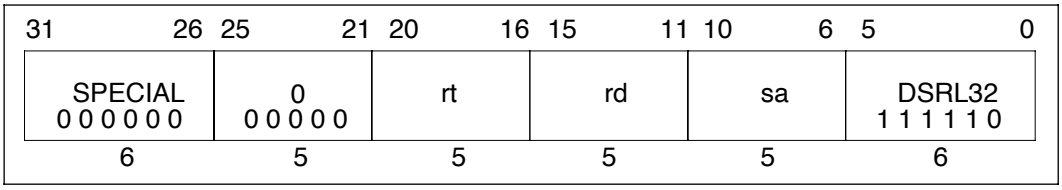
```
64          T:   s ← GPR[rs]5...0
                 GPR[rd] ← 0^s || GPR[rt]63...s
```

Remark Same operation in the 32-bit Kernel mode.

Exceptions:

Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

DSRL32 Doubleword Shift Right Logical + 32 DSRL32



Format:

DSRL32 rd, rt, sa

Description:

The contents of general purpose register *rt* are shifted right by *32+sa* bits, inserting zeros into the high-order bits. The result is stored in general purpose register *rd*.

This operation is defined for the V_R4300 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.

Operation:

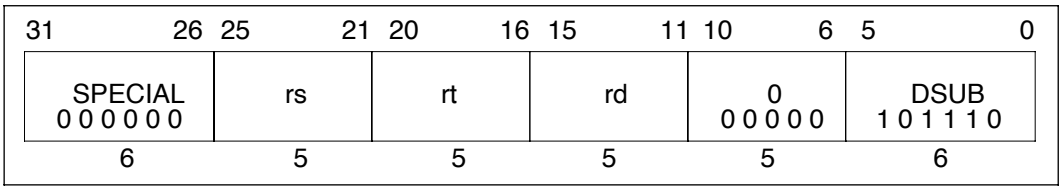
```
64          T:   s ← 1 || sa
                 GPR[rd] ← 0^s || GPR[rt]63...s
```

Remark Same operation in the 32-bit Kernel mode.

Exceptions:

Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

DSUB Doubleword Subtract DSUB



Format:

DSUB rd, rs, rt

Description:

The contents of general purpose register *rt* are subtracted from the contents of general purpose register *rs*, and the result is stored in general purpose register *rd*.

An integer overflow exception takes place if the carries out of bits 62 and 63 differ (2's complement overflow). The contents of destination register *rd* are not modified when an integer overflow exception occurs.

This operation is defined for the V_R4300 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.

Operation:

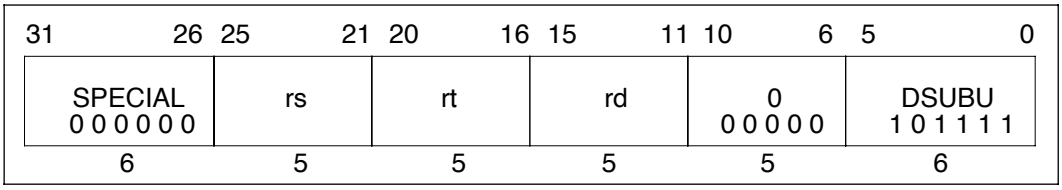
```
64          T:    GPR[rd] ← GPR[rs] – GPR[rt]
```

Remark Same operation in the 32-bit Kernel mode.

Exceptions:

Integer overflow exception
Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

DSUBU Doubleword Subtract Unsigned DSUBU



Format:

DSUBU rd, rs, rt

Description:

The contents of general purpose register *rt* are subtracted from the contents of general purpose register *rs*, and the result is stored in general purpose register *rd*.

The only difference between this instruction and the DSUB instruction is that DSUBU instruction never causes an integer overflow exception.

This operation is defined for the V_R4300 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.

Operation:

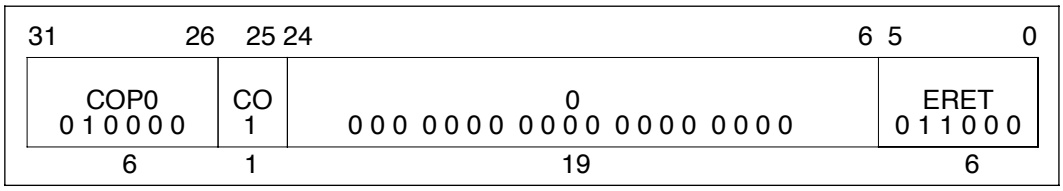
```
64          T:    GPR[rd] ← GPR[rs] – GPR[rt]
```

Remark Same operation in the 32-bit Kernel mode.

Exceptions:

Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

ERET Return From Exception ERET



Format:

ERET

Description:

ERET is the V_R4300 instruction for returning from an interrupt, exception, or error exception. Unlike a branch or jump instruction, ERET does not execute the next instruction.

ERET instruction must not itself be placed in a branch delay slot.

If the *ERL* bit of the *Status* register is set ($SR_2 = 1$), load the contents of the *ErrorEPC* register to the PC and clear the *ERL* bit to zero. Otherwise ($SR_2 = 0$), load the PC from the *EPC*, and clear the *EXL* bit of the *Status* register to zero ($SR_1 = 0$).

An ERET instruction executed between a LL instruction and SC instruction also causes the SC instruction to fail, since ERET instruction clears the *LL* bit to zero.

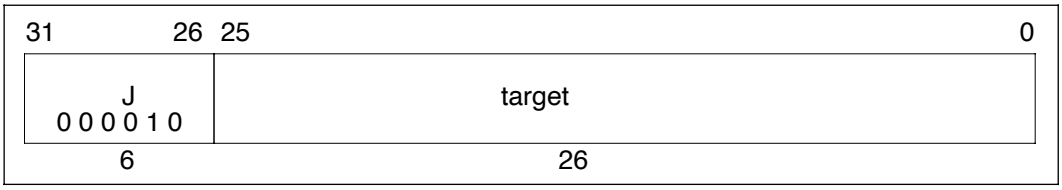
Operation:

```
32, 64      T:  if SR2 = 1 then
                    PC ← ErrorEPC
                    SR ← SR31...3 || 0 || SR1...0
                else
                    PC ← EPC
                    SR ← SR31...2 || 0 || SR0
                endif
                LLbit ← 0
```

Exceptions:

Coprocessor unusable exception

J Jump J



Format:

J target

Description:

The 26-bit target is shifted left two bits and combined with the high-order four bits of the address of the delay slot to calculate the target address. The program unconditionally jumps to this calculated address with a delay of one instruction.

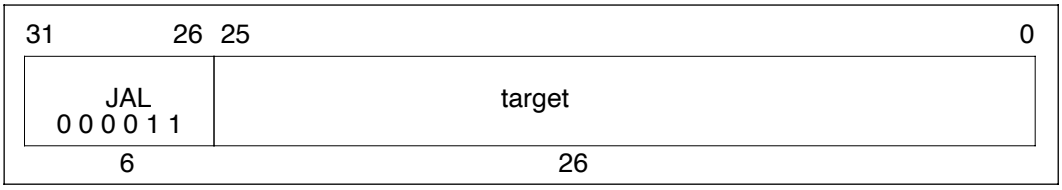
Operation:

```
32      T:    temp ← target
        T+1:  PC ← PC31...28 || temp || 0^2

64      T:    temp ← target
        T+1:  PC ← PC63...28 || temp || 0^2
```

Exceptions:

None

JAL — Jump And Link — JAL



Format:

JAL target

Description:

The 26-bit target is shifted left two bits and combined with the high-order four bits of the address of the delay slot to calculate the address. The program unconditionally jumps to this calculated address with a delay of one instruction. The address of the instruction after the delay slot is placed in the link register, *r31*.

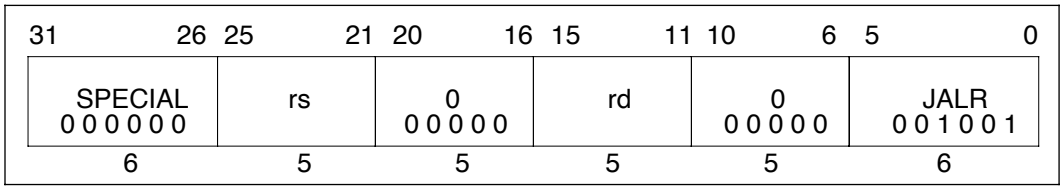
Operation:

```
32    T:    temp ← target
            GPR[31] ← PC + 8
      T+1:  PC ← PC 31...28 || temp || 0^2
64    T:    temp ← target
            GPR[31] ← PC + 8
      T+1:  PC ← PC 63...28 || temp || 0^2
```

Exceptions:

None

JALR Jump And Link Register JALR



Format:

JALR rs
JALR rd, rs

Description:

The program unconditionally jumps to the address contained in general purpose register *rs*, with a delay of one instruction. The address of the instruction after the delay slot is stored in general purpose register *rd*. The default value of *rd*, if omitted in the assembly language instruction, is 31.

Register numbers *rs* and *rd* should not be equal, because such an instruction does not have the same effect when re-executed. If they are equal, the contents of *rs* are destroyed by storing link address. However, if an attempt is made to execute this instruction, an exception will not occur, 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 low-order two bits are zero. If these low-order two bits are not zero, an address exception will occur when the jump target instruction is fetched.

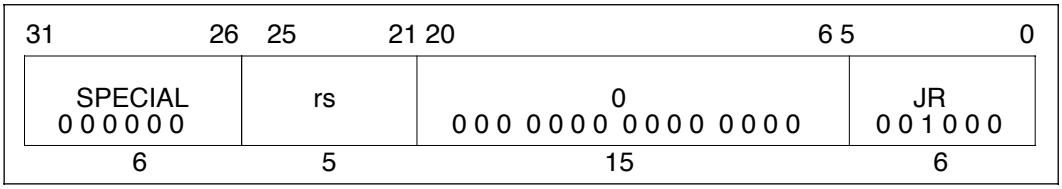
Operation:

```
32, 64      T:      temp ← GPR [rs]
                    GPR[rd] ← PC + 8
            T+1:    PC ← temp
```

Exceptions:

None

JR Jump Register JR



Format:

JR rs

Description:

The program unconditionally jumps to the address contained in general purpose 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 low-order two bits are zero. If these low-order two bits are not zero, an address exception will occur when the jump target instruction is fetched.

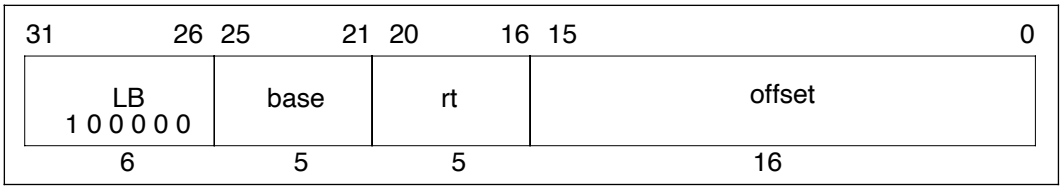
Operation:

```
32, 64      T:      temp ← GPR[rs]
            T+1:    PC ← temp
```

Exceptions:

None

LB Load Byte LB



Format:

LB rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general purpose register *base* to form a virtual address. The contents of the byte at the memory location specified by the address are sign-extended and loaded into general purpose register *rt*.

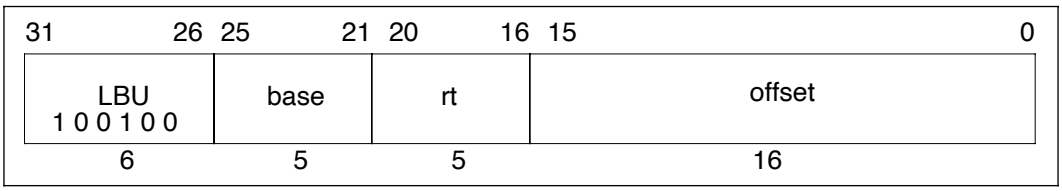
Operation:

```
32    T:    vAddr ← ((offset15)16 || offset15...0) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← pAddrPSIZE – 1 ... 3 || (pAddr2...0 xor ReverseEndian3)
            mem ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
            byte ← vAddr2...0 xor BigEndianCPU3
            GPR[rt] ← (mem7+8*byte)24 || mem7+8*byte...8*byte
64    T:    vAddr ← ((offset15)48 || offset15...0) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← pAddrPSIZE – 1 ... 3 || (pAddr2...0 xor ReverseEndian3)
            mem ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
            byte ← vAddr2...0 xor BigEndianCPU3
            GPR[rt] ← (mem7+8*byte)56 || mem7+8*byte...8*byte
```

Exceptions:

TLB miss exception
TLB invalid exception
Bus error exception
Address error exception

LBU Load Byte Unsigned LBU



Format:

LBU rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general purpose register *base* to form a virtual address. The contents of the byte at the memory location specified by the address are zero-extended and loaded into general purpose register *rt*.

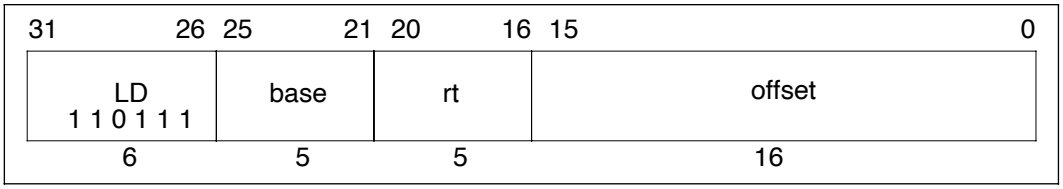
Operation:

```
32    T:   vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE – 1 ... 3 || (pAddr2...0 xor ReverseEndian^3)
           mem ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
           byte ← vAddr2...0 xor BigEndianCPU^3
           GPR[rt] ← 0^24 || mem7+8* byte...8* byte
64    T:   vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE – 1...3 || (pAddr2...0 xor ReverseEndian^3)
           mem ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
           byte ← vAddr2...0 xor BigEndianCPU^3
           GPR[rt] ← 0^56 || mem7+8* byte...8* byte
```

Exceptions:

TLB miss exception
TLB invalid exception
Bus error exception
Address error exception

LD Load Doubleword LD



Format:

LD rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general purpose register *base* to form a virtual address. The contents of the 64-bit doubleword at the memory location specified by the address are loaded into general purpose register *rt*.

If any of the low-order three bits of the address are not zero, an address error exception occurs.

This operation is defined for the V_R4300 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.

Operation:

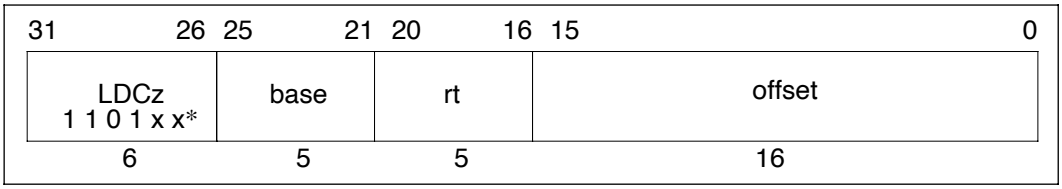
```
64    T:    vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
            GPR[rt] ← mem
```

Remark In the 32-bit Kernel mode, the high-order 32 bits are ignored during virtual address creation.

Exceptions:

TLB miss exception
TLB invalid exception
Bus error exception
Address error exception
Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

LDCz Load Doubleword To Coprocessor z LDCz



Format:

LDCz rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general purpose register *base* to form a virtual address. The processor loads a doubleword from the addressed memory location to CPz. The manner in which each coprocessor uses the data is defined by the individual coprocessor specifications.

If any of the low-order three bits of the address are not zero, an address error exception takes place.

This instruction is not valid for use with CP0.

When the CP1 is specified, the *FR* bit of the *Status* register equals zero, and the least-significant bit in the *rt* field is not zero; the operation of the instruction is undefined. If *FR* bit equals one, an odd or even register is specified by the *rt*.

* Refer to the table **Opcode Bit Encoding** on next page, or **16.7 CPU Instruction Opcode Bit Encoding**.

LDCz Load Doubleword To Coprocessor z (continued) LDCz

Operation:

```
32    T:   vAddr ← ((offset15)16 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
           COPzLD (rt, mem)

64    T:   vAddr ← ((offset15)48 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
           COPzLD (rt, mem)
```

Exceptions:

TLB miss exception
TLB invalid exception
Bus error exception
Address error exception
Coprocessor unusable exception

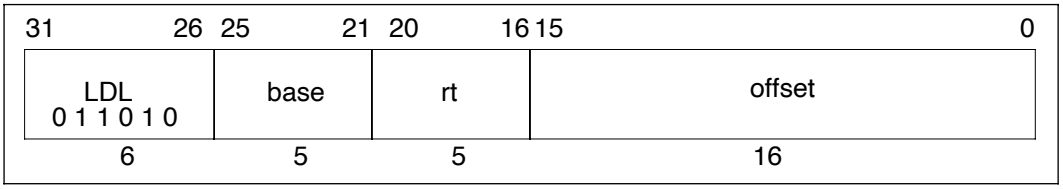
Opcode Bit Encoding:

LDCz

	Bit #	31	30	29	28	27	26	0
LDC1		1	1	0	1	0	1	
	Bit #	31	30	29	28	27	26	0
LDC2		1	1	0	1	1	0	

Opcode Coprocessor Number

LDL Load Doubleword Left LDL



Format:

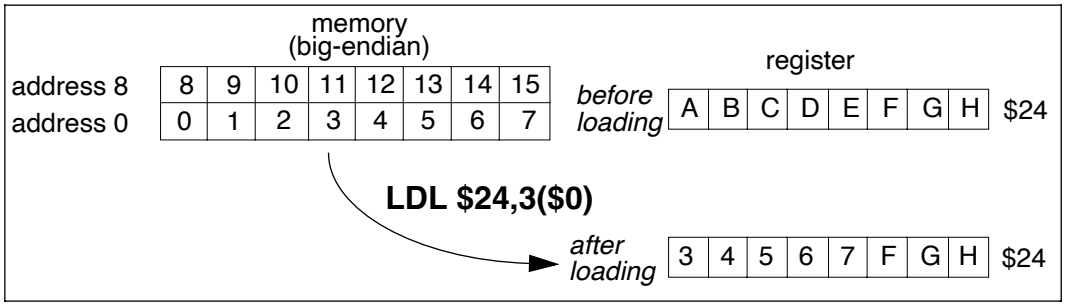
LDL rt, offset(base)

Description:

This instruction is used in combination with the LDR instruction to load the doubleword data in the memory that is not at the word boundary to general purpose register *rt*. The LDL instruction loads the high-order portion of the data to the register, while the LDR instruction loads the low-order portion.

The 16-bit offset is sign-extended and added to the contents of general purpose register base to generate a virtual address that can specify any byte. Of the doubleword data in the memory whose most-significant byte is specified by the generated address, only the data at the same word boundary as the target address is loaded and stored to the high-order portion of general purpose register *rt*. The remaining portion of the register is not affected. Depending on the address specified, the number of bytes to be loaded changes from 1 to 8.

In other words, first the addressed byte is stored to the most-significant byte position of general purpose register *rt*. If there is data of the low-order byte that follows the same doubleword boundary, the operation to store this data to the next byte of general purpose register rt is repeated. The remaining low-order byte is not affected.



LDL Load Doubleword Left (continued) LDL

The contents of general purpose register *rt* are internally bypassed within the processor so that no NOP instruction is needed between an immediately preceding load instruction which targets general purpose register *rt* and a subsequent LDL (or LDR) instruction.

The address error exception does not occur even if the specified address is not at the doubleword boundary.

This operation is defined for the V_R4300 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.

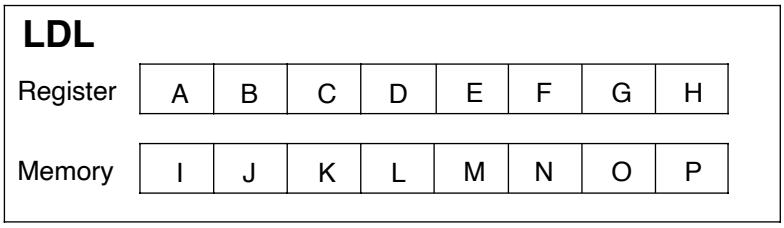
Operation:

```
64    T:   vAddr ← ((offset15)48 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE–1...3 || (pAddr2...0 xor ReverseEndian3)
           if BigEndianMem = 0 then
                      pAddr ← pAddrPSIZE–1...3 || 03
           endif
           byte ← vAddr2...0 xor BigEndianCPU3
           mem ← LoadMemory (uncached, byte, pAddr, vAddr, DATA)
           GPR[rt] ← mem7+8*byte...0 || GPR[rt]55–8*byte...0
```

Remark In the 32-bit Kernel mode, the high-order 32 bits are ignored during virtual address creation.

LDL Load Doubleword Left (continued) LDL

The relationship between the address given to the LDL instruction and the result (bytes for registers) are shown below:



$vAddr_{2...0}$	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

Remark Type: access type output to memory (Refer to **Figure 3-2 Byte Access within a Doubleword**.)

Offset: $pAddr_{2...0}$ output to memory

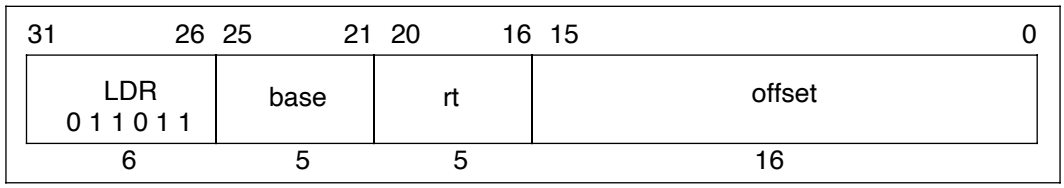
LEM Little-endian memory (BigEndianMem = 0)

BEM Big-endian memory (BigEndianMem = 1)

Exceptions:

TLB miss exception
TLB invalid exception
Bus error exception
Address error exception
Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

LDR Load Doubleword Right LDR



Format:

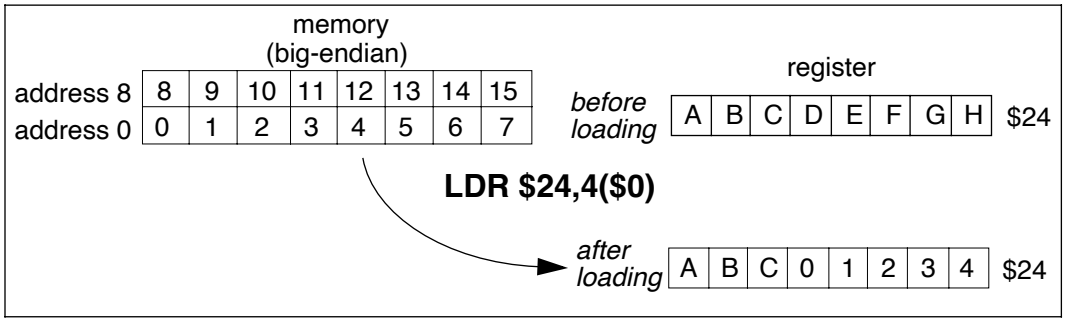
LDR rt, offset(base)

Description:

This instruction is used in combination with the LDL instruction to load the word data in the memory that is not at the word boundary to general purpose register rt. The LDL instruction loads the high-order portion of the data to the register, while the LDR instruction loads the low-order portion.

The 16-bit offset is sign-extended and added to the contents of general purpose register base to generate a virtual address that can specify any byte. Of the word data in the memory whose least-significant byte is specified by the generated address, only the data at the same doubleword boundary as the target address is loaded and stored to the low-order portion of general purpose register *rt*. The remaining portion of the register is not affected. Depending on the address specified, the number of bytes to be loaded changes from 1 to 8.

In other words, first the addressed byte is stored to the least-significant byte position of general purpose register *rt*. If there is data of the high-order byte that follows the same doubleword boundary, the operation to store this data to the next byte of general purpose register *rt* is repeated. The remaining high-order byte is not affected.



LDR Load Doubleword Right (continued) LDR

The contents of general purpose register *rt* are bypassed within the processor so that no NOP instruction is needed between an immediately preceding load instruction which targets general purpose register *rt* and a subsequent LDR (or LDL) instruction.

The address error exception does not occur even if the specified address is not located at the doubleword boundary.

This operation is defined for the V_R4300 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.

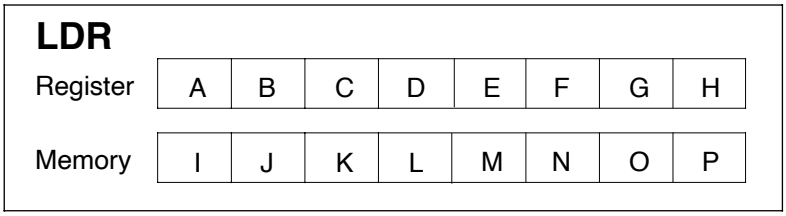
Operation:

```
64  T: vAddr ← ((offset15)48 || offset15...0) + GPR[base]
       (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
       pAddr ← pAddrPSIZE–1...3 || (pAddr2...0  xor  ReverseEndian3)
       if BigEndianMem = 1 then
              pAddr ←  pAddr31...3 || 03
       endif
       byte ← vAddr2...0 xor BigEndianCPU3
       mem ← LoadMemory (uncached, DOUBLEWORD - byte, pAddr, vAddr, DATA)
       GPR[rt] ← GPR[rt]63...64-8*byte || mem63...8*byte
```

Remark In the 32-bit Kernel mode, the high-order 32 bits are ignored during virtual address creation.

LDR — Load Doubleword Right (continued) — LDR

The relationship between the address given to the LDR instruction and the result (bytes for registers) is shown below:



$vAddr_{2..0}$	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

Remark Type: access type output to memory (Refer to **Figure 3-2 Byte Access within a Doubleword**.)

Offset: $pAddr_{2...0}$ output to memory

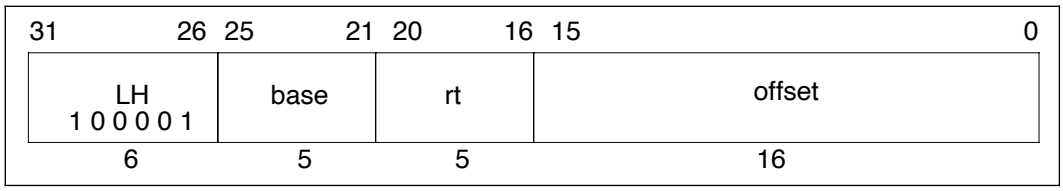
LEM Little-endian memory (BigEndianMem = 0)

BEM Big-endian memory (BigEndianMem = 1)

Exceptions:

TLB miss exception
TLB invalid exception
Bus error exception
Address error exception
Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

LH Load Halfword LH



Format:

LH rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general purpose register *base* to form a virtual address. The contents of the halfword at the memory location specified by the address are sign-extended and loaded into general purpose register *rt*.

If the least-significant bit of the address is not zero, an address error exception occurs.

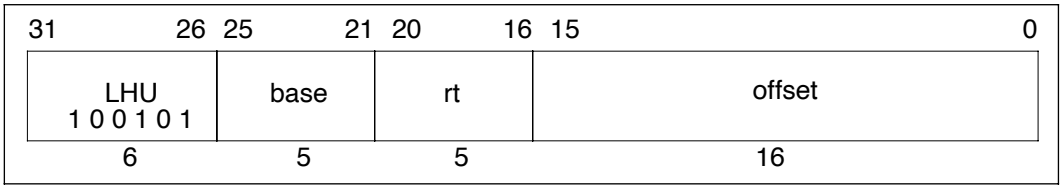
Operation:

```
32   T:  vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
         (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
         pAddr ← pAddrPSIZE – 1...3 || (pAddr2...0 xor (ReverseEndian^2 || 0))
         mem ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
         byte ← vAddr2...0 xor (BigEndianCPU^2 || 0)
         GPR[rt] ← (mem15+8*byte)^16 || mem15+8*byte...8* byte
64   T:  vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
         (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
         pAddr ← pAddrPSIZE – 1...3 || (pAddr2...0 xor (ReverseEndian^2 || 0))
         mem ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
         byte ← vAddr2...0 xor (BigEndianCPU^2 || 0)
         GPR[rt] ← (mem15+8*byte)^16 || mem15+8*byte...8* byte
```

Exceptions:

TLB miss exception
TLB invalid exception
Bus error exception
Address error exception

LHU Load Halfword Unsigned LHU



Format:

LHU rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general purpose register *base* to form a virtual address. The contents of the halfword at the memory location specified by the address are zero-extended and loaded into general purpose register *rt*.

If the least-significant bit of the address is not zero, an address error exception occurs.

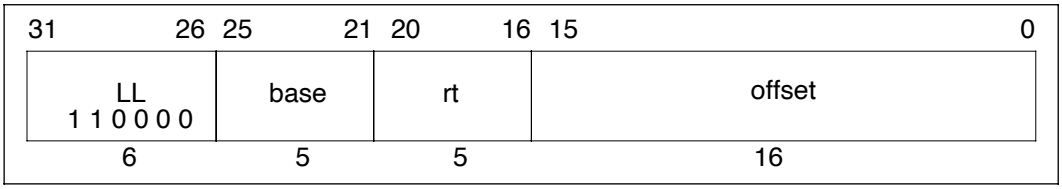
Operation:

```
32    T:   vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE – 1...3 || (pAddr2...0 xor (ReverseEndian^2 || 0))
           mem ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
           byte ← vAddr2...0 xor (BigEndianCPU^2 || 0)
           GPR[rt] ← 0^16 || mem15+8*byte...8*byte

64    T:   vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE – 1...3 || (pAddr2...0 xor (ReverseEndian^2 || 0))
           mem ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
           byte ← vAddr2...0 xor (BigEndianCPU^2 || 0)
           GPR[rt] ← 0^48 || mem15+8*byte...8*byte
```

Exceptions:

TLB miss exception	TLB invalid exception
Bus error exception	Address error exception

LL Load Linked LL



Format:

LL rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general purpose register *base* to form a virtual address. The contents of the word at the memory location specified by the address are loaded into general purpose register *rt*. In 64-bit mode, the loaded word is sign-extended. In addition, the specified physical address of the memory is stored to the LLAddr register, and sets 1 to LLbit. Afterward, the processor checks whether the address stored to the LLAddr register is not rewritten by the other processors or devices.

Load Linked (LL) and Store Conditional (SC) instructions can be used to atomically update memory:

```
L1:
    LL      T1, (T0)
    ADD     T2, T1, 1
    SC      T2, (T0)
    BEQ     T2, 0, L1
    NOP
```

This atomically increments the word addressed by T0. Changing the ADD instruction to an OR instruction changes this to an atomic bit set.

This instruction is available in User mode, and it is not necessary to enable CP0.

This instruction is defined to maintain the software compatibility with the V_R4400.

LL Load Linked (continued) LL

If the specified address is in the non-cache area, the operation of the LL instruction is undefined. A cache miss that occurs between the LL and SC instructions hinders execution of the SC instruction. Usually, therefore, do not use a load or store instruction between the LL and SC instructions. Otherwise, the operation of the SC instruction is not guaranteed. If an exception frequently occurs, the exception also hinders execution of the SC instruction. It is therefore necessary to disable the exception temporarily.

If either of the low-order two bits of the address are not zero, an address error exception takes place.

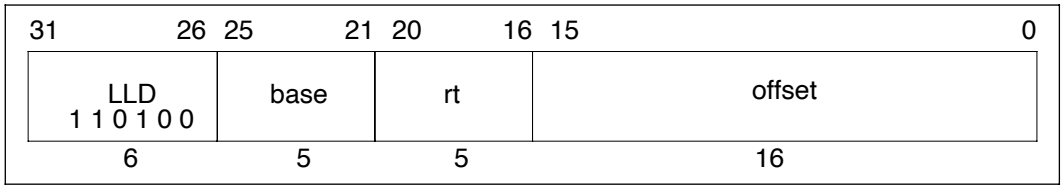
Operation:

```
32    T:   vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor (ReverseEndian || 0^2))
           mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
           byte ← vAddr2...0 xor (BigEndianCPU || 0^2)
           GPR[rt] ← mem31+8*byte...8*byte
           LLbit ← 1
           LLAddr ← pAddr
64    T:   vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor (ReverseEndian || 0^2))
           mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
           byte ← vAddr2...0 xor (BigEndianCPU || 0^2)
           GPR[rt] ← (mem31+8*byte)^32 || mem31+8*byte...8*byte
           LLbit ← 1
           LLAddr ← pAddr
```

Exceptions:

TLB miss exception
TLB invalid exception
Bus error exception
Address error exception

LLD Load Linked Doubleword LLD



Format:

LLD rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general purpose register *base* to form a virtual address. The contents of the doubleword at the memory location specified by the address are loaded into general purpose register *rt*. In addition, the specified physical address of the memory is stored to the LLAddr register, and sets 1 to LLbit. Afterward, the processor checks whether the address stored to the LLAddr register is not rewritten by the other processors or devices.

Load Linked Doubleword (LLD) instruction and Store Conditional Doubleword (SCD) instruction can be used to atomically update the memory:

```
L1:
    LLD     T1, (T0)
    DADD    T2, T1, 1
    SCD     T2, (T0)
    BEQ     T2, 0, L1
    NOP
```

This atomically increments the doubleword addressed by T0. Changing the DADD instruction to an OR instruction changes this to an atomic bit set.

This instruction is defined to maintain the software compatibility with the V_R4400.

LLD Load Linked Doubleword (continued) LLD

If the specified address is in the non-cache area, the operation of the LLD instruction is undefined. A cache miss that may occur between the LLD and SCD instructions hinders execution of the SCD instruction. Usually, therefore, do not use a load or store instruction between the LLD and SCD instructions. Otherwise, the operation of the SCD instruction will not be guaranteed. If an exception frequently occurs, the exception also hinders execution of the SCD instruction. It is therefore necessary to disable the exception temporarily.

This operation is defined for the V_R4300 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.

Operation:

```
32   T:   vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
     mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
     GPR[rt] ← mem
     LLbit ← 1
     LLAddr ← pAddr
64   T:   vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
     mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
     GPR[rt] ← mem
     LLbit ← 1
     LLAddr ← pAddr
```

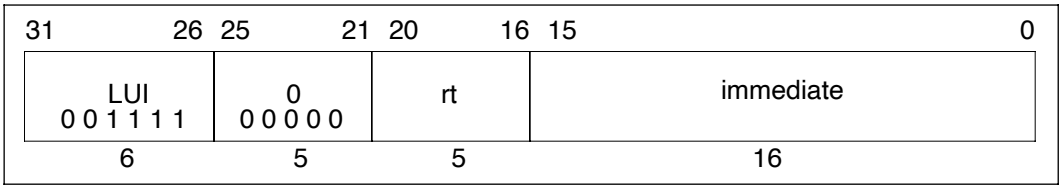
Remark In the 32-bit Kernel mode, the high-order 32 bits are ignored during virtual address creation.

LLD Load Linked Doubleword (continued) LLD

Exceptions:

TLB miss exception
TLB invalid exception
Bus error exception
Address error exception
Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

LUI Load Upper Immediate LUI



Format:

LUI rt, immediate

Description:

The 16-bit *immediate* is shifted left 16 bits and combined to 16 bits of zeros. The result is placed into general purpose register *rt*. In 64-bit mode, the loaded word is sign-extended to 64 bits.

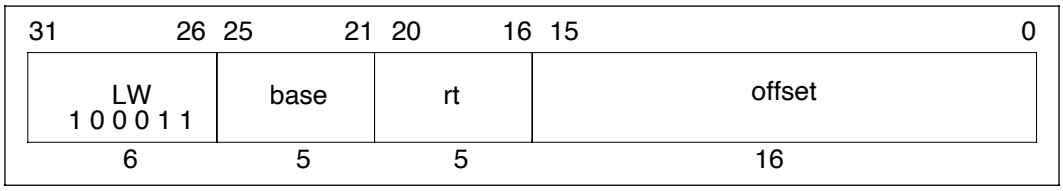
Operation:

```
32    T:    GPR[rt] ← immediate || 0^16

64    T:    GPR[rt] ← (immediate_15)^32 || immediate || 0^16
```

Exceptions:

None

LW Load Word LW



Format:

LW rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general purpose register *base* to form a virtual address. The contents of the word at the memory location specified by the address are loaded into general purpose register *rt*. In 64-bit mode, the loaded word is sign-extended to 64 bits.

If either of the low-order two bits of the address is not zero, an address error exception occurs.

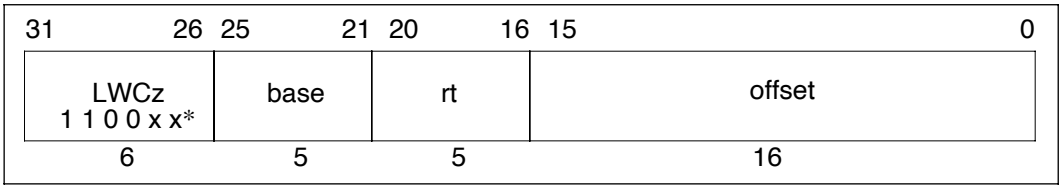
Operation:

```
32    T:    vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
            GPR[rt] ← mem

64    T:    vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
            GPR[rt] ← mem
```

Exceptions:

TLB miss exception
TLB invalid exception
Bus error exception
Address error exception

LWCz Load Word To Coprocessor z LWCz



Format:

LWCz rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general purpose register *base* to form a virtual address. The processor loads a word at the addressed memory location to the general purpose register *rt* of the CPz. The manner in which each coprocessor uses the data is defined by the individual coprocessor specifications.

If either of the low-order two bits of the address is not zero, an address error exception occurs.

This instruction is not valid for use with CP0.

* Refer to the table **Opcode Bit Encoding** on next page, or **16.7 CPU Instruction Opcode Bit Encoding**.

LWCz Load Word To Coprocessor z (continued) LWCz

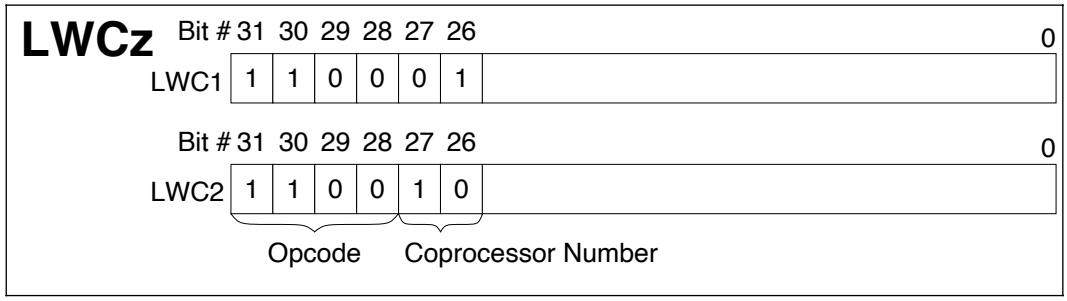
Operation:

```
32   T:   vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor (ReverseEndian || 0^2))
          mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
          byte ← vAddr2...0 xor (BigEndianCPU || 0^2)
          COPzLW (byte, rt, mem)
64   T:   vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
          (pAddr, uncached)← AddressTranslation (vAddr, DATA)
          pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor (ReverseEndian || 0^2))
          mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
          byte ← vAddr2...0 xor (BigEndianCPU || 0^2)
          COPzLW (byte, rt, mem)
```

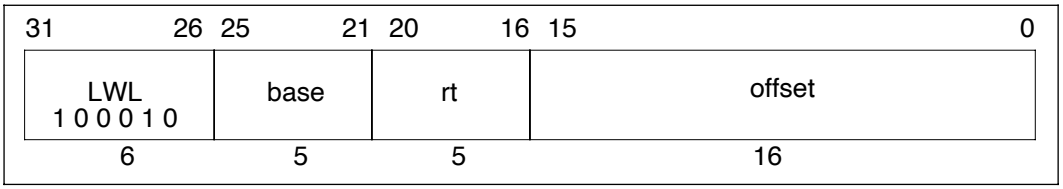
Exceptions:

TLB miss exception
TLB invalid exception
Bus error exception
Address error exception
Coprocessor unusable exception

Opcode Bit Encoding:



LWL Load Word Left LWL



Format:

LWL rt, offset(base)

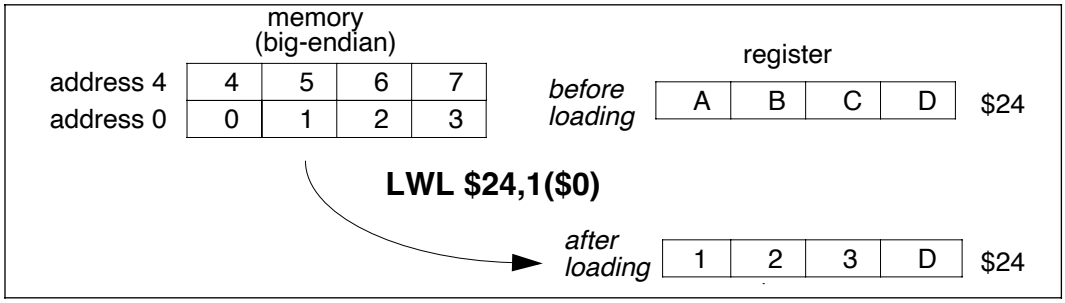
Description:

This instruction is used in combination with the LWR instruction to load the word data in the memory that is not at the word boundary to general purpose register rt. The LWL instruction loads the high-order portion of the data to the register, while the LWR instruction loads the low-order portion.

The 16-bit offset is sign-extended and added to the contents of general purpose register *base* to generate a virtual address that can specify any byte. Of the word data in the memory whose most-significant byte is specified by the generated address, only the data at the same word boundary as the target address is loaded and stored to the high-order portion of general purpose register *rt*. The remaining portion of the register is not affected. Depending on the address specified, the number of bytes to be loaded changes from 1 to 4.

In other words, first the addressed byte is stored to the most-significant byte position of general purpose register *rt*. If there is data of the low-order byte that follows the same word boundary, the operation to store this data to the next byte of general purpose register *rt* is repeated.

The remaining low-order byte is not affected.



LWL Load Word Left (continued) LWL

The contents of general purpose register *rt* are bypassed within the processor so that no NOP instruction is needed between an immediately preceding load instruction which targets general purpose register *rt* and a subsequent LWL (or LWR) instruction.

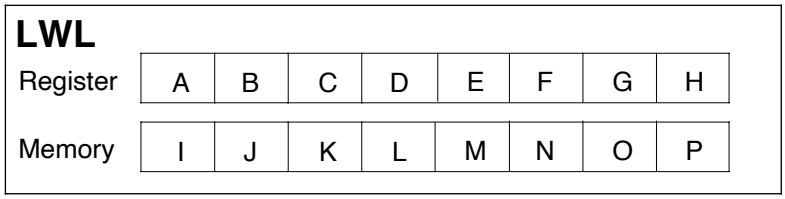
The address exception error does not occur even if the specified address is not located at the word boundary.

Operation:

```
32    T:   vAddr ← ((offset15)16 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE–1...3 || (pAddr2...0 xor ReverseEndian3)
           if BigEndianMem = 0 then
                 pAddr ← pAddrPSIZE–1...2 || 02
           endif
           byte ← vAddr1...0 xor BigEndianCPU2
           word ← vAddr2 xor BigEndianCPU
           mem ← LoadMemory (uncached, byte, pAddr, vAddr, DATA)
           temp ← mem32*word+8*byte+7 || GPR[rt]23-8*byte...0
           GRP[rt] ← temp

64    T:   vAddr ← ((offset15)48 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE–1...3 || (pAddr2...0 xor ReverseEndian3)
           if BigEndianMem = 0 then
                 pAddr ← pAddrPSIZE–1...2|| 02
           endif
           byte ← vAddr1...0 xor BigEndianCPU2
           word ← vAddr2 xor BigEndianCPU
           mem ← LoadMemory (uncached, byte, pAddr, vAddr, DATA)
           temp ← mem32*word+8*byte+7 || GPR[rt]23-8*byte...0
           GPR[rt] ← (temp31)32 || temp
```

LWL Load Word Left (continued) LWL

The relationship, between the address given to the LWL instruction and the result (bytes for registers) is shown below:



$vAddr_{2...0}$	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

Remark Type: access type output to memory (Refer to **Figure 3-2 Byte Access within a Doubleword**.)

Offset: $pAddr_{2...0}$ output to memory

LEM Little-endian memory (BigEndianMem = 0)

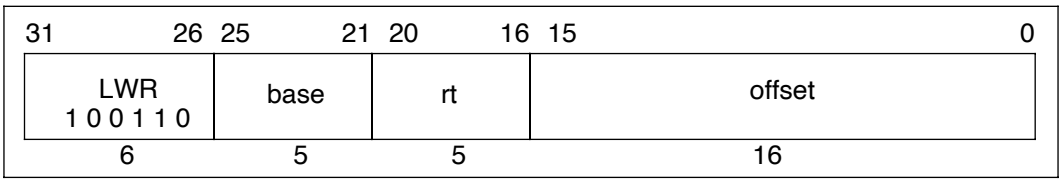
BEM Big-endian memory (BigEndianMem = 1)

S: sign-extension of destination bit 31

Exceptions:

TLB miss exception
TLB invalid exception
Bus error exception
Address error exception

LWR Load Word Right LWR



Format:

LWR rt, offset(base)

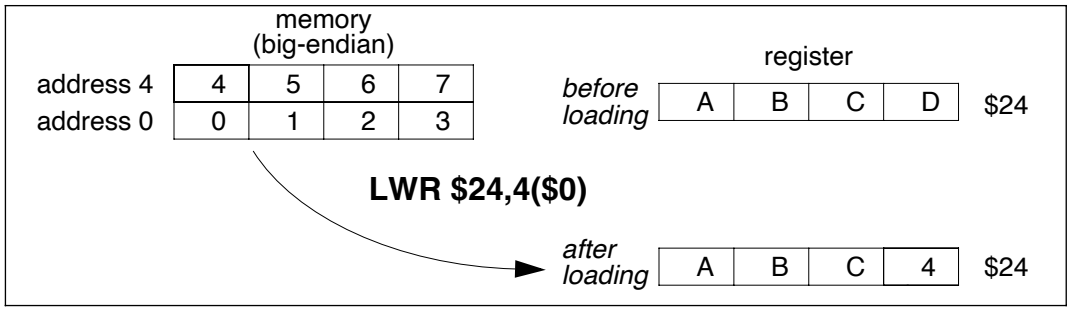
Description:

This instruction is used in combination with the LWL instruction to load the word data in the memory that is not at the word boundary to general purpose register *rt*. The LWL instruction loads the high-order portion of the data to the register, while the LWR instruction loads the low-order portion.

The 16-bit offset is sign-extended and added to the contents of general purpose register base to generate a virtual address that can specify any byte. Of the word data in the memory whose least-significant byte is specified by the generated address, only the data at the same word boundary as the target address is loaded and stored to the low-order portion of general purpose register *rt*. The remaining portion of the register is not affected. Depending on the address specified, the number of bytes to be loaded changes from 1 to 4.

In other words, first the addressed byte is stored to the least-significant byte position of general purpose register *rt*. If there is data of the high-order byte that follows the same word boundary, the operation to store this data to the next byte of general purpose register rt is repeated.

The remaining high-order byte is not affected.



LWR Load Word Right (continued) LWR

The contents of general purpose register *rt* are bypassed within the processor so that no NOP instruction is needed between an immediately preceding load instruction which targets general purpose register *rt* and a following LDL (or LWR) instruction.

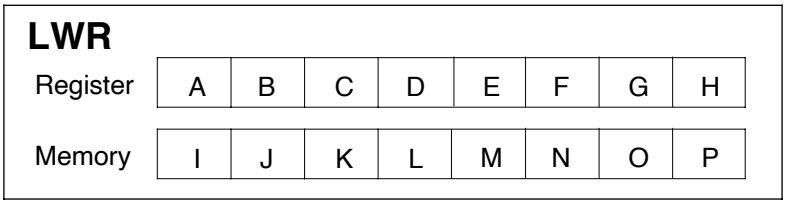
The address error exception does not occur even if the specified address is not located at the word boundary.

Operation:

```
32    T:   vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE–1...3 || (pAddr2...0 xor ReverseEndian^3)
           if BigEndianMem = 1 then
                 pAddr ← pAddrPSIZE–31...3 || 0^3
           endif
           byte ← vAddr1...0 xor BigEndianCPU^2
           word ← vAddr2 xor BigEndianCPU
           mem ← LoadMemory (uncached, 0 || byte, pAddr, vAddr, DATA)
           temp ← mem31...32-8*byte...0 || mem31+32*word-32*word+8*byte
           GPR[rt] ← temp

64    T:   vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE–1...3 || (pAddr2...0 xor ReverseEndian^3)
           if BigEndianMem = 1 then
                 pAddr ← pAddrPSIZE–31...3 || 0^3
           endif
           byte ← vAddr1...0 xor BigEndianCPU^2
           word ← vAddr2 xor BigEndianCPU
           mem ← LoadMemory (uncached, 0 || byte, pAddr, vAddr, DATA)
           temp ← mem31...32-8*byte...0 || mem31+32*word-32*word+8*byte
           GPR[rt] ← (temp31)^32 || temp
```

LWR Load Word Right (continued) LWR

The relationship between the address given to the LWR instruction and the result (bytes for registers) are shown below:



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	3	0	4	X X X X E F G I	0	7	0
1	X X X X E M N O	2	1	4	X X X X E F I J	1	6	0
2	X X X X E F M N	1	2	4	X X X X E I J K	2	5	0
3	X X X X E F G M	0	3	4	S S S S I J K L	3	4	0
4	S S S S I J K L	3	4	0	X X X X E F G M	0	3	4
5	X X X X E I J K	2	5	0	X X X X E F M N	1	2	4
6	X X X X E F I J	1	6	0	X X X X E M N O	2	1	4
7	X X X X E F G I	0	7	0	S S S S M N O P	3	0	4

Remark Type: access type output to memory (Refer to **Figure 3-2 Byte Access within a Doubleword**.)

Offset: pAddr$_{2...0}$ output to memory

LEM Little-endian memory (BigEndianMem = 0)

BEM Big-endian memory (BigEndianMem = 1)

S: Sign-extension of destination bit 31

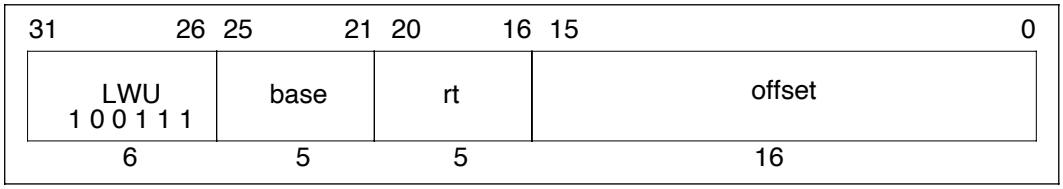
x: Not affected (in 32-bit mode)

Sign-extension of destination bit 31 (in 64-bit mode)

Exceptions:

TLB miss exception
TLB invalid exception
Bus error exception
Address error exception

LWU Load Word Unsigned LWU



Format:

LWU rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general purpose register *base* to form a virtual address. The contents of the word at the memory location specified by the address are loaded into general purpose register *rt*. The loaded word is zero-extended in 64-bit mode.

If either of the low-order two bits of the effective address is not zero, an address error exception occurs.

This operation is defined for the V_R4300 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.

Operation:

```
32   T:   vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
          GPR[rt] ← mem
64   T:   vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
          GPR[rt] ← 0^32 || mem
```

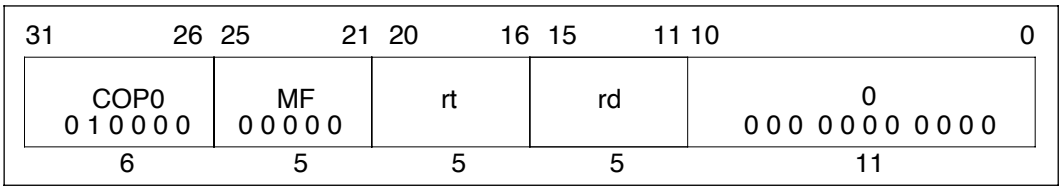
Remark In the32-bit Kernel mode, the high-order 32 bits are ignored during virtual address creation.

LWU Load Word Unsigned (continued) LWU

Exceptions:

TLB miss exception
TLB invalid exception
Bus error exception
Address error exception
Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

MFC0 Move From System Control Coprocessor MFC0



Format:

MFC0 rt, rd

Description:

The contents of general purpose register *rd* of the CP0 are loaded into general purpose register *rt*.

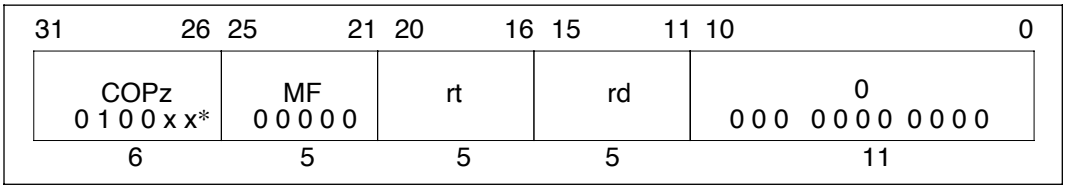
Operation:

```
32    T:    data ← CPR[0,rd]
      T+1:  GPR[rt] ← data

64    T:    data ← CPR[0,rd]
      T+1:  GPR[rt] ← (data31)^32 || data31...0
```

Exceptions:

Coprocessor unusable exception (V_R4300 in 64-/32-bit User and Supervisor mode if CP0 is disabled)

MFCz Move From Coprocessor z MFCz



Format:

MFCz rt, rd

Description:

The contents of general purpose register *rd* of CPz are loaded into general purpose register *rt*.

Operation:

```
32      T:    data ← CPR[z,rd]
        T+1:  GPR[rt] ← data
64      T:    if rd0 = 0 then
                  data ← CPR[z, rd4...1 || 0]31...0
              else
                  data ← CPR[z, rd4...1 || 0]63...32
              endif
        T+1:  GPR[rt] ← (data31)32 || data
```

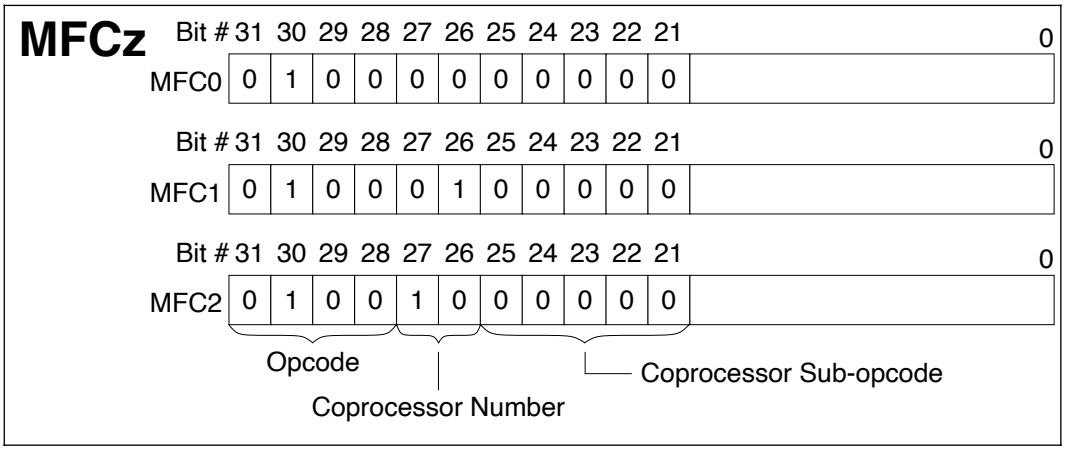
Exceptions:

Coprocessor unusable exception

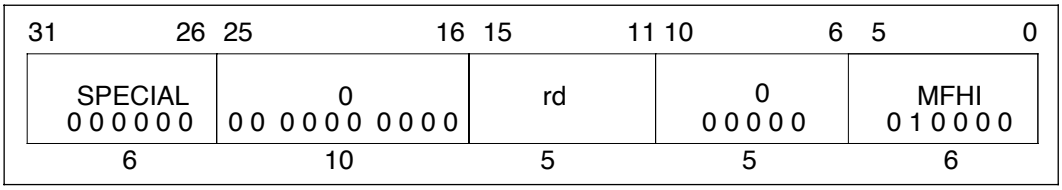
* Refer to the table **Opcode Bit Encoding** on next page, or **16.7 CPU Instruction Opcode Bit Encoding**.

MFCz Move From Coprocessor z (continued) MFCz

Opcode Bit Encoding:



MFHI Move From HI MFHI



Format:

MFHI rd

Description:

The contents of special register *HI* are loaded into general purpose register *rd*.

To ensure proper operation in the event of interruptions, the two instructions which follow a MFHI instruction may not be any of the instructions which modify the *HI* register: MULT, MULTU, DIV, DIVU, MTHI, DMULT, DMULTU, DDIV, DDIVU.

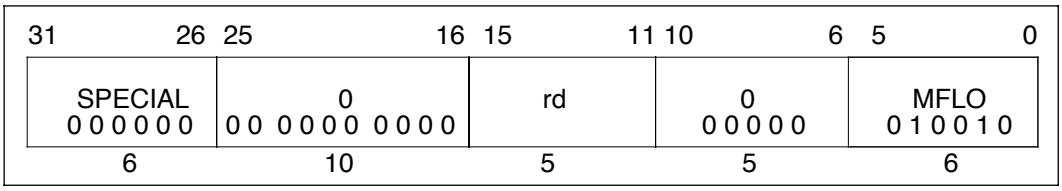
Operation:

```
32, 64      T:    GPR[rd] ← HI
```

Exceptions:

None

MFLO Move From LO MFLO



Format:

MFLO rd

Description:

The contents of special register *LO* are loaded into general purpose register *rd*.

To ensure proper operation in the event of interruptions, the two instructions which follow a MFLO instruction may not be any of the instructions which modify the *LO* register: MULT, MULTU, DIV, DIVU, MTLO, DMULT, DMULTU, DDIV, DDIVU.

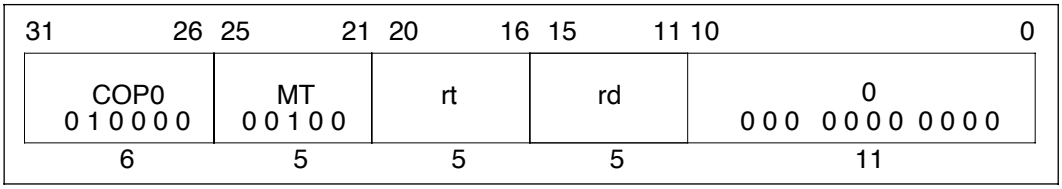
Operation:

```
32, 64      T:      GPR[rd] ← LO
```

Exceptions:

None

MTC0 Move To System Control Coprocessor MTC0



Format:

MTC0 rt, rd

Description:

The contents of general purpose register *rt* are loaded into general purpose register *rd* of CP0.

Because the contents of the TLB 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.

If the register manipulated by this instruction is used by an instruction before or after this instruction, place that instruction at an appropriate position by referring to **Chapter 19 Coprocessor 0 Hazards**.

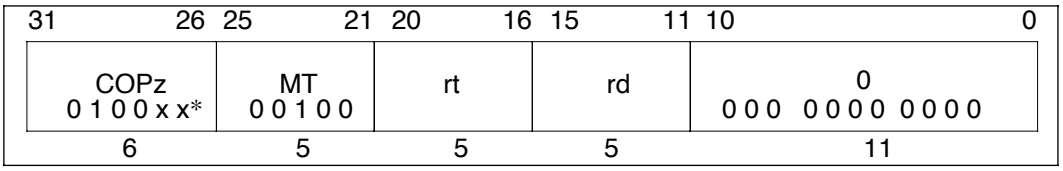
Operation:

```
32, 64    T:      data ← GPR[rt]
          T+1:    CPR[0, rd] ← data
```

Exceptions:

Coprocessor unusable exception (V_R4300 in 64-/32-bit User and Supervisor mode if CP0 is disabled)

MTCz Move To Coprocessor z MTCz



Format:

MTCz rt, rd

Description:

The contents of general purpose register *rt* are loaded into general purpose register *rd* of CPz.

Operation:

```
32    T:    data ← GPR[rt]
      T+1:  CPR[z, rd] ← data

64    T:    data ← GPR[rt]31...0
      T+1:  if rd0 = 0
                CPR[z, rd4...1 || 0] ← CPR[z, rd4...1 || 0]63...32 || data
            else
                CPR[z, rd4...1 || 0] ← data || CPR[z, rd4...1 || 0]31...0
            endif
```

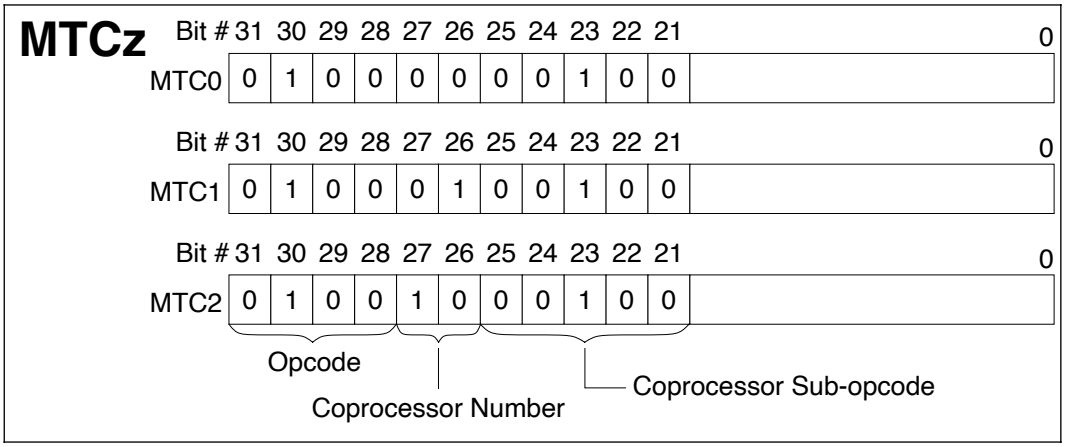
Exceptions:

Coprocessor unusable exception

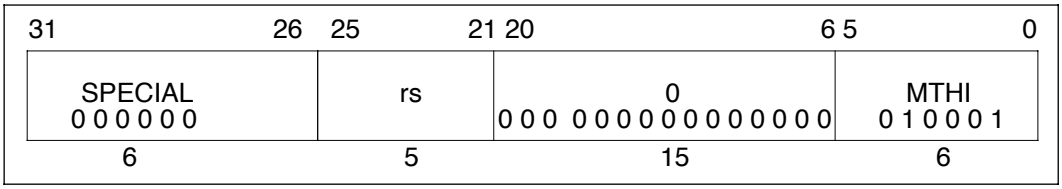
* Refer to the table **Opcode Bit Encoding** on next page, or **16.7 CPU Instruction Opcode Bit Encoding**.

MTCz Move To Coprocessor z (continued) MTCz

Opcode Bit Encoding:



MTHI Move To HI MTHI



Format:

MTHI rs

Description:

The contents of general purpose register *rs* are loaded into special register *HI*.

If the MTHI instruction is executed following the MULT, MULTU, DIV, or DIVU instruction, the operation is performed normally. However, if the MFLO, MFHI, MTLO, or MTHI instruction is executed following the MTHI instruction, the contents of special register *LO* are undefined.

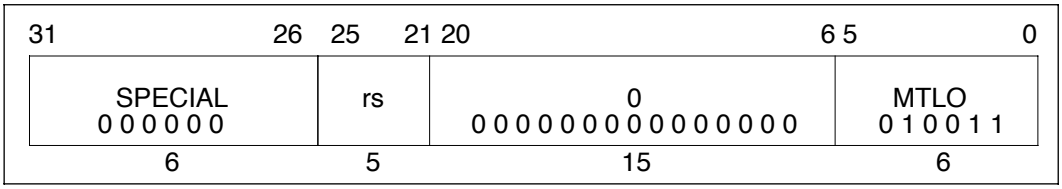
Operation:

```
32,64       T–2:  HI ← undefined
            T–1:  HI ← undefined
            T:    HI ← GPR[rs]
```

Exceptions:

None

MTLO Move To LO MTLO



Format:

MTLO rs

Description:

The contents of general purpose register *rs* are loaded into special register *LO*.

If the MTLO instruction is executed following the MULT, MULTU, DIV, or DIVU instruction, the operation is performed normally. However, if the MFLO, MFHI, MTLO, or MTHI instruction is executed following the MTLO instruction, the contents of special register *HI* are undefined.

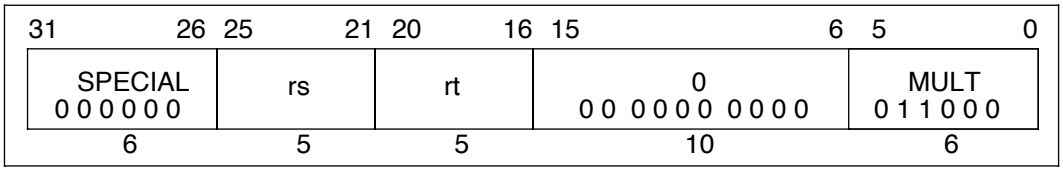
Operation:

```
32,64       T–2:  LO ← undefined
            T–1:  LO ← undefined
            T:    LO ← GPR[rs]
```

Exceptions:

None

MULT Multiply MULT



Format:

MULT rs, rt

Description:

The contents of general purpose registers *rs* and *rt* are multiplied, treating both operands as 32-bit signed integers. An integer overflow exception never occurs.

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*. In the 64-bit mode, the respective results are sign-extended and stored.

If either the two instructions immediately preceding this instruction is the MFHI or MFLO instruction, the execution result of the transfer instruction is undefined. To obtain the correct result, insert two or more other instructions in between the MFHI or MFLO and MULT instruction.

MULT Multiply (continued) MULT

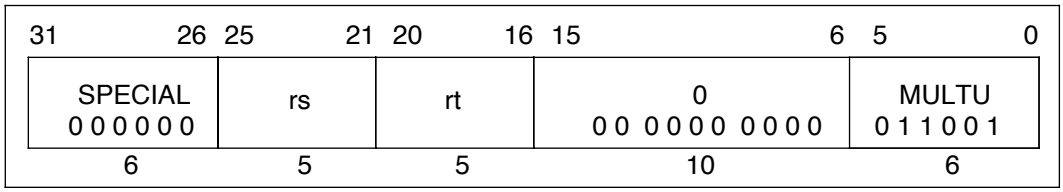
Operation:

32	T–2:	LO	← undefined
		HI	← undefined
	T–1:	LO	← undefined
		HI	← undefined
	T:	t	← GPR[rs] * GPR[rt]
		LO	← $t_{31...0}$
		H I	← $t_{63...32}$
64	T–2:	LO	← undefined
		HI	← undefined
	T–1:	LO	← undefined
		HI	← undefined
	T:	t	← $GPR[rs]_{31...0}$ * $GPR[rt]_{31...0}$
		LO	← $(t_{31})^{32}$ ll $t_{31...0}$
		HI	← $(t_{63})^{32}$ ll $t_{63...32}$

Exceptions:

None

MULTU Multiply Unsigned MULTU



Format:

MULTU rs, rt

Description:

The contents of general purpose register *rs* and the contents of general purpose register *rt* are multiplied, treating both operands as 32-bit unsigned values. An overflow exception never occurs.

In 64-bit mode, the operands must be valid 32-bit, sign-extended values.

When the operation completes, the low-order word of the doubleword result is loaded into special register *LO*, and the high-order word of the doubleword result is loaded into special register *HI*. In 64-bit mode, these results are sign-extended and loaded.

If either of the two preceding instructions is MFHI or MFLO, the execution results of these transfer instructions are undefined. To obtain the correct result, insert two or more additional instructions in between the MFHI or MFLO and MULT instructions.

MULTU — Multiply Unsigned (continued) — MULTU

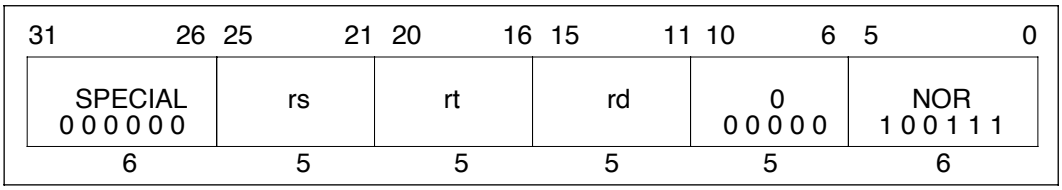
Operation:

```
32    T–2: LO    ← undefined
           HI    ← undefined
      T–1: LO    ← undefined
           HI    ← undefined
      T:   t     ← (0 || GPR[rs]) * (0 || GPR[rt])
           LO    ← t31...0
           HI    ← t63...32
64    T–2: LO    ← undefined
           HI    ← undefined
      T–1: LO    ← undefined
           HI    ← undefined
      T:   t     ← (0 || GPR[rs]31...0) * (0 || GPR[rt]31...0)
           LO    ← (t31)32 || t31...0
           HI    ← (t63)32 || t63...32
```

Exceptions:

None

NOR Nor NOR



Format:

NOR rd, rs, rt

Description:

A logical NOR operation applied between the contents of general purpose registers *rs* and *rt* is executed in bit units. The result is stored in general purpose register *rd*.

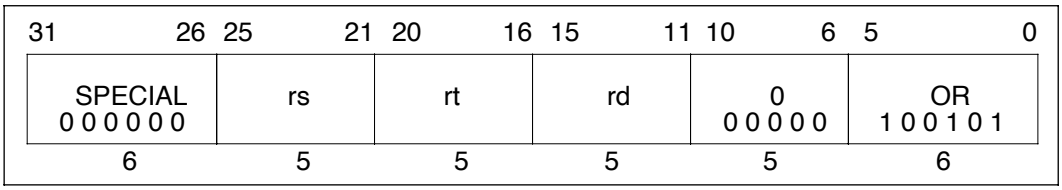
Operation:

32, 64	T:	GPR[rd] ← GPR[rs] nor GPR[rt]

Exceptions:

None

OR Or OR



Format:

OR rd, rs, rt

Description:

A logical OR operation applied between the contents of general purpose registers *rs* and *rt* is executed in bit unites. The result is stored in general purpose register *rd*.

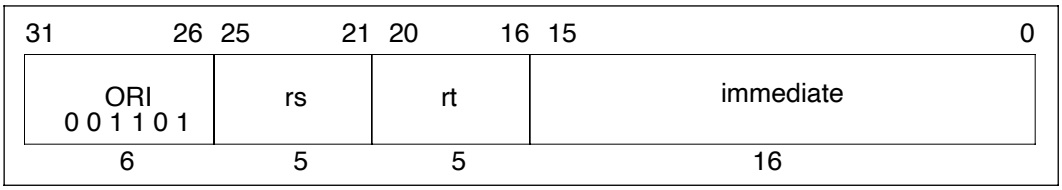
Operation:

32, 64	T:	GPR[rd] ← GPR[rs] or GPR[rt]

Exceptions:

None

ORI Or Immediate ORI



Format:

ORI rt, rs, immediate

Description:

A logical OR operation applied between 16-bit zero-extended *immediate* and the contents of general purpose register *rs* is executed in bit units. The result is stored in general purpose register *rt*.

Operation:

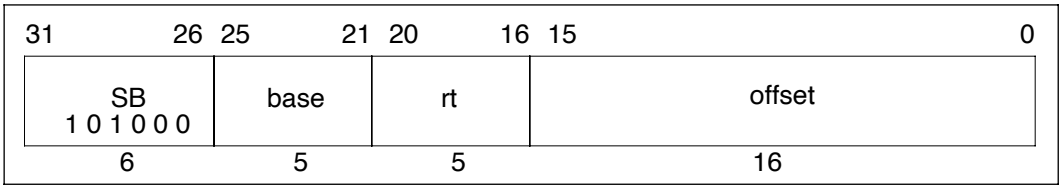
32 T: $GPR[rt] \leftarrow GPR[rs]_{31...16} \,||\, (immediate \text{ or } GPR[rs]_{15...0})$

64 T: $GPR[rt] \leftarrow GPR[rs]_{63...16} \,||\, (immediate \text{ or } GPR[rs]_{15...0})$

Exceptions:

None

SB Store Byte SB



Format:

SB rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general purpose register *base* to form a virtual address. The least-significant byte of register *rt* is stored in the memory specified by the address.

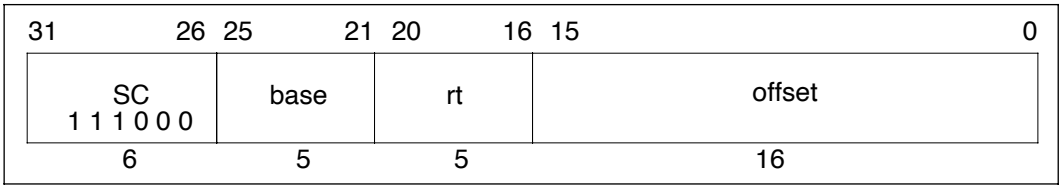
Operation:

```
32   T:  vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
         (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
         pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor ReverseEndian^3)
         byte ← vAddr2...0 xor BigEndianCPU^3
         data ← GPR[rt]63–8*byte...0 || 0^8*byte
         StoreMemory (uncached, BYTE, data, pAddr, vAddr, DATA)
64   T:  vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
         (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
         pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor ReverseEndian^3)
         byte ← vAddr2...0 xor BigEndianCPU^3
         data ← GPR[rt]63–8*byte...0 || 0^8*byte
         StoreMemory (uncached, BYTE, data, pAddr, vAddr, DATA)
```

Exceptions:

TLB miss exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception

SC Store Conditional SC



Format:

SC rt, offset(base)

Description:

The 16-bit offset is sign-extended and added to the contents of general purpose register *base* to form a virtual address. The contents of general purpose register *rt* are stored at the memory location specified by the address only when the LL bit is set.

If the other processor or device changes the physical address after the previous LL instruction has been executed, or if the ERET instruction exists between the LL and SC instructions, the register contents are not stored to the memory, and storing fails.

The success or failure of the SC operation is indicated by the contents of general purpose register *rt* after execution of the instruction. A successful SC instruction sets the contents of general purpose register *rt* to 1; an unsuccessful SC instruction sets it to 0.

The operation of SC is undefined when the address is different from the address used in the last LL instruction.

This instruction is available in User mode; it is not necessary for CP0 to be enabled.

If either of the low-order two bits of the address is not zero, an address error exception takes place.

If this instruction both fails and causes an exception, the exception takes precedence.

This instruction is defined to maintain software compatibility with the V_R4400.

SC Store Conditional (continued) SC

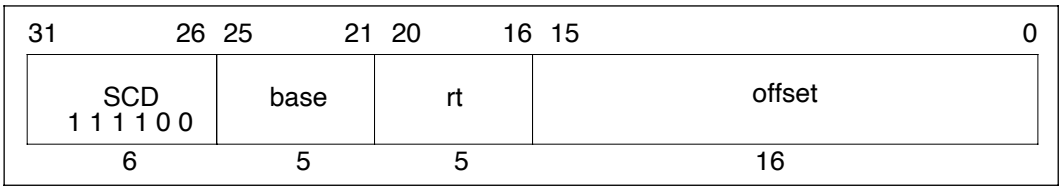
Operation:

```
32   T:   vAddr ← ((offset15)16 || offset15...0) + GPR[base]
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          data ← GPR[rt]31...0
          if LLbit then
              StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
          endif
          GPR[rt] ← 031 || LLbit
64   T:   vAddr ← ((offset15)48 || offset15...0) + GPR[base]
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          data ← GPR[rt]31...0
          if LLbit then
              StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
          endif
          GPR[rt] ← 063 || LLbit
```

Exceptions:

TLB miss exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception

SCD Store Conditional Doubleword SCD



Format:

SCD rt, offset(base)

Description:

The 16-bit offset is sign-extended and added to the contents of general purpose register *base* to form a virtual address. The contents of general purpose register *rt* are stored at the memory location specified by the address only when the LL bit is set.

If another processor or device changes the target address after the previous LLD instruction has been executed, or if the ERET instruction exists between the LLD and SCD instructions, the register contents are not stored to the memory, and storing fails.

The success or failure of the SCD operation is indicated by the contents of general purpose register *rt* after execution of the instruction. A successful SCD instruction sets the contents of general purpose register *rt* to 1; an unsuccessful SCD instruction sets it to 0.

The operation of SCD is undefined when the address is different from the address used in the last LLD.

This instruction is available in User mode; it is not necessary for CP0 to be enabled.

If either of the low-order three bits of the address is not zero, an address error exception takes place.

If this instruction both fails and causes an exception, the exception takes precedence.

This instruction is defined in the 64-bit mode and 32-bit Kernel mode. If this instruction is executed in the 32-bit User or Supervisor mode, the reserved instruction exception occurs.

This instruction is defined to maintain software compatibility with the V_R4400.

SCD Store Conditional Doubleword (continued) SCD

Operation:

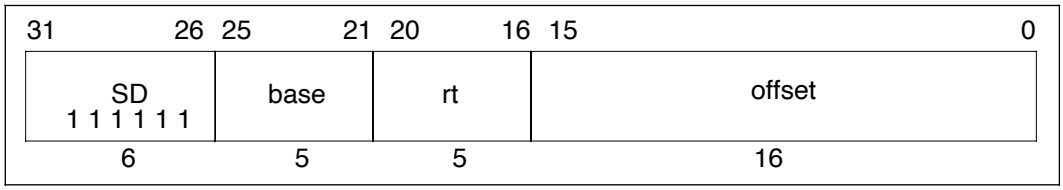
```
64    T:    vAddr ← ((offset15)48 || offset15...0) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            data ← GPR[rt]
            if LLbit then
                StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
            endif
            GPR[rt] ← 063 || LLbit
```

Remark In the 32-bit Kernel mode, the high-order 32 bits are ignored during virtual address creation.

Exceptions:

TLB miss exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception
Reserved instruction exception (32-bit User or Supervisor mode)

SD Store Doubleword SD



Format:

SD rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general purpose register *base* to form a virtual address. The contents of general purpose register *rt* are stored at the memory location specified by the address.

If either of the low-order three bits of the address are not zero, an address error exception occurs.

This operation is defined for the V_R4300 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.

Operation:

```
32   T:  vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
         (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
         data ← GPR[rt]
         StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
64   T:  vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
         (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
         data ← GPR[rt]
         StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
```

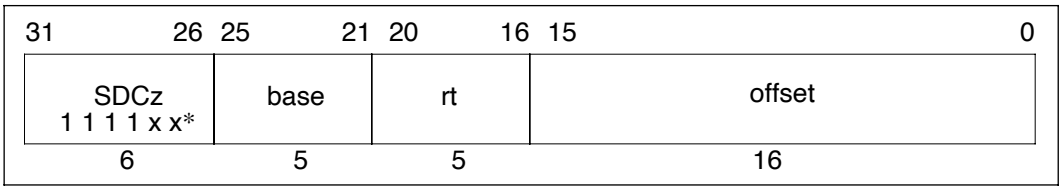
Remark In the 32-bit Kernel mode, the high-order 32 bits are ignored during virtual address creation.

SD Store Doubleword (continued) SD

Exceptions:

TLB miss exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception
Reserved instruction exception (32-bit User or Supervisor mode)

SDCz Store Doubleword From Coprocessor z SDCz



Format:

SDCz rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general purpose register *base* to form a virtual address. Register *rt* of coprocessor unit z sources a doubleword, which the processor writes to the addressed memory location. The stored data is defined by individual coprocessor specifications.

If any of the low-order three bits of the address is not zero, an address error exception takes place.

This instruction is not valid for use with CP0.

When the CP1 is specified, the *FR* bit of the *Status* register equals 0, and the least-significant bit in the *rt* field is not 0, the operation of this instruction is undefined. If the *FR* bit equals 1, both odd and even registers can be specified by *rt*.

* Refer to the table, **Opcode Bit Encoding** on next page, or **16.7 CPU Instruction Opcode Bit Encoding**.

SDCz — Store Doubleword From Coprocessor z (continued) — SDCz

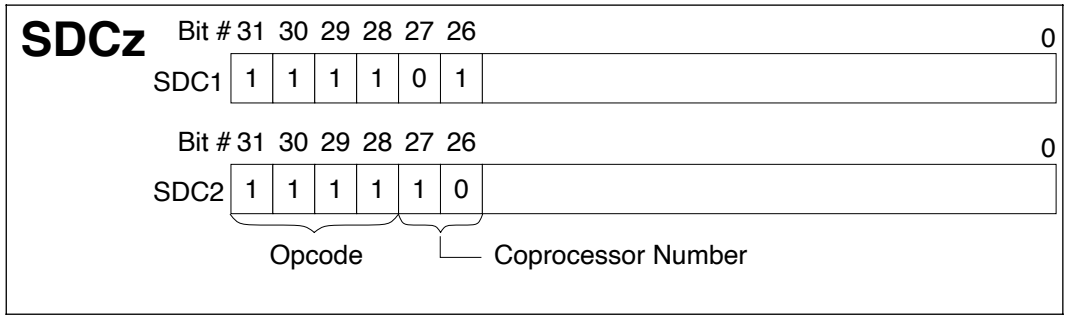
Operation:

```
32   T:  vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
         (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
         data ← GPR(rt),
         StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)

64   T:  vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
         (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
         data ← GPR(rt),
         StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
```

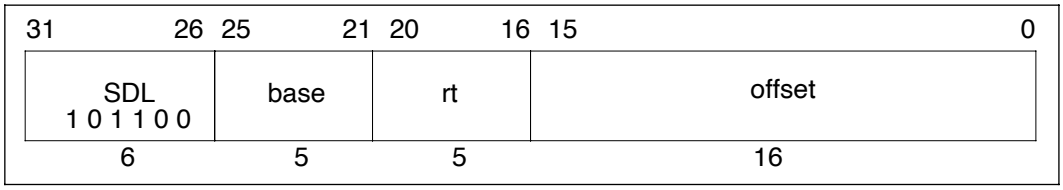
Exceptions:

TLB miss exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception
Coprocessor unusable exception

Opcode Bit Encoding:



SDL Store Doubleword Left SDL



Format:

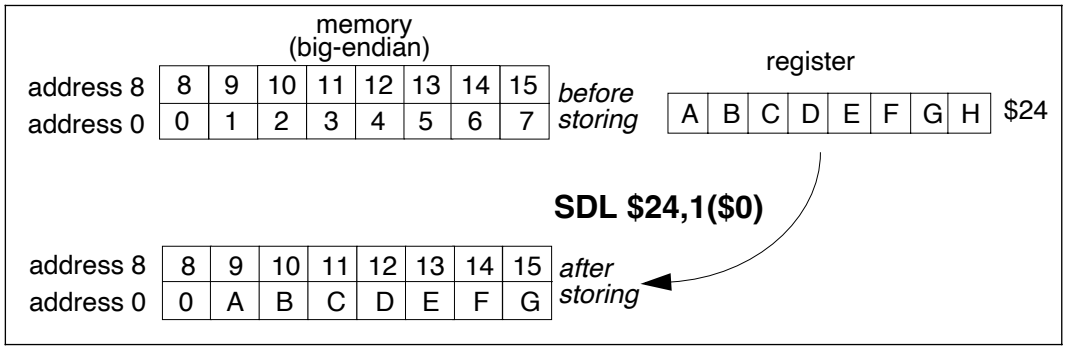
SDL rt, offset(base)

Description:

This instruction is used in combination with the SDR instruction to store the doubleword data in the register to the doubleword in the memory that is not at the doubleword boundary. The SDL instruction stores the high-order portion of the data to the memory, while the SDR instruction stores the low-order portion.

The 16-bit offset is sign-extended and added to the contents of general purpose register base to generate a virtual address. Of the doubleword data in the memory whose most-significant byte is specified by the generated address, only the high-order portion of general purpose register *rt* is stored to the memory at the same doubleword boundary as the target address. Depending on the address specified, the number of bytes to be stored changes from 1 to 8.

In other words, first the most-significant byte position of general purpose register *rt* is stored to the bytes in the addressed memory. If there is data of the low-order byte that follows the same doubleword boundary, the operation to store this data to the next byte of the memory is repeated.



SDL Store Doubleword Left (continued) SDL

The address error exception does not occur even if the specified address is not located at the doubleword boundary. This operation is defined in the 64-bit mode and 32-bit Kernel mode. If this instruction is executed in the 32-bit User or Supervisor mode, the reserved instruction exception occurs.

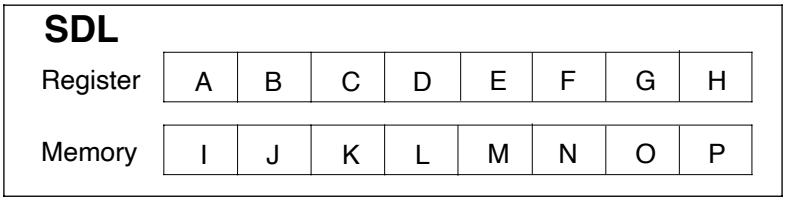
Operation:

```
64    T:   vAddr ← ((offset15)^48 || offset 15...0) + GPR[base]
               (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE –1...3 || (pAddr2...0 xor ReverseEndian^3)
           If BigEndianMem = 0 then
               pAddr ← pAddr31...3 || 0^3
           endif
           byte ← vAddr2...0 xor BigEndianCPU^3
           data ← 0^56–8*byte || GPR[rt]63...56–8*byte
           Storememory (uncached, byte, data, pAddr, vAddr, DATA)
```

Remark In the 32-bit Kernel mode, the high-order 32 bits are ignored during virtual address creation.

SDL Store Doubleword Left (continued) SDL

The relationships between the addresses given to the SDL instruction and the result (bytes for doubleword in the memory) are shown below:



$vAddr_{2...0}$	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

Remark Type: access type output to memory (Refer to **Figure 3-2 Byte Access within a Doubleword**.)

Offset: $pAddr_{2...0}$ output to memory

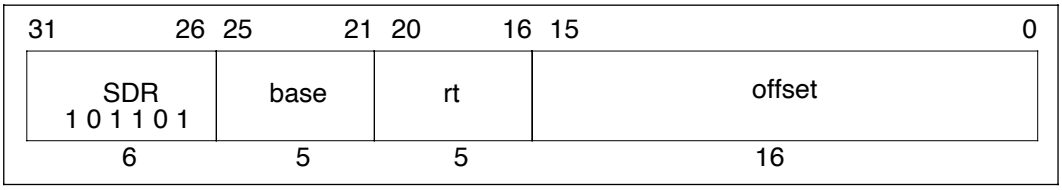
LEM Little-endian memory (BigEndianMem = 0)

BEM Big-endian memory (BigEndianMem = 1)

Exceptions:

TLB miss exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception
Reserved instruction exception (32-bit User or Supervisor mode)

SDR Store Doubleword Right SDR



Format:

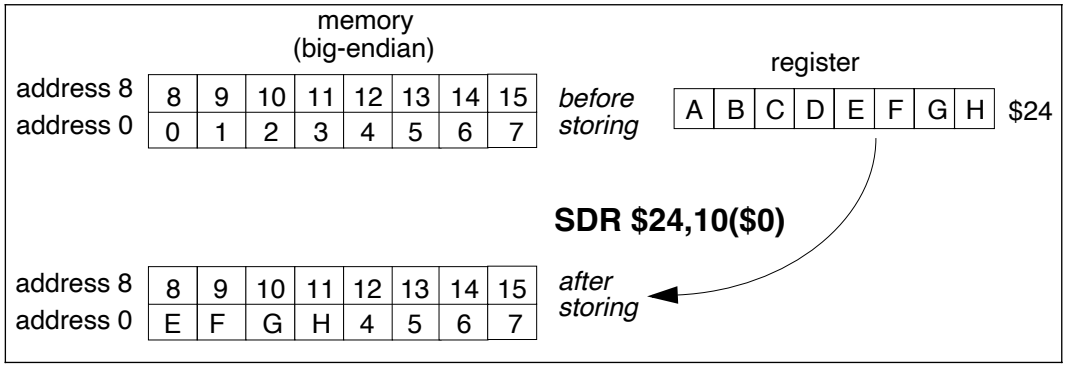
SDR rt, offset(base)

Description:

This instruction is used in combination with the SDL instruction to store the doubleword data in the register to the word data in the memory that is not at the doubleword boundary. The SDL instruction stores the high-order portion of the data to the memory, while the SDR instruction stores the low-order portion.

The 16-bit offset is sign-extended and added to the contents of general purpose register base to generate a virtual address. Of the doubleword data in the memory whose least-significant byte is specified by the generated address, only the low-order portion of general purpose register *rt* is stored to the memory at the same doubleword boundary as the target address. Depending on the address specified, the number of bytes to be stored changes from 1 to 8.

In other words, first the least-significant byte position of general purpose register *rt* is stored to the bytes in the addressed memory. If there is data of the high-order byte that follows the same doubleword boundary, the operation to store this data to the next byte of the memory is repeated.



SDR Store Doubleword Right (continued) SDR

The address error exception does not occur even if the specified address is not located at the doubleword boundary. This operation is defined in the 64-bit mode and 32-bit Kernel mode. If this instruction is executed in the 32-bit User or Supervisor mode, the reserved instruction exception occurs.

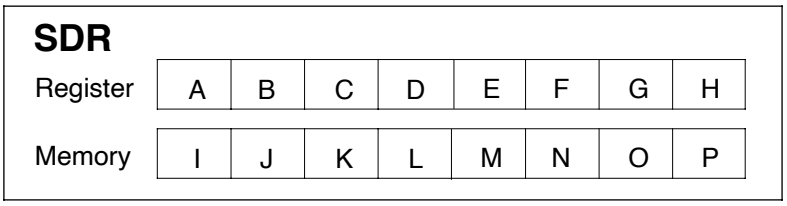
Operation:

```
64    T:   vAddr ← ((offset15)48 || offset 15...0) + GPR[base]
               (pAddr, uncached) ← AddressTranslation (vAddr, DATA
           pAddr ← pAddrPSIZE – 1...3 || (pAddr2...0 xor ReverseEndian3)
           If BigEndianMem = 0 then
               pAddr ← pAddrPSIZE – 1...3 || 03
           endif
           byte ← vAddr2...0 xor BigEndianCPU3
           data ← GPR[rt]63–8*byte || 08*byte
           StoreMemory (uncached, DOUBLEWORD-byte, data, pAddr, vAddr,
           DATA)
```

Remark In the 32-bit Kernel mode, the high-order 32 bits are ignored during virtual address creation.

SDR Store Doubleword Right (continued) SDR

The relationships between the addresses given to the SDR instruction and the result (bytes for doubleword in the memory) are shown below:



$vAddr_{2...0}$	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

Remark Type: access type output to memory (Refer to **Figure 3-2 Byte Access within a Doubleword**.)

Offset: $pAddr_{2...0}$ output to memory

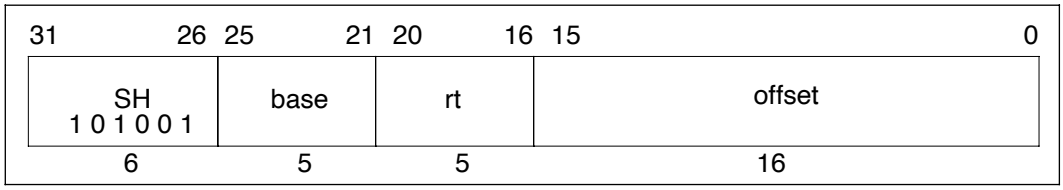
LEM Little-endian memory (BigEndianMem = 0)

BEM Big-endian memory (BigEndianMem = 1)

Exceptions:

TLB miss exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception
Reserved instruction exception (32-bit User or Supervisor mode)

SH Store Halfword SH



Format:

SH rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general purpose register *base* to form a virtual address. The least-significant halfword of register *rt* is stored in the memory specified by the address.

If the least-significant bit of the address is not zero, an address error exception occurs.

Operation:

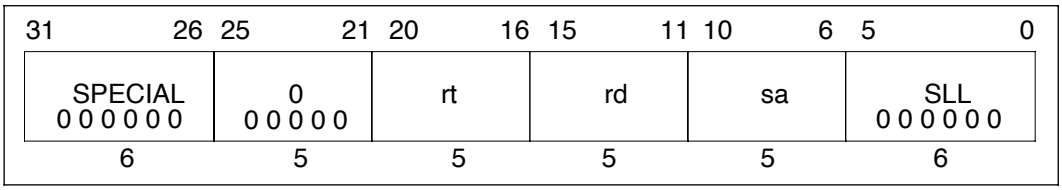
```
32    T:   vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor (ReverseEndian^2 || 0))
           byte ← vAddr2...0 xor (BigEndianCPU^2 || 0)
           data ← GPR[rt]63–8*byte...0 || 0^8*byte
           StoreMemory (uncached, HALFWORD, data, pAddr, vAddr, DATA)
64    T:   vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor (ReverseEndian^2 || 0))
           byte ← vAddr2...0 xor (BigEndianCPU^2 || 0)
           data ← GPR[rt]63–8*byte...0 || 0^8*byte
           StoreMemory (uncached, HALFWORD, data, pAddr, vAddr, DATA)
```

SH Store Halfword (Continued) SH

Exceptions:

TLB miss exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception

SLL Shift Left Logical SLL



Format:

SLL rd, rt, sa

Description:

The contents of general purpose register *rt* are shifted left by *sa* bits, inserting zeros into the low-order bits. The result is stored in general purpose register *rd*. In the 64-bit mode, the value resulting from sign-extending the shifted 32-bit value is stored as a result. If the shift value is 0, the low-order 32 bits of the 64-bit value is sign-extended. This instruction can generate a 64-bit value that sign-extends a 32-bit value.

Operation:

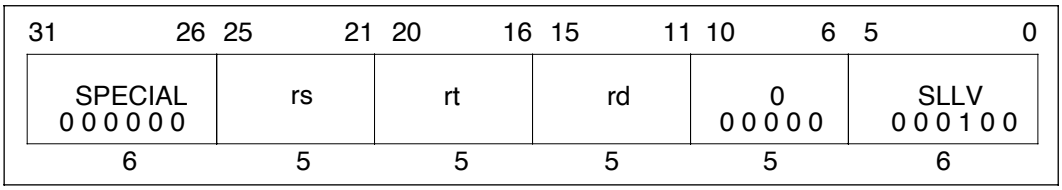
```
32   T:   GPR[rd] ← GPR[rt]31– sa...0 || 0sa
64   T:   s ← 0 || sa
          temp ← GPR[rt]31-s...0 || 0s
          GPR[rd] ← (temp31)32 || temp
```

Exceptions:

None

Caution **If the shift value of this instruction is 0, the assembler may treats this instruction as NOP. When using this instruction for sign extension, check the specifications of the assembler.**

SLLV Shift Left Logical Variable SLLV



Format:

SLLV rd, rt, rs

Description:

The contents of general purpose register *rt* are shifted left the number of bits specified by the low-order five bits of the contents of the general purpose register *rs*, inserting zeros into the low-order bits. The result is stored in general purpose register *rd*. In the 64-bit mode, the value resulting from sign-extending the shifted 32-bit value is stored as a result. If the shift value is 0, the low-order 32 bits of the 64-bit value is sign-extended. This instruction can generate a 64-bit value that sign-extends a 32-bit value.

Operation:

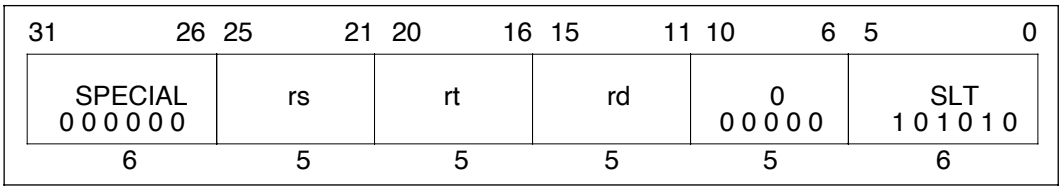
```
32    T:   s ← GPR[rs]4...0
           GPR[rd]← GPR[rt](31–s)...0 || 0^s
64    T:   s ← 0 || GPR[rs]4...0
           temp ← GPR[rt](31–s)...0 || 0^s
           GPR[rd] ← (temp31)^32 || temp
```

Exceptions:

None

Caution **If the shift value of this instruction is 0, the assembler may treats this instruction as NOP. When using this instruction for sign extension, check the specifications of the assembler.**

SLT Set On Less Than SLT



Format:

SLT rd, rs, rt

Description:

The contents of general purpose register *rt* are subtracted from the contents of general purpose register *rs*. Assuming these register contents as signed integers, if the contents of general purpose register *rs* are less than the contents of general purpose register *rt*, one is stored in the general purpose register *rd*; otherwise zero is stored in the general purpose register *rd*.

An integer overflow exception never occurs. The comparison is valid even if the subtraction used during the comparison overflows.

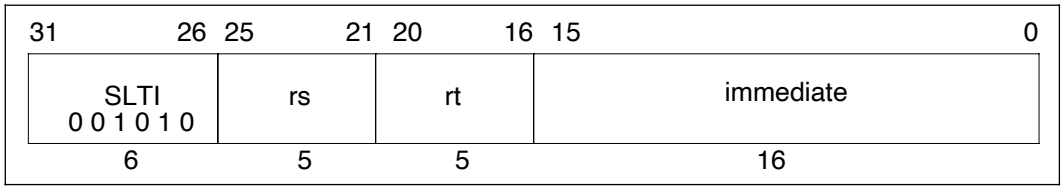
Operation:

```
32    T:    if GPR[rs] < GPR[rt] then
                    GPR[rd] ← 0^31 || 1
            else
                    GPR[rd] ← 0^32
            endif
64    T:    if GPR[rs] < GPR[rt] then
                    GPR[rd] ← 0^63 || 1
            else
                    GPR[rd] ← 0^64
            endif
```

Exceptions:

None

SLTI Set On Less Than Immediate SLTI



Format:

SLTI rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and subtracted from the contents of general purpose register *rs*. Assuming these values are signed integers, if *rs* contents are less than the sign-extended *immediate*, one is stored in the general purpose register *rt*; otherwise zero is stored in the general purpose register *rt*.

An integer overflow exception never occurs. The comparison is valid even if the subtraction overflows.

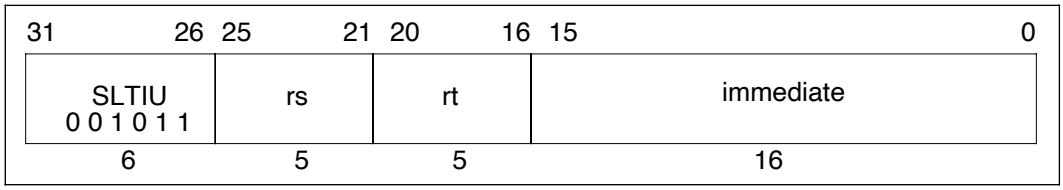
Operation:

```
32    T:    if GPR[rs] < (immediate15)16 || immediate15...0 then
                    GPR[rt] ← 031 || 1
            else
                    GPR[rt] ← 032
            endif
64    T:    if GPR[rs] < (immediate15)48 || immediate15...0 then
                    GPR[rt] ← 063 || 1
            else
                    GPR[rt] ← 064
            endif
```

Exceptions:

None

SLTIU — Set On Less Than Immediate Unsigned — SLTIU



Format:

SLTIU rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and subtracted from the contents of general purpose register *rs*. Assuming these values are unsigned integers, if *rs* contents are less than the sign-extended *immediate*, one is stored in the general purpose register *rt*; otherwise zero is stored in the general purpose register *rt*.

An integer overflow exception never occurs. The comparison is valid even if the subtraction overflows.

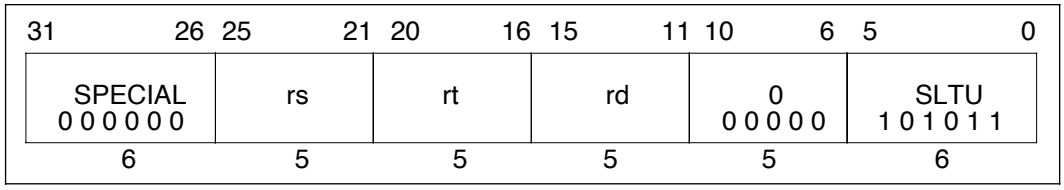
Operation:

```
32    T:    if (0 || GPR[rs]) < (immediate15)^16 || immediate15...0 then
                    GPR[rt] ← 0^31 || 1
            else
                    GPR[rt] ← 0^32
            endif
64    T:    if (0 || GPR[rs]) < (immediate15)^48 || immediate15...0 then
                    GPR[rt] ← 0^63 || 1
            else
                    GPR[rt] ← 0^64
            endif
```

Exceptions:

None

SLTU Set On Less Than Unsigned SLTU



Format:

SLTU rd, rs, rt

Description:

The contents of general purpose register *rt* are subtracted from the contents of general purpose register *rs*. Assuming these values are unsigned integers, if the contents of general purpose register *rs* are less than the contents of general purpose register *rt*, one is stored in the general purpose register *rd*; otherwise zero is stored in the general purpose register *rd*.

An integer overflow exception never occurs. The comparison is valid even if the subtraction overflows.

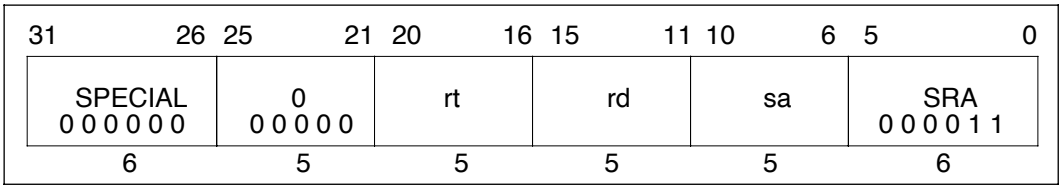
Operation:

```
32    T:    if (0 || GPR[rs]) < 0 || GPR[rt] then
                    GPR[rd] ← 0^31 || 1
            else
                    GPR[rd] ← 0^32
            endif
64    T:    if (0 || GPR[rs]) < 0 || GPR[rt] then
                    GPR[rd] ← 0^63 || 1
            else
                    GPR[rd] ← 0^64
            endif
```

Exceptions:

None

SRA Shift Right Arithmetic SRA



Format:

SRA rd, rt, sa

Description:

The contents of general purpose register *rt* are shifted right by *sa* bits, inserting signed bits into the high-order bits. The result is stored in the general purpose register *rd*. In 64-bit mode, the sign-extended 32-bit value is stored as the result.

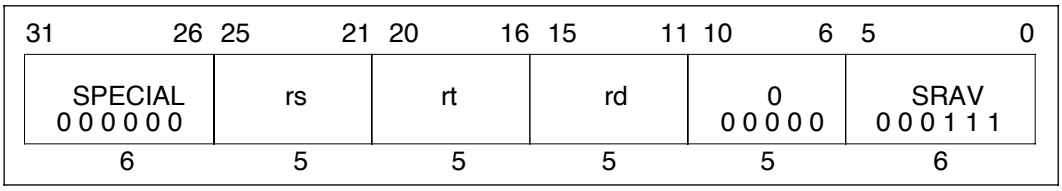
Operation:

```
32  T:  GPR[rd] ← (GPR[rt]31)^sa || GPR[rt] 31...sa
64  T:  s ← 0 || sa
        temp ← (GPR[rt]31)^s || GPR[rt] 31...s
        GPR[rd] ← (temp31)^32 || temp
```

Exceptions:

None

SRAV — Shift Right Arithmetic Variable — SRAV



Format:

SRAV rd, rt, rs

Description:

The contents of general purpose register *rt* are shifted right by the number of bits specified by the low-order five bits of general purpose register *rs*, sign-extending the high-order bits. The result is stored in the general purpose register *rd*. In 64-bit mode, the sign-extended 32-bit value is stored as the result.

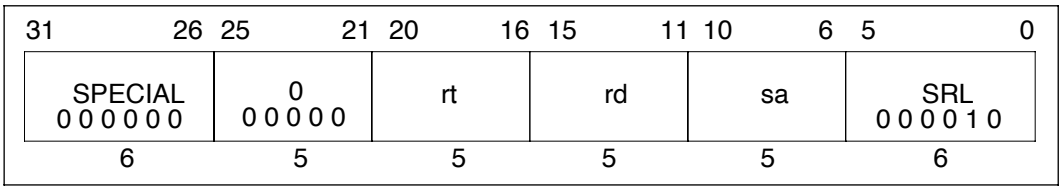
Operation:

```
32    T:   s ← GPR[rs]4...0
           GPR[rd] ← (GPR[rt]31)^s || GPR[rt]31...s

64    T:   s ← GPR[rs]4...0
           temp ← (GPR[rt]31)^s || GPR[rt]31...s
           GPR[rd] ← (temp31)^32 || temp
```

Exceptions:

None

SRL Shift Right Logical SRL



Format:

SRL rd, rt, sa

Description:

The contents of general purpose register *rt* are shifted right by *sa* bits, inserting zeros into the high-order bits. The result is stored in the general purpose register *rd*. In 64-bit mode, the sign-extended 32-bit value is stored as the result.

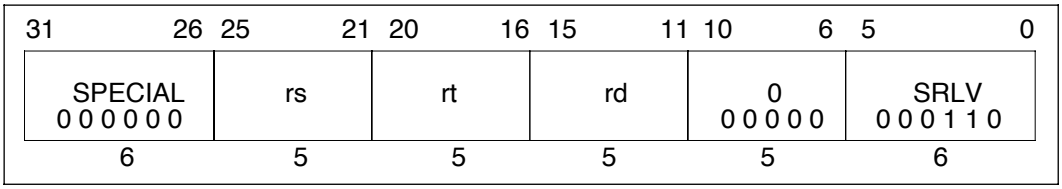
Operation:

```
32   T:   GPR[rd] ← 0^sa || GPR[rt]31...sa
64   T:   s ← 0 || sa
          temp ← 0^s || GPR[rt]31...s
          GPR[rd] ← (temp31)^32 || temp
```

Exceptions:

None

SRLV Shift Right Logical Variable SRLV



Format:

SRLV rd, rt, rs

Description:

The contents of general purpose register *rt* are shifted right by the number of bits specified by the low-order five bits of general purpose register *rs*, inserting zeros into the high-order bits. The result is stored in the general purpose register *rd*. In 64-bit mode, the sign-extended 32-bit value is stored as the result.

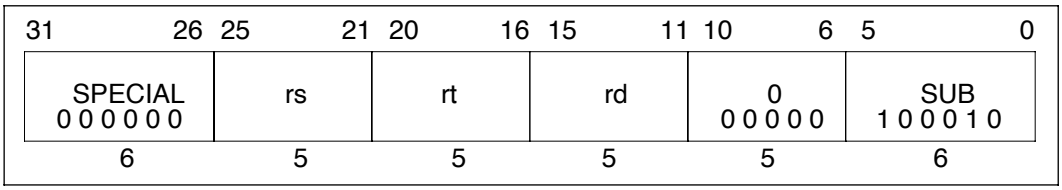
Operation:

```
32    T:   s ← GPR[rs]4...0
           GPR[rd] ← 0^s || GPR[rt]31...s

64    T:   s ← GPR[rs]4...0
           temp ← 0^s || GPR[rt]31...s
           GPR[rd] ← (temp31)^32 || temp
```

Exceptions:

None

SUB Subtract SUB



Format:

SUB rd, rs, rt

Description:

The contents of general purpose register *rt* are subtracted from the contents of general purpose register *rs*, and result is stored into general purpose register *rd*. In 64-bit mode, the sign-extended 32-bit values is stored as the result.

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

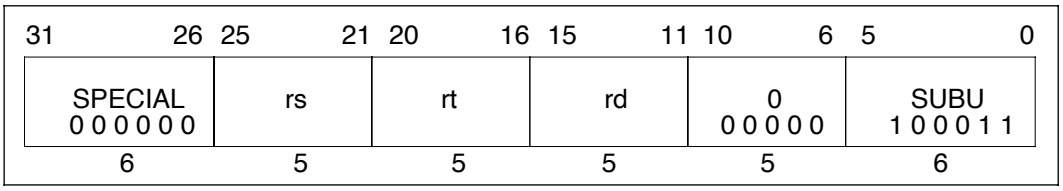
Operation:

```
32    T:    GPR[rd] ← GPR[rs] – GPR[rt]

64    T:    temp ← GPR[rs] – GPR[rt]
            GPR[rd] ← (temp31)32 || temp31...0
```

Exceptions:

Integer overflow exception

SUBU Subtract Unsigned SUBU



Format:

SUBU rd, rs, rt

Description:

The contents of general purpose register *rt* are subtracted from the contents of general purpose register *rs* and the result is stored in general purpose register *rd*. In 64-bit mode, the sign-extended 32-bit values is stored as the result.

The only difference between this instruction and the SUB instruction is that SUBU never causes an integer overflow exception.

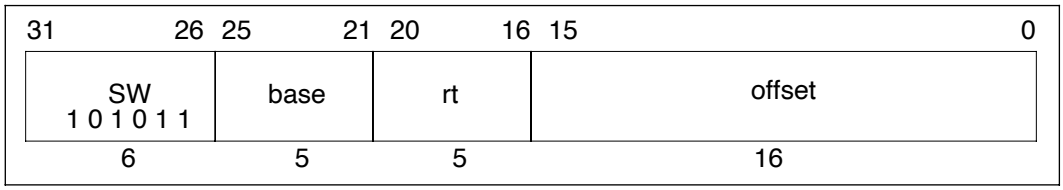
Operation:

```
32   T:   GPR[rd] ← GPR[rs] – GPR[rt]

64   T:   temp ← GPR[rs] – GPR[rt]
          GPR[rd] ← (temp31)32 || temp31...0
```

Exceptions:

None

SW Store Word SW



Format:

SW rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general purpose register *base* to form a virtual address. The contents of general purpose register *rt* are stored in the memory location specified by the address. If either of the low-order two bits of the address are not zero, an address error exception occurs.

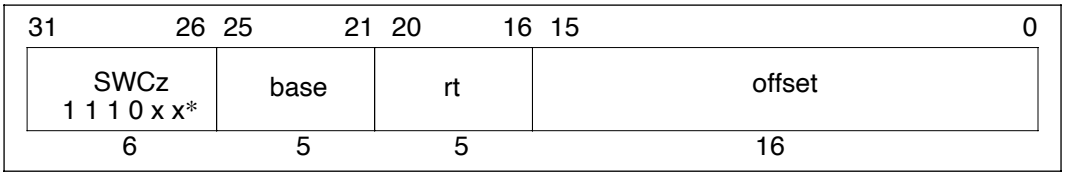
Operation:

```
32    T:   vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           data ← GPR[rt]31...0
           StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)

64    T:   vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
           (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
           data ← GPR[rt]31...0
           StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
```

Exceptions:

TLB miss exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception

SWCz Store Word From Coprocessor z SWCz



Format:

SWCz rt, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general purpose register *base* to form a virtual address. Coprocessor register *rt* of the CPz is stored in the addressed memory. The data to be stored is defined by individual coprocessor specifications. This instruction is not valid for use with CP0.

If either of the low-order two bits of the address is not zero, an address error exception occurs.

Operation:

```
32    T:  vAddr ← ((offset15)^16 || offset15...0) + GPR[base]
              (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor (ReverseEndian || 0^2))
          byte ← vAddr2...0 xor (BigEndianCPU || 0^2)
          data ← COPzSW (byte, rt)
          StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)

64    T:  vAddr ← ((offset15)^48 || offset15...0) + GPR[base]
              (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          pAddr ← pAddrPSIZE-1...3 || (pAddr2...0 xor (ReverseEndian || 0^2))
          byte ← vAddr2...0 xor (BigEndianCPU || 0^2)
          data ← COPzSW (byte,rt)
          StoreMemory (uncached, WORD, data, pAddr, vAddr DATA)
```

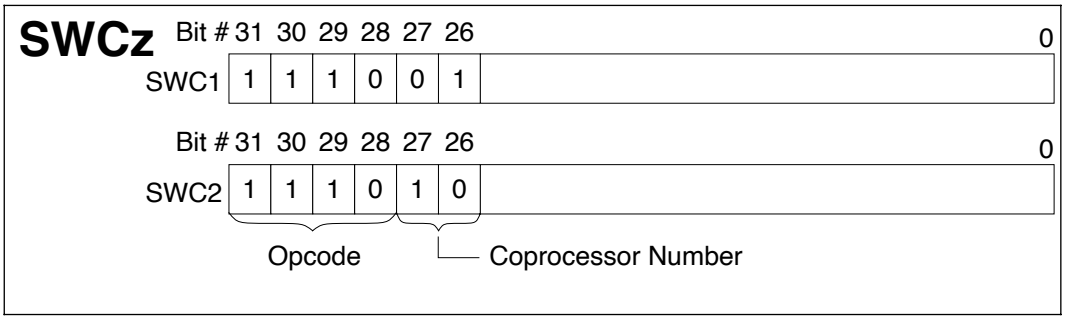
* Refer to the table **Opcode Bit Encoding** on next page, or **16.7 CPU Instruction Opcode Bit Encoding**.

SWCz Store Word From Coprocessor z (Continued) SWCz

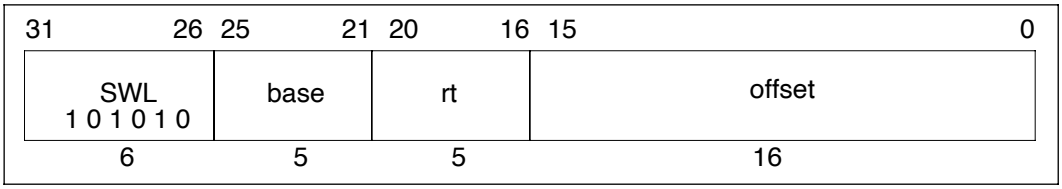
Exceptions:

TLB miss exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception
Coprocessor unusable exception

Opcode Bit Encoding:



SWL Store Word Left SWL



Format:

SWL rt, offset(base)

Description:

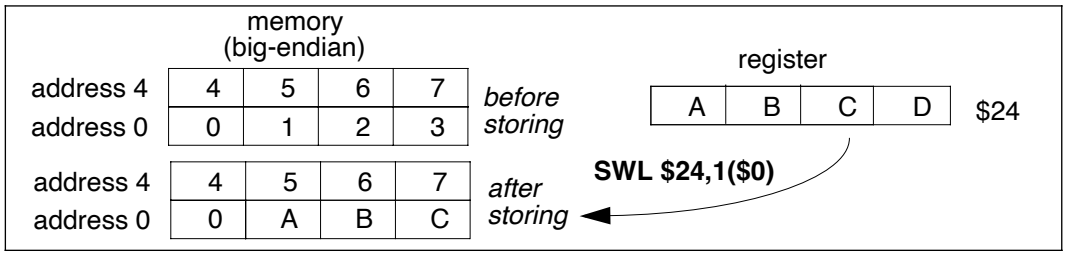
This instruction is used in combination with the SWR instruction to store the word in the register to the word in the memory that is not at the word boundary. The SWL instruction stores the high-order portion of the data to the memory, while the SWR instruction stores the low-order portion.

The 16-bit offset is sign-extended and added to the contents of general purpose register *base* to generate a virtual address. Of the word data in the memory whose most-significant byte is specified by the generated address, only the high-order portion of general purpose register *rt* is stored to the memory at the same word boundary as the target address.

Depending on the address specified, the number of bytes to be stored changes from 1 to 4.

In other words, first the most-significant byte position of general purpose register *rt* is stored to the bytes in the addressed memory. If there is data of the low-order byte that follows the same word boundary, the operation to store this data to the next byte of the memory is repeated.

No address exceptions occur due to the specified address which is not located at the word boundary.



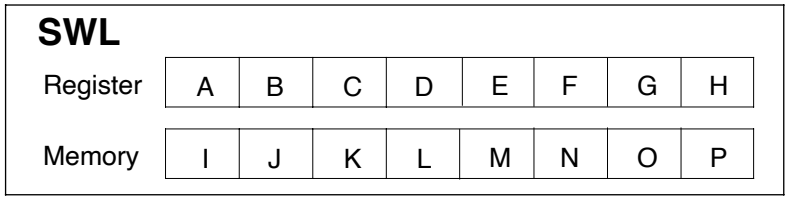
SWL — Store Word Left (Continued) — SWL

Operation:

```
32      T:  vAddr ← ((offset15)^16 || offset 15...0) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← pAddrPSIZE –1...3 || (pAddr2...0 xor ReverseEndian^3)
            If BigEndianMem = 0 then
                pAddr ← pAddr31...2 || 0^2
            endif
            byte ← vAddr1...0 xor BigEndianCPU^2
            if (vAddr2 xor BigEndianCPU) = 0 then
                data ← 0^32 || 0^24-8*byte || GPR[rt]31...24-8*byte
            else
                data ← 0^24-8*byte || GPR[rt]31...24-8*byte || 0^32
            endif
            Storememory (uncached, byte, data, pAddr, vAddr, DATA)
64      T:  vAddr ← ((offset15)^48 || offset 15...0) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← pAddr31...3 || (pAddr2...0 xor ReverseEndian^3)
            If BigEndianMem = 0 then
                pAddr ← pAddr31...2 || 0^2
            endif
            byte ← vAddr1...0 xor BigEndianCPU^2
            if (vAddr2 xor BigEndianCPU) = 0 then
                data ← 0^32 || 0^24-8*byte || GPR[rt]31...24-8*byte
            else
                data ← 0^24-8*byte || GPR[rt]31...24-8*byte || 0^32
            endif
            StoreMemory (uncached, byte, data, pAddr, vAddr, DATA)
```

SWL Store Word Left (Continued) SWL

The relationships between the contents given to the SWL instruction and the result (bytes for words in the memory) are shown below:



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

Remark Type: access type output to memory (Refer to **Figure 3-2 Byte Access within a Doubleword**.)

Offset: pAddr$_{2...0}$ output to memory

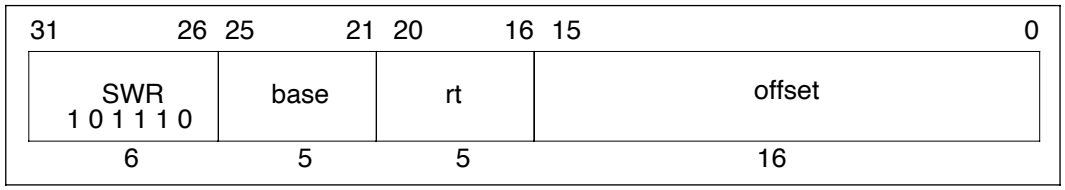
LEM Little-endian memory (BigEndianMem = 0)

BEM Big-endian memory (BigEndianMem = 1)

Exceptions:

TLB miss exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception

SWR Store Word Right SWR



Format:

SWR rt, offset(base)

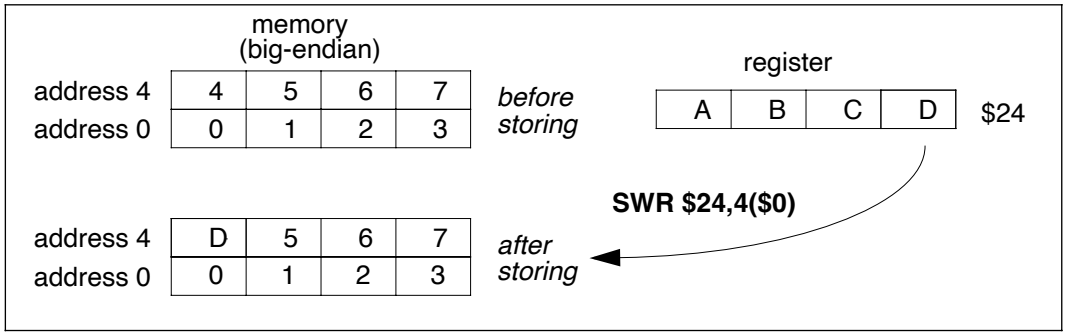
Description:

This instruction is used in combination with the SWL instruction to store word data in the register to the word data in the memory that is not at the word boundary. The SWL instruction stores the high-order portion of the data to the memory, while the SWR instruction stores the low-order portion.

The 16-bit offset is sign-extended and added to the contents of general purpose register base to generate a virtual address. Of the word data in the memory whose least-significant byte is specified by the generated address, only the low-order portion of general purpose register *rt* is stored to the memory at the same word boundary as the target address. Depending on the address specified, the number of bytes to be stored changes from 1 to 4.

In other words, first the least-significant byte position of general purpose register *rt* is stored to the bytes in the addressed memory. If there is data of the high-order byte that follows the same word boundary, the operation to store this data to the next byte of the memory is repeated.

No address exceptions occur due to the specified address which is not located at the word boundary.



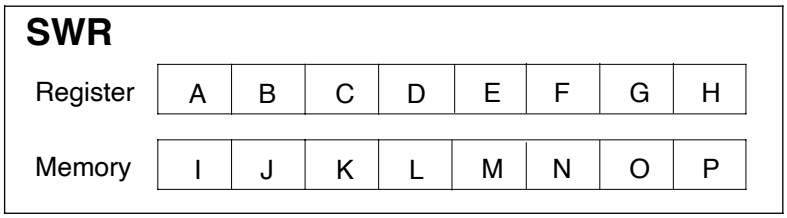
SWR Store Word Right (Continued) SWR

Operation:

```
32      T:  vAddr ← ((offset15)16 || offset 15...0) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← pAddrPSIZE – 1...3 || (pAddr2...0 xor ReverseEndian3)
            BigEndianMem = 0 then
                pAddr ← pAddr31...2 || 02
            endif
            byte ← vAddr1...0 xor BigEndianCPU2
            if (vAddr2 xor BigEndianCPU) = 0 then
                data ← 032 || GPR[rt]31-8*byte...0 || 08*byte
            else
                data ← GPR[rt]31-8*byte...0 || 08*byte || 032
            endif
            Storememory (uncached, WORD-byte, data, pAddr, vAddr, DATA)
64      T:  vAddr ← ((offset15)48 || offset 15...0) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← pAddrPSIZE – 1...3 || (pAddr2...0 xor ReverseEndian3)
            If BigEndianMem = 0 then
                pAddr ← pAddr31...2 || 02
            endif
            byte ← vAddr1...0 xor BigEndianCPU2
            if (vAddr2 xor BigEndianCPU) = 0 then
                data ← 032 || GPR[rt]31-8*byte...0 || 08*byte
            else
                data ← GPR[rt]31-8*byte...0 || 08*byte || 032
            endif
            StoreMemory (uncached, WORD-byte, data, pAddr, vAddr, DATA)
```

SWR Store Word Right (Continued) SWR

The relationships between the register contents given to the SWR instruction and the result (bytes for words in the memory) are shown below:



$vAddr_{2...0}$	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

Remark Type: access type output to memory (Refer to **Figure 3-2 Byte Access within a Doubleword**.)

Offset: $pAddr_{2...0}$ output to memory

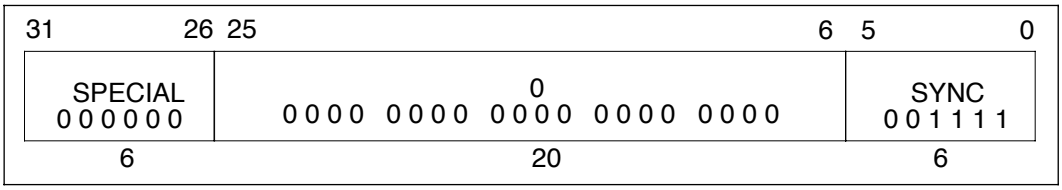
LEM Little-endian memory (BigEndianMem = 0)

BEM Big-endian memory (BigEndianMem = 1)

Exceptions:

TLB miss exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception

SYNC Synchronize SYNC



Format:

SYNC

Description:

The SYNC instruction is executed as a NOP on the V_R4300. This operation maintains compatibility with code that conforms to the V_R4400.

This instruction is defined to maintain software compatibility with the V_R4400.

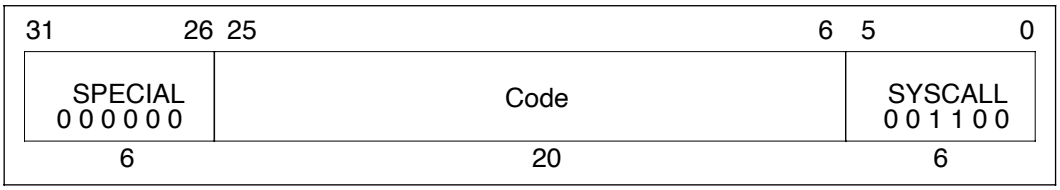
Operation:

```
32, 64   T:   SyncOperation ()
```

Exceptions:

None

SYSCALL System Call SYSCALL



Format:

SYSCALL

Description:

A system call exception occurs after this instruction is executed, unconditionally transferring control to the exception handler.

A parameter can be sent to the exception handler by using the code area. If the exception handler uses this parameter, the contents of the memory word including the instruction must be loaded as data.

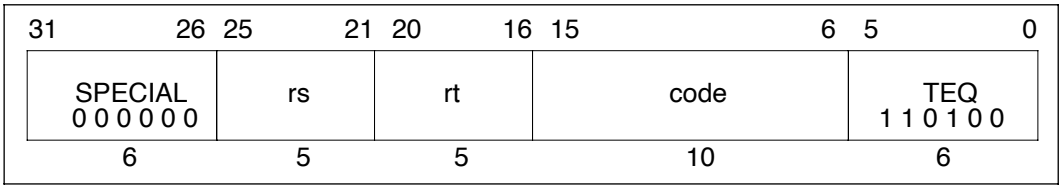
Operation:

```
32, 64   T:      SystemCallException
```

Exceptions:

System Call exception

TEQ Trap If Equal TEQ



Format:

TEQ rs, rt

Description:

The contents of general purpose register *rt* are compared with general purpose register *rs*. If the contents of general purpose register *rs* are equal to the contents of general purpose register *rt*, a trap exception occurs.

A parameter can be sent to the exception handler by using the code area. If the exception handler uses this parameter, the contents of the memory word including the instruction must be loaded as data.

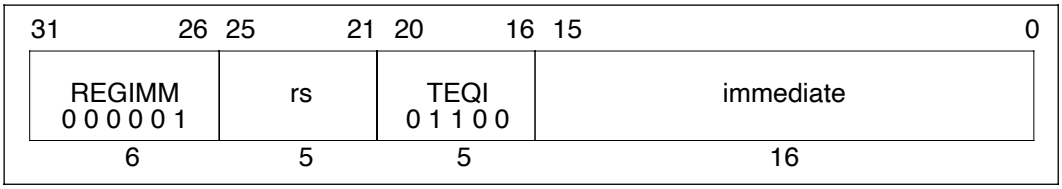
Operation:

```
32, 64      T:   if GPR[rs] = GPR[rt] then
                     TrapException
                 endif
```

Exceptions:

Trap exception

TEQI Trap If Equal Immediate TEQI



Format:

TEQI rs, immediate

Description:

The 16-bit *immediate* is sign-extended and compared with the contents of general purpose register *rs*. If the contents of general purpose register *rs* are equal to the sign-extended *immediate*, a trap exception occurs.

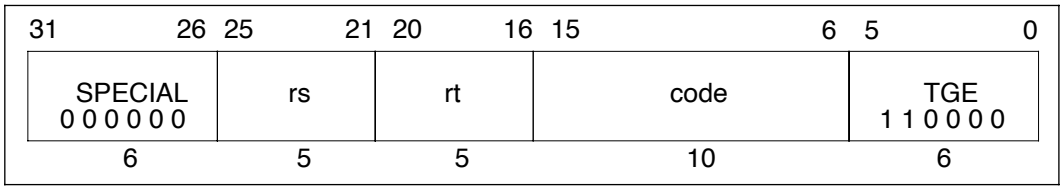
Operation:

```
32    T:    if GPR[rs] = (immediate15)16 || immediate15...0 then
              TrapException
            endif

64    T:    if GPR[rs] = (immediate15)48 || immediate15...0 then
              TrapException
            endif
```

Exceptions:

Trap exception

TGE Trap If Greater Than Or Equal TGE



Format:

TGE rs, rt

Description:

The contents of general purpose register *rt* are compared with the contents of general purpose register *rs*. Assuming both register contents are signed integers, if the contents of general purpose register *rs* are greater than or equal to the contents of general purpose register *rt*, a trap exception occurs.

A parameter can be sent to the exception handler by using the code area. If the exception handler uses this parameter, the contents of the memory word including the instruction must be loaded as data.

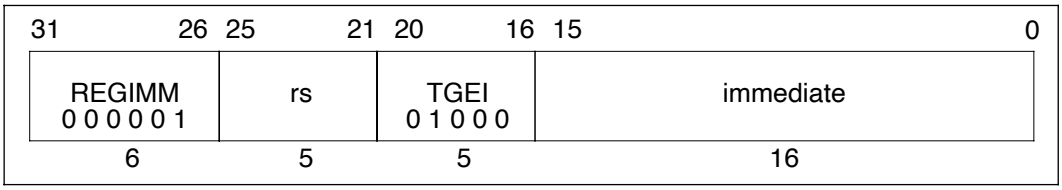
Operation:

```
32, 64  T:  if GPR[rs] ≥ GPR[rt] then
                TrapException
            endif
```

Exceptions:

Trap exception

TGEI Trap If Greater Than Or Equal Immediate TGEI



Format:

TGEI rs, immediate

Description:

The 16-bit *immediate* is sign-extended and compared with the contents of general purpose register *rs*. Assuming both values are signed integers, if the contents of general purpose register *rs* are greater than or equal to the sign-extended *immediate*, a trap exception occurs.

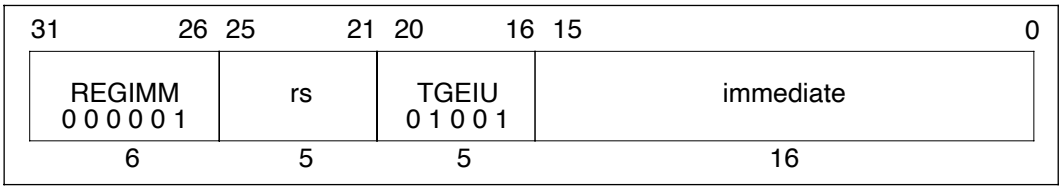
Operation:

```
32    T:  if GPR[rs] ≥ (immediate15)16 || immediate15...0 then
              TrapException
          endif
64    T:  if GPR[rs] ≥ (immediate15)48 || immediate15...0 then
              TrapException
          endif
```

Exceptions:

Trap exception

TGEIU — Trap If Greater Than Or Equal Immediate Unsigned — TGEIU



Format:

TGEIU rs, immediate

Description:

The 16-bit *immediate* is sign-extended and compared with the contents of general purpose register *rs*. Assuming both values are unsigned integers, if the contents of general purpose register *rs* are greater than or equal to the sign-extended *immediate*, a trap exception occurs.

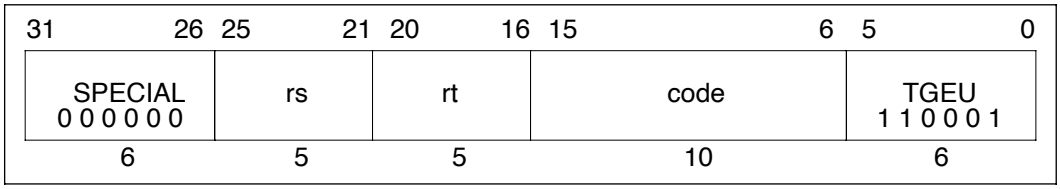
Operation:

```
32      T:  if (0 || GPR[rs]) ≥ (0 || (immediate15)16 || immediate15...0) then
                TrapException
            endif
64      T:  if (0 || GPR[rs]) ≥ (0 || (immediate15)48 || immediate15...0) then
                TrapException
            endif
```

Exceptions:

Trap exception

TGEU Trap If Greater Than Or Equal Unsigned TGEU



Format:

TGEU rs, rt

Description:

The contents of general purpose register *rt* are compared with the contents of general purpose register *rs*. Assuming both values are unsigned integers, if the contents of general purpose register *rs* are greater than or equal to the contents of general purpose register *rt*, a trap exception occurs.

A parameter can be sent to the exception handler by using the code area. If the exception handler uses this parameter, the contents of the memory word including the instruction must be loaded as data.

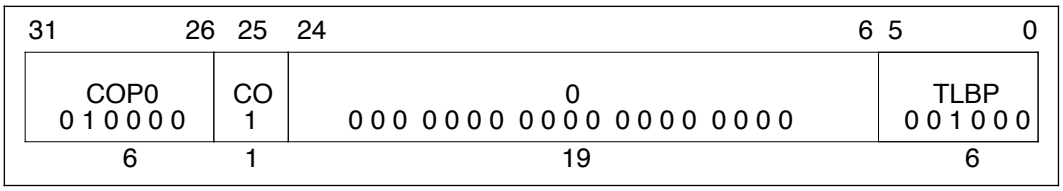
Operation:

```
32, 64    T:    if (0 || GPR[rs]) ≥ (0 || GPR[rt]) then
                    TrapException
                endif
```

Exceptions:

Trap exception

TLBP Probe TLB For Matching Entry TLBP



Format:

TLBP

Description:

Searches a TLB entry that matches with the contents of the entry *Hi* register and sets the number of that TLB entry to the index register. If a TLB entry that matches is not found, sets the most significant bit of the index register.

The architecture does not specify the operation of memory references associated with the instruction immediately after a TLBP instruction, nor is the operation specified if more than one TLB entry matches.

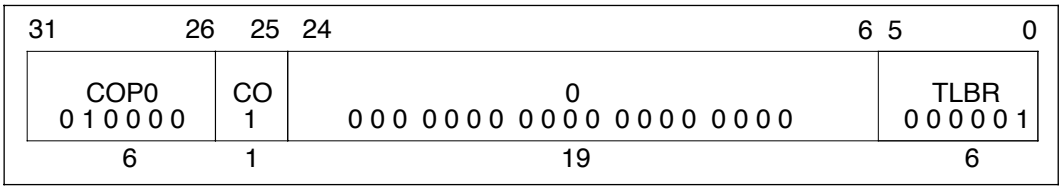
Operation:

```
32    T:   Index← 1 || 0^25 || Undefined^6
           for i in 0...TLBEntries–1
                if (TLB[i]95...77 = EntryHi31...13) and  (TLB[i]76 or
                (TLB[i]71...64 = EntryHi7...0)) then
                     Index ← 0^26 || i 5...0
                endif
           endfor
64    T:   Index← 1 || 0^25 || Undefined^6
           for i in 0...TLBEntries–1
                if (TLB[i]167...141 and not (0^15 || TLB[i]216...205))
                 = (EntryHi39...13 and not (0^15 || TLB[i]216...205)) and
                (TLB[i]140 or (TLB[i]135...128 = EntryHi7...0)) then
                     Index ← 0^26 || i 5...0
                endif
           endfor
```

Exceptions:

Coprocessor unusable exception

TLBR Read Indexed TLB Entry TLBR



Format:

TLBR

Description:

The *EntryHi* and *EntryLo* registers are loaded with the contents of the TLB entry pointed at by the contents of the *Index* register. The *G* bit (which controls ASID matching) read from the TLB is written into both of the *EntryLo0* and *EntryLo1* registers.

The operation is invalid if the contents of the *Index* register are greater than the number of TLB entries in the processor.

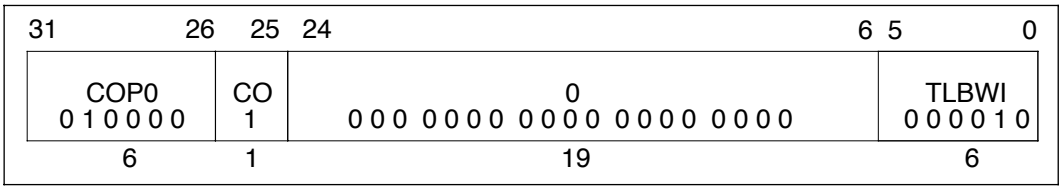
Operation:

```
32    T:  PageMask ← TLB[Index5...0]127...96
          EntryHi ← TLB[Index5...0]95...64 and not TLB[Index5...0]127...96
          EntryLo1 ←TLB[Index5...0]63...33|| TLB[Index5...0]76
          EntryLo0 ← TLB[Index5...0]31...1|| TLB[Index5...0]76
64    T:  PageMask ← TLB[Index5...0]255...192
          EntryHi ← TLB[Index5...0]191...128 and not TLB[Index5...0]255...192
          EntryLo1 ←TLB[Index5...0]127...65 || TLB[Index5...0]140
          EntryLo0 ← TLB[Index5...0]63...1 || TLB[Index5...0]140
```

Exceptions:

Coprocessor unusable exception

TLBWI Write Indexed TLB Entry TLBWI



Format:

TLBWI

Description:

The TLB entry pointed at by the *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 if the contents of the *Index* register are greater than the number of TLB entries in the processor.

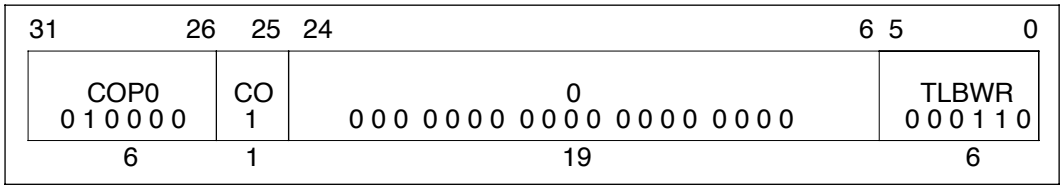
Operation:

```
32, 64   T:   TLB[Index5...0] ←
                  PageMask || (EntryHi and not PageMask)  || EntryLo1 || EntryLo0
```

Exceptions:

Coprocessor unusable exception

TLBWR Write Random TLB Entry TLBWR



Format:

TLBWR

Description:

The TLB entry pointed at by the *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.

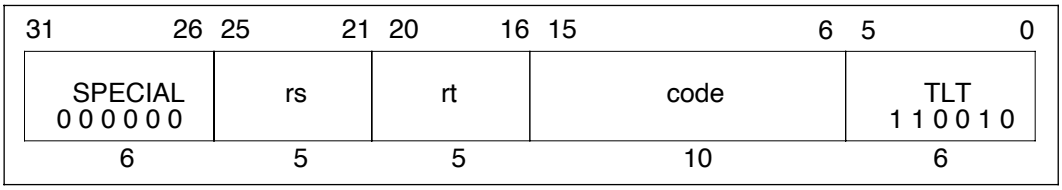
Operation:

```
32, 64   T:   TLB[Random5...0] ←
                  PageMask || (EntryHi and not PageMask)  || EntryLo1 || EntryLo0
```

Exceptions:

Coprocessor unusable exception

TLT Trap If Less Than TLT



Format:

TLT rs, rt

Description:

The contents of general purpose register *rt* are compared with general purpose register *rs*. Assuming both values are signed integers, if the contents of general purpose register *rs* are less than the contents of general purpose register *rt*, a trap exception occurs.

A parameter can be sent to the exception handler by using the code area. If the exception handler uses this parameter, the contents of the memory word including the instruction must be loaded as data.

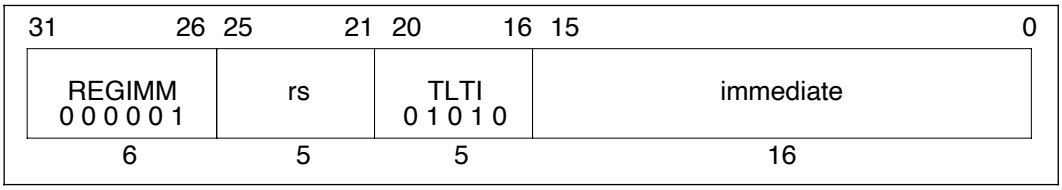
Operation:

```
32, 64  T:  if GPR[rs] < GPR[rt] then
                TrapException
            endif
```

Exceptions:

Trap exception

TLTI Trap If Less Than Immediate TLTI



Format:

TLTI rs, immediate

Description:

The 16-bit *immediate* is sign-extended and compared with the contents of general purpose register *rs*. Assuming both values are signed integers, if the contents of general purpose register *rs* are less than the sign-extended *immediate*, a trap exception occurs.

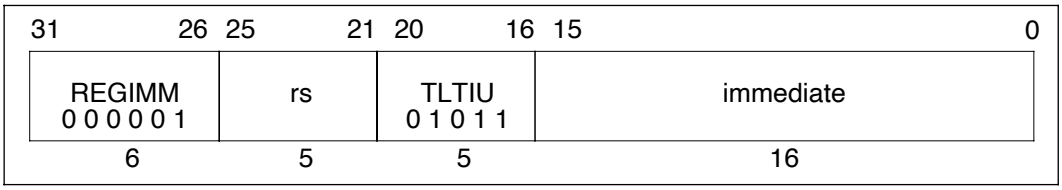
Operation:

```
32    T:  if GPR[rs] < (immediate15)16 || immediate15...0 then
              TrapException
          endif
64    T:  if GPR[rs] < (immediate15)48 || immediate15...0 then
              TrapException
          endif
```

Exceptions:

Trap exception

TLTIU Trap If Less Than Immediate Unsigned TLTIU



Format:

TLTIU rs, immediate

Description:

The 16-bit *immediate* is sign-extended and compared with the contents of general purpose register *rs*. Assuming both values are unsigned integers, if the contents of general purpose register *rs* are less than the sign-extended *immediate*, a trap exception occurs.

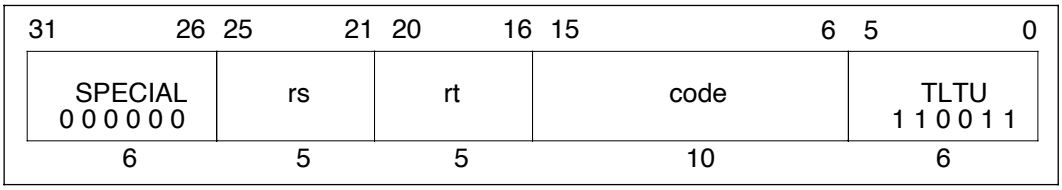
Operation:

```
32    T:   if (0 || GPR[rs]) < (0 || (immediate15)16 || immediate15...0) then
               TrapException
           endif
64    T:   if (0 || GPR[rs]) < (0 || (immediate15)48 || immediate15...0) then
               TrapException
           endif
```

Exceptions:

Trap exception

TLTU Trap If Less Than Unsigned TLTU



Format:

TLTU rs, rt

Description:

The contents of general purpose register *rt* are compared with general purpose register *rs*. Assuming both values are unsigned integers, if the contents of general purpose register *rs* are less than the contents of general purpose register *rt*, a trap exception occurs.

A parameter can be sent to the exception handler by using the code area. If the exception handler uses this parameter, the contents of the memory word including the instruction must be loaded as data.

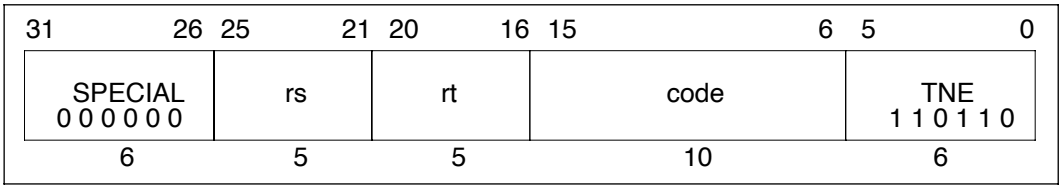
Operation:

```
32, 64T:    if (0 || GPR[rs]) < (0 || GPR[rt]) then
                TrapException
            endif
```

Exceptions:

Trap exception

TNE Trap If Not Equal TNE



Format:

TNE rs, rt

Description:

The contents of general purpose register *rt* are compared with general purpose register *rs*. If the contents of general purpose register *rs* are not equal to the contents of general purpose register *rt*, a trap exception occurs.

A parameter can be sent to the exception handler by using the code area. If the exception handler uses this parameter, the contents of the memory word including the instruction must be loaded as data.

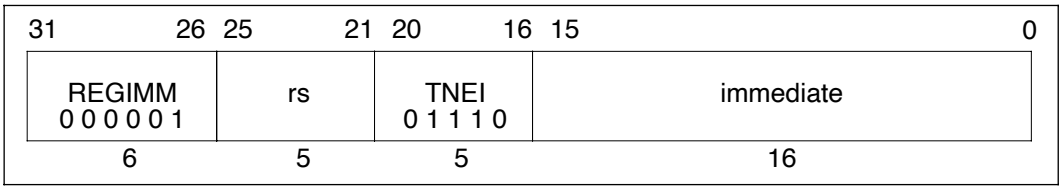
Operation:

```
32, 64T:    if GPR[rs] ≠ GPR[rt] then
                TrapException
            endif
```

Exceptions:

Trap exception

TNEI Trap If Not Equal Immediate TNEI



Format:

TNEI rs, immediate

Description:

The 16-bit *immediate* is sign-extended and compared with the contents of general purpose register *rs*. If the contents of general purpose register *rs* are not equal to the sign-extended *immediate*, a trap exception occurs.

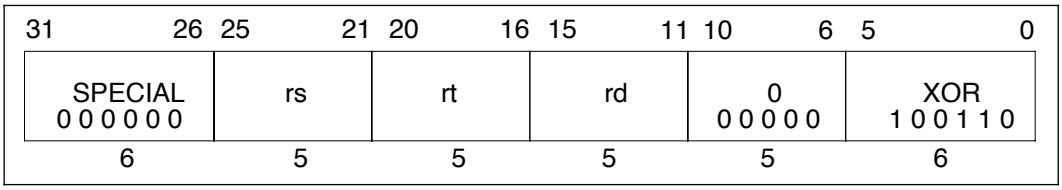
Operation:

```
32    T:   if GPR[rs] ≠ (immediate15)16 || immediate15...0 then
             TrapException
           endif
64    T:   if GPR[rs] ≠ (immediate15)48 || immediate15...0 then
             TrapException
           endif
```

Exceptions:

Trap exception

XOR Exclusive Or XOR



Format:

XOR rd, rs, rt

Description:

The contents of general purpose register *rs* and the contents of general purpose register *rt* are logical exclusive ORed bit-wise. The result is stored into general purpose register *rd*.

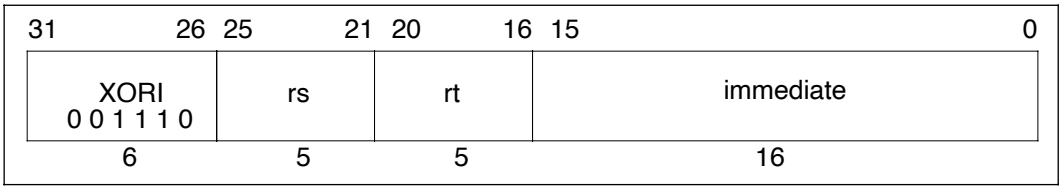
Operation:

```
32, 64      T:      GPR[rd] ← GPR[rs] xor GPR[rt]
```

Exceptions:

None

XORI Exclusive Or Immediate XORI



Format:

XORI rt, rs, immediate

Description:

The 16-bit zero-extended *immediate* and the contents of general purpose register *rs* are logical exclusive ORed bit-wise.

The result is stored in general purpose register *rt*.

Operation:

```
32    T:    GPR[rt] ← GPR[rs] xor (0^16 || immediate)
64    T:    GPR[rt] ← GPR[rs] xor (0^48 || immediate)
```

Exceptions:

None

16.7 CPU Instruction Opcode Bit Encoding

Figure 16-1 lists the V_R4300 Opcode Bit Encoding.

Opcode

31...29 \ 28...26	0	1	2	3	4	5	6	7
0	SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	COP0	COP1	COP2	*	BEQL	BNEL	BLEZL	BGTZL
3	DADDIε	DADDIUε	LDLε	LDRε	*	*	*	*
4	LB	LH	LWL	LW	LBU	LHU	LWR	LWUε
5	SB	SH	SWL	SW	SDLε	SDRε	SWR	CACHE δ
6	LL	LWC1	LWC2	*	LLDε	LDC1	LDC2	LDε
7	SC	SWC1	SWC2	*	SCDε	SDC1	SDC2	SDε

SPECIAL function

5...3 \ 2...0	0	1	2	3	4	5	6	7
0	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
1	JR	JALR	*	*	SYSCALL	BREAK	*	SYNC
2	MFHI	MTHI	MFLO	MTLO	DSLLVε	*	DSRLVε	DSRAVε
3	MULT	MULTU	DIV	DIVU	DMULTε	DMULTUε	DDIVε	DDIVUε
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	*	*	SLT	SLTU	DADDε	DADDUε	DSUBε	DSUBUε
6	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	DSLLε	*	DSRLε	DSRAε	DSLL32ε	*	DSRL32ε	DSRA32ε

REGIMM rt

20...19 \ 18...16	0	1	2	3	4	5	6	7
0	BLTZ	BGEZ	BLTZL	BGEZL	*	*	*	*
1	TGEI	TGEIU	TLTI	TLTIU	TEQI	*	TNEI	*
2	BLTZAL	BGEZAL	BLTZALL	BGEZALL	*	*	*	*
3	*	*	*	*	*	*	*	*

COPz rs

25...24 \ 23...21	0	1	2	3	4	5	6	7
0	MF	DMFε	CF	γ	MT	DMTε	CT	γ
1	BC	γ	γ	γ	γ	γ	γ	γ
2	CO							
3								

Figure 16-1 V_R4300 Opcode Bit Encoding (1/2)

COPz rt

20...19 \ 18...16	0	1	2	3	4	5	6	7
0	BCF	BCT	BCFL	BCTL	γ	γ	γ	γ
1	γ	γ	γ	γ	γ	γ	γ	γ
2	γ	γ	γ	γ	γ	γ	γ	γ
3	γ	γ	γ	γ	γ	γ	γ	γ

CP0 Function

5 ... 3 \ 2 ... 0	0	1	2	3	4	5	6	7
0	φ	TLBR	TLBWI	φ	φ	φ	TLBWR	φ
1	TLBP	φ	φ	φ	φ	φ	φ	φ
2	ξ	φ	φ	φ	φ	φ	φ	φ
3	ERET χ	φ	φ	φ	φ	φ	φ	φ
0	φ	φ	φ	φ	φ	φ	φ	φ
1	φ	φ	φ	φ	φ	φ	φ	φ
2	φ	φ	φ	φ	φ	φ	φ	φ
3	φ	φ	φ	φ	φ	φ	φ	φ

Figure 16-1 V_R4300 Opcode Bit Encoding (2/2)

Key:

*	If the operation code marked with an asterisk is executed with the current V_R4300, the reserved instruction exception occurs. This code is reserved for future expansion.
γ	Operation codes marked with a gamma cause a reserved instruction exception. They are reserved for future expansion.
δ	Operation codes marked with a delta are valid only for V_R4000 processors with CP0 enabled, and cause a reserved instruction exception on other processors.
φ	Operation codes marked with a phi are invalid but do not cause reserved instruction exceptions in V_R4300 operation.
ξ	Operation codes marked with a xi cause a reserved instruction exception on only V_R4300 processors.
χ	Operation codes marked with a chi are valid only on V_R4000 series processors.
ε	The operation code marked with an epsilon is valid in the 64-bit mode and 32-bit Kernel mode. In the 32-bit User or Supervisor mode, this code generates the reserved instruction exception.

[MEMO]

FPU Instruction Set Details

17

This chapter provides a detailed description of each floating-point unit (FPU) instruction in alphabetical order.

17.1 Instruction Formats

There are three basic instruction format types:

- I-Type, or Immediate format, which include load and store instructions
- R-Type, or Register format, which include the two- and three-register floating-point instructions
- Other, which includes Branch, and Transfer to and from instructions

The instruction description subsections that follow show how these three basic instruction formats are used by:

- Load and store instructions
- Transfer instructions
- Floating-Point arithmetic instructions
- Floating-Point branch instructions

Floating-point instructions are mapped onto the MIPS coprocessor instructions, defining coprocessor unit number one (CP1) as the floating-point unit.

Each operation is valid only for certain formats. Implementations may support some of these formats and operations through emulation, but they only need to support combinations that are valid (marked *V* in Table 17-1). Combinations marked *R* in Figure 17-1 are not currently specified by this architecture, and cause an unimplemented instruction exception. They will be available for future extensions of the architecture.

Table 17-1 Valid FPU Instruction Formats

Operation	Source Format			
	Single	**Double**	**Word**	**Longword**
ADD	V	V	R	R
SUB	V	V	R	R
MUL	V	V	R	R
DIV	V	V	R	R
SQRT	V	V	R	R
ABS	V	V	R	R
MOV	V	V		
NEG	V	V	R	R
TRUNC.L	V	V		
ROUND.L	V	V		
CEIL.L	V	V		
FLOOR.L	V	V		
TRUNC.W	V	V		
ROUND.W	V	V		
CEIL.W	V	V		
FLOOR.W	V	V		
CVT.S		V	V	V
CVT.D	V		V	V
CVT.W	V	V		
CVT.L	V	V		
C	V	V	R	R

The FPU branch instruction can be used with the logic of the condition reversed. To compare all the 32 conditions, therefore, comparison need only be performed 16 times, as shown in Table 17-2.

Table 17-2 Logical Reverse of Predicates by Condition True/False

Condition			Relations				Invalid Operation Exception If Unordered
Mnemonic		Code	Greater Than	Less Than	Equal	Unordered	
True	False						
F	T	0	F	F	F	F	No
UN	OR	1	F	F	F	T	No
EQ	NEQ	2	F	F	T	F	No
UEQ	OGL	3	F	F	T	T	No
OLT	UGE	4	F	T	F	F	No
ULT	OGE	5	F	T	F	T	No
OLE	UGT	6	F	T	T	F	No
ULE	OGT	7	F	T	T	T	No
SF	ST	8	F	F	F	F	Yes
NGLE	GLE	9	F	F	F	T	Yes
SEQ	SNE	10	F	F	T	F	Yes
NGL	GL	11	F	F	T	T	Yes
LT	NLT	12	F	T	F	F	Yes
NGE	GE	13	F	T	F	T	Yes
LE	NLE	14	F	T	T	F	Yes
NGT	GT	15	F	T	T	T	Yes

Remark F: False
T: True

Floating-Point Loads, Stores, and Transfers

All movement of data between the floating-point unit (FPU) and memory is accomplished by unit load and store instructions, which reference the floating-point unit General Purpose registers. These instructions are unformatted; no format conversions are performed and, therefore, no floating-point exceptions can occur due to these instructions.

Data may also be directly moved between the floating-point unit and the processor by *move to coprocessor* (MTC) and *move from coprocessor* (MFC) instructions. Like the floating-point load and store instructions, these instructions perform no format conversions and never cause floating-point exceptions.

In addition, two floating-point control registers can be used as the FPU registers. These registers can support only the CTC1 and CFC1 instructions.

Floating-Point Operations

The floating-point unit instruction set includes:

- floating-point add
- floating-point subtract
- floating-point multiply
- floating-point divide
- floating-point square root
- convert between fixed-point and floating-point formats
- convert between floating-point formats
- floating-point compare

These operations satisfy the requirements of IEEE Standard 754 requirements for accuracy. Specifically, these operations obtain a result which is identical to an infinite-precision result rounded to the specified format, using the current rounding mode.

Instructions must specify the format of their operands. Except for conversion functions, mixed-format operations cannot be performed.

17.2 Instruction Notation Conventions

In this chapter, all variable subfields in an instruction format (such as *fs, ft, immediate*, and so on) are shown in lowercase. Instruction names (such as ADD, SUB, and so on) are shown in uppercase.

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 lowercase, since it refers to a variable subfield.

In some instructions, the instruction subfields *op* and *function* have fixed 6-bit values. These instructions use uppercase mnemonic. For instance, in the floating-point ADD instruction we use *op* = COP1 and *function* = FADD. In other cases, a single field has both fixed and variable subfields, so the name contains both uppercase and lowercase characters. The actual code of all the mnemonics and the codes in the function fields are indicated in **17.6 FPU Instruction Opcode Bit Encoding**. The operation executed by each instruction by using representation in a high-level language is explained in the description of the operation of each instruction. For the meanings of the special symbols in the description, refer to **Table 16-1 CPU Instruction Operation Notations**.

Instruction Notation Examples

The following examples illustrate the application of some of the instruction notations:

Example #1:

$GPR[rt] \leftarrow immediate \parallel 0^{16}$

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

Example #2:

$(immediate_{15})^{16} \parallel immediate_{15...0}$

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.

Example #3:

$CPR[1, ft] \leftarrow data$

Data is assigned to general purpose register *ft* of CP1, in other words *Floating-Point General Purpose* register *FGR*.

17.3 Load and Store Instructions

In the V_R4300 implementation, the instruction immediately following a load may use the contents of the register being loaded. In such cases, the hardware *interlocks*, by the number of cycles required for reading, so scheduling load delay slots is still desirable, although not required for functional code when performance is regarded as the most significant factor, or compatibility with the V_R3000 series is required.

The operation of the load and store instructions is dependent on the width of the *FGR*s.

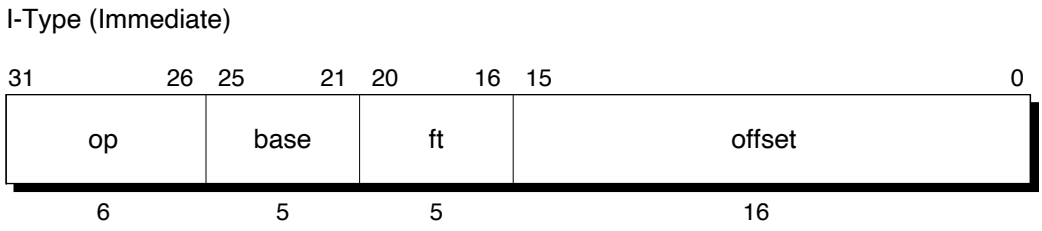
- When the *FR* bit in the *Status* register equals zero, the *Floating-Point general purpose register*s (*FGR*s) are 32-bits wide.
  To retain single-precision floating-point format data, sixteen even number registers out of thirty-two *FGR*s can be accessed.
  To retain double-precision floating-point format data, even number registers are used for low-order bits of data, and odd number registers for high-order bits.
  The registers are used as even-odd pairs, and can retain sixteen double-precision format data.
- When the *FR* bit in the *Status* register equals one, the *Floating-Point general purpose register*s (*FGR*s) are 64-bits wide.
  To retain single-precision floating-point format data, low-order bits of thirty-two *FGRs* are used.
  To retain double-precision floating-point format data, thirty-two *FGRs* are used.

In the load and store operation descriptions, the functions listed in Table 17-3 are used to summarize the handling of virtual addresses and physical memory.

Table 17-3 Load and Store Instructions Common Functions

Function	Meaning
AddressTranslation	Uses the TLB to find the physical address given by the virtual address. The function fails and a TLB miss exception occurs if the required translation is not present in the TLB.
LoadMemory	Searches cache and main memory to find contents of specified physical address at specified data length (doubleword or word), and loads contents. If cache is enabled, contents are loaded to cache.
StoreMemory	Searches and stores cache, write buffer, and main memory to store contents of specified physical address at specified data length (doubleword or word).

Figure 17-1 shows the I-Type instruction format used by load and store instructions.



op is a 6-bit opcode
base is the 5-bit base register specifier
ft is a 5-bit source (for stores) or destination (for loads) FPU register specifier
offset is the 16-bit signed immediate offset

Figure 17-1 Load and Store Instruction Format

All coprocessor loads and stores reference data which is located at the word boundary. Thus, for word loads and stores, the access type field is always WORD, and the low-order two bits of the address must always be zero. For doubleword loads and stores, the access type field is always DOUBLEWORD, and the low-order three bits of the address must always be zero.

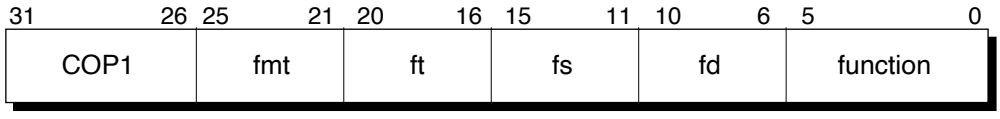
Regardless of byte-numbering order (endianness), the address specifies that byte which has the smallest byte-address in the accessed field. For a big-endian system, this is the leftmost byte; for a little-endian system, this is the rightmost byte.

17.4 Floating-Point Computational Instructions

Computational instructions include all of the floating-point computational operations performed by the FPU.

Figure 17-2 shows the R-Type instruction format used for computational operations.

R-Type (Register)



COP1	is a 6-bit opcode
fmt	is a 5-bit format specifier
fs	is a 5-bit source1 register
ft	is a 5-bit source2 register
fd	is a 5-bit destination register
function	is a 6-bit function field

Figure 17-2 Computational Instruction Format

The *function* field indicates the floating-point operation to be performed.

Each floating-point instruction can be applied to a number of operand *formats*. The operand format for an instruction is specified by the 5-bit *format* field (fmt); decoding for this field is shown in Table 17-4.

Table 17-4 Format Field Decoding

Code	Mnemonic	Size	Format
16	S	Single (32 bits)	Binary floating-point
17	D	Double (64 bits)	Binary floating-point
18	Reserved		
19	Reserved		
20	W	32 bits	Binary fixed-point
21	L	64 bits	Binary fixed-point
22–31	Reserved		

Table 17-5 lists all floating-point computational instructions.

Table 17-5 Floating-Point Computational Instructions and Operations

Code (5: 0)	Mnemonic	Operation
0	ADD	Add
1	SUB	Subtract
2	MUL	Multiply
3	DIV	Divide
4	SQRT	Square root
5	ABS	Absolute value
6	MOV	Transfer
7	NEG	Sign reverse
8	ROUND.L	Convert to 64-bit fixed-point, rounded to nearest/even
9	TRUNC.L	Convert to 64-bit fixed-point, rounded toward zero
10	CEIL.L	Convert to 64-bit fixed-point, rounded to + ∞
11	FLOOR.L	Convert to 64-bit fixed-point, rounded to – ∞
12	ROUND.W	Convert to 32-bit fixed-point, rounded to nearest/even
13	TRUNC.W	Convert to 32-bit fixed-point, rounded toward zero
14	CEIL.W	Convert to 32-bit fixed-point, rounded to + ∞
15	FLOOR.W	Convert to 32-bit fixed-point, rounded to – ∞
16–31	–	Reserved
32	CVT.S	Convert to single floating-point
33	CVT.D	Convert to double floating-point
34	–	Reserved
35	–	Reserved
36	CVT.W	Convert to 32-bit fixed-point
37	CVT.L	Convert to 64-bit fixed-point
38–47	–	Reserved
48–63	C	Floating-point compare

In the following pages, the notation *FGR* means the 32 FPU *General Purpose* registers *FGR0* through *FGR31* of the FPU, and *FPR* refers to the floating-point registers of the FPU.

An FGR (for some parts, CPR is described instead) is used for the load/store instructions, and the data transfer instruction to/from the CPU. FPR is used for the transfer instruction, arithmetic instruction, and conversion instruction in the CP1.

- When the *FR* bit in the *Status* register (26 bit) equals zero, only the even floating-point registers are valid and the 32 FPUs are 32-bit wide.
- When the *FR* bit in the *Status* register (26 bit) equals one, both odd and even FPRs can be used and the 32 FPUs are 64-bit wide.

The following routines are used in the description of the floating-point operations to retrieve the value of an FPR or to change the value of an FGR:

32 Bit Mode

```
value <-- ValueFPR(fpr, fmt)
    /* undefined for odd fpr */
    case fmt of
        S, W:
            value <-- FGR[fpr+0]
        D:
            value <-- FGR[fpr+1] || FGR[fpr+0]
    end

StoreFPR(fpr, fmt, value):
    /* undefined for odd fpr */
    case fmt of
        S, W:
            FGR[fpr+1] <-- undefined
            FGR[fpr+0] <-- value
        D:
            FGR[fpr+1] <-- value63...32
            FGR[fpr+0] <-- value31...0
    end
```

64 Bit Mode

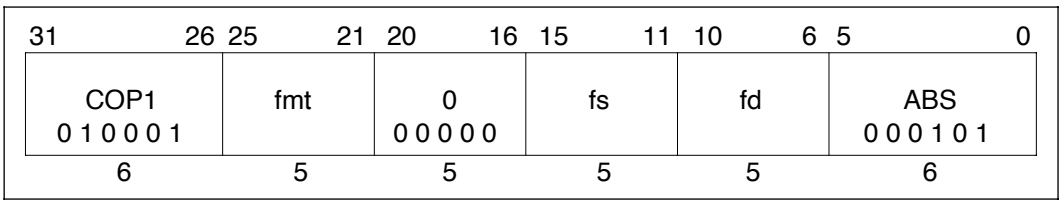
```
value <-- ValueFPR(fpr, fmt)
    case fmt of
        S, W:
            value <-- FGR[fpr]31...0
        D, L:
            value <-- FGR[fpr]
    end

StoreFPR(fpr, fmt, value):
    case fmt of
        S, W:
            FGR[fpr] <-- undefined32 || value
        D, L:
            FGR[fpr] <-- value
    end
```

17.5 FPU Instructions

This section describes in detail the floating-point (FPU) instructions.

The exceptions that may occur as a result of executing each instruction are described at the end of the description of each instruction. For the details of the exceptions and exception processing, refer to **Chapter 8 Floating-Point Exceptions**.

ABS.fmt Floating-point Absolute Value ABS.fmt



Format:

ABS.fmt fd, fs

Description:

The absolute value of the contents of floating-point register *fs* is taken and the value to floating-point register *fd* is stored. The operand is processed in the floating-point format *fmt*.

The absolute value operation is arithmetically performed. If the operand is NaN, therefore, the invalid operation exception occurs.

This instruction is valid only in the single- and double-precision floating-point formats.

If the *FR* bit of the *Status* register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as a floating-point registers. If an odd number is specified, the operation is undefined.

If the *FR* bit of the *Status* bit is 1, both the odd and even register numbers are valid.

Operation:

```
32, 64      T:      StoreFPR (fd, fmt, AbsoluteValue (ValueFPR (fs, fmt) ) )
```

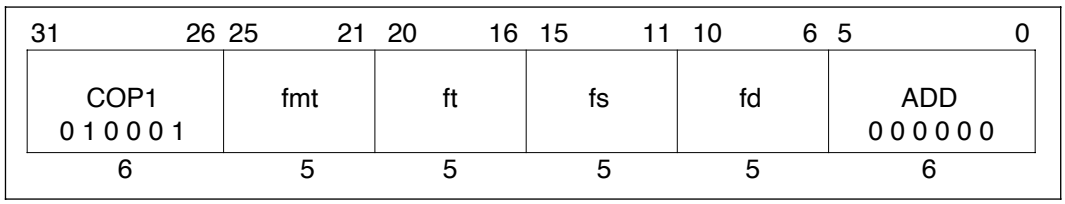
Exceptions:

Coprocessor unusable exception
Floating-point exception

Floating-Point Exceptions:

Unimplemented operation exception
Invalid operation exception

ADD.fmt Floating-point Add ADD.fmt



Format:

ADD.fmt fd, fs, ft

Description:

The contents of floating-point registers fs and ft are added, and stores the result is stored to floating-point register fd. The operand is processed in the floating-point format *fmt*. The operation is executed as if the accuracy were infinite, and the result is rounded according to the current rounding mode.

This instruction is valid only in the single- and double-precision floating-point formats.

If the *FR* bit of the status register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as a floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit of the *Status* bit is 1, both the odd and even register numbers are valid.

Operation:

```
32, 64    T:    StoreFPR (fd, fmt, ValueFPR (fs, fmt) + ValueFPR (ft, fmt) )
```

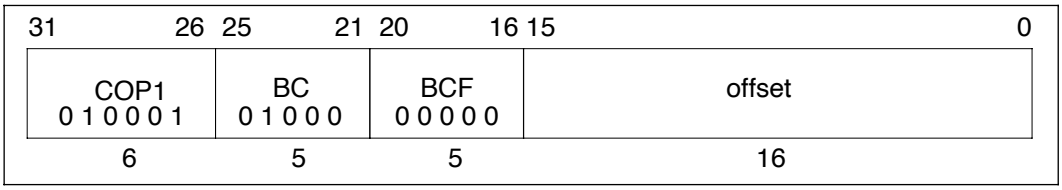
Exceptions:

Coprocessor unusable exception
Floating-point exception

Floating-Point Exceptions:

Unimplemented operation exception
Invalid operation exception
Inexact operation exception
Overflow exception
Underflow exception

BC1F — Branch On FPU False (Coprocessor 1) — BC1F



Format:

BC1F offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the CPz condition signal sampled while the instruction immediately preceding is being executed is false (0), the program branches to the branch target address, with a delay of one instruction.

Because the result of comparison is sampled while the instruction immediately preceding is executed, at least one instruction must be inserted in between the floating-point compare instruction and this instruction.

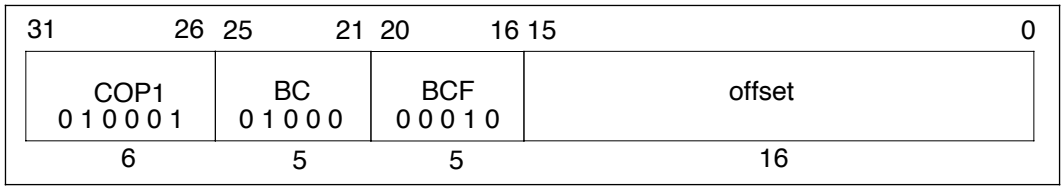
Operation:

```
32      T–1:    condition ← not COC[1]
        T:      target ← (offset15)^14 || offset || 0^2
        T+1:    if condition then
                    PC ← PC + target
                endif

64      T–1:    condition ← not COC[1]
        T:      target ← (offset15)^46 || offset || 0^2
        T+1:    if condition then
                    PC ← PC + target
                endif
```

Exceptions:

Coprocessor unusable exception

BC1FL Branch On FPU False Likely (Coprocessor 1) BC1FL



Format:

BC1FL offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the CPz condition signal sampled while the instruction immediately preceding is being executed is false (0), the program branches to the branch target address, with a delay of one instruction. If the branch is not taken, the instruction in the branch delay slot is nullified.

Because the result of comparison is sampled while the instruction immediately preceding is executed, at least one instruction must be inserted in between the floating-point compare instruction and this instruction.

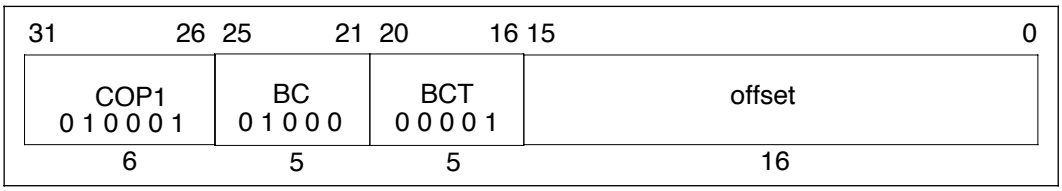
Operation:

```
32      T–1:    condition ← not COC[1]
        T:      target ← (offset15)14 || offset || 02
        T+1:    if condition then
                    PC ← PC + target
                else
                    NullifyCurrentInstruction
                endif
64      T–1:    condition ← not COC[1]
        T:      target ← (offset15)46 || offset || 02
        T+1:    if condition then
                    PC ← PC + target
                else
                    NullifyCurrentInstruction
                endif
```

Exceptions:

Coprocessor unusable exception

BC1T Branch On FPU True (Coprocessor 1) BC1T



Format:

BC1T offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the CPz condition signal sampled while the instruction immediately preceding is being executed is true (1), the program branches to the branch target address, with a delay of one instruction.

Because the result of comparison is sampled while the instruction immediately preceding is executed, at least one instruction must be inserted in between the floating-point compare instruction and this instruction.

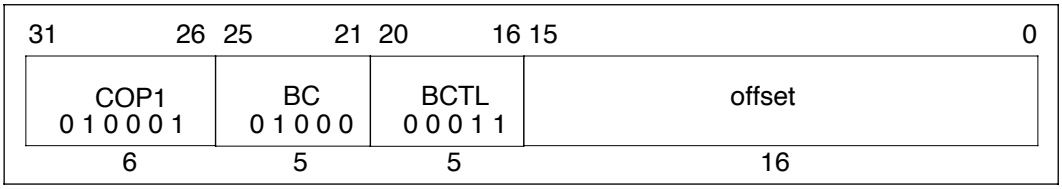
Operation:

```
32      T–1:    condition ← COC[1]
        T:      target ← (offset15)^14 || offset || 0^2
        T+1:    if condition then
                    PC ← PC + target
                endif

64      T–1:    condition ← COC[1]
        T:      target ← (offset15)^46 || offset || 0^2
        T+1:    if condition then
                    PC ← PC + target
                endif
```

Exceptions:

Coprocessor unusable exception

BC1TL Branch On FPU True Likely (Coprocessor 1) BC1TL



Format:

BC1TL offset

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the result of the last floating-point compare is true (1), the program branches to the branch target address, with a delay of one instruction. If the branch is not taken, the instruction in the branch delay slot is nullified.

Because the result of comparison is sampled while the instruction immediately preceding is executed, at least one instruction must be inserted in between the floating-point compare instruction and this instruction.

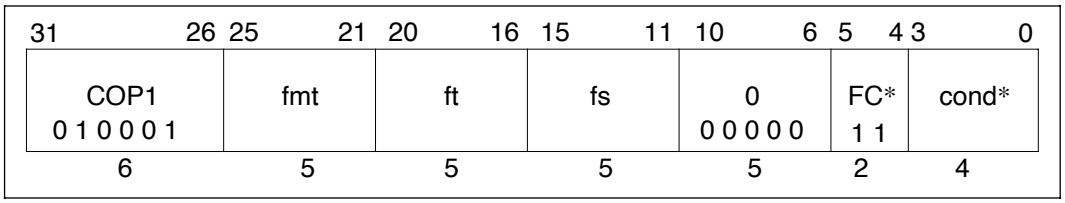
Operation:

```
32      T–1:    condition ← COC[1]
        T:      target ← (offset15)^14 || offset || 0^2
        T+1:    if condition then
                    PC ← PC + target
                else
                    NullifyCurrentInstruction
                endif
64      T–1:    condition ← COC[1]
        T:      target ← (offset15)^46 || offset || 0^2
        T+1:    if condition then
                    PC ← PC + target
                else
                    NullifyCurrentInstruction
                endif
```

Exceptions:

Coprocessor unusable exception

C.cond.fmt Floating-point Compare C.cond.fmt



Format:

C.cond.fmt fs, ft

Description:

Compares the contents of floating-point register *fs* with those of floating-point register *ft* based on compare condition cond, and sets the result to condition signal COC [1]. The operand is processed in the floating-point format *fmt*. If one of the values is NaN and if the most-significant bit of compare condition cond is set, the invalid operation exception occurs (the result of the comparison is used to test the FPU branch instruction). At least one instruction is necessary between this instruction and the FPU branch instruction.

Comparison is performed normally, and does not overflow or underflow. One of four mutually exclusive relations results, “less than”, “equal to”, “greater than”, or “cannot be compared”, occurs. If one of or both the operands are NaN, the result of the comparison is always “cannot be compared”.

During comparison, the sign of 0 is ignored (+0 = –0).

This instruction is valid only in the single- and double-precision floating-point format.

If the *FR* bit of the status register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as a floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit of the status bit is 1, both the odd and even register numbers are valid.

* See **17.6 FPU Instruction Opcode Bit Encoding**.

C.cond.fmt Floating-point Compare (continued) C.cond.fmt

Operation:

```
32, 64    T:    if NaN (ValueFPR (fs, fmt) ) or NaN (ValueFPR (ft, fmt) ) then
                    less ← false
                    equal ← false
                    unordered ← true
                    if    cond3 then
                        signal InvalidOperationException
                    endif
                else
                    less ← ValueFPR (fs, fmt) < ValueFPR (ft, fmt)
                    equal ← ValueFPR (fs, fmt) = ValueFPR (ft, fmt)
                    unordered ← false
                endif
                condition ← (cond2 and less) or (cond1 and equal) or
                            (cond0 and unordered)
                FCR[31]23  ← condition
                COC[1] ←  condition
```

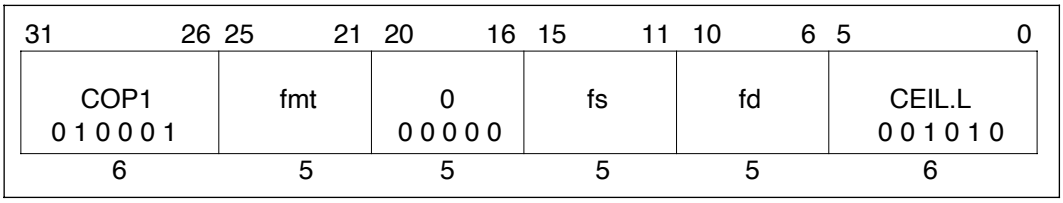
Exceptions:

Coprocessor unusable
Floating-point exception

Floating-Point Exceptions:

Unimplemented operation exception
Invalid operation exception

CEIL.L.fmt Floating-point Ceiling To Long Fixed-point Format CEIL.L.fmt



Format:

CEIL.L.fmt fd, fs

Description:

The contents of floating-point register fs are arithmetically converted into a 64-bit fixed-point format, and the result is stored to floating-point register *fd*. The source operand is processed in the floating-point format *fmt*.

The result of the conversion is rounded toward the + ∞ direction, regardless of the current rounding mode.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the *Status* register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as a floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit of the *Status* register is 1, both the odd and even register numbers are valid.

If the source operand is infinite or NaN, and if the rounded result is outside the range of $2^{63} - 1$ to -2^{63}, the invalid operation exception occurs. If the invalid operation exception is not enabled, the exception does not occur, and $2^{63} - 1$ is returned.

This operation is defined in the 64-bit mode and 32-bit Kernel mode. If this instruction is executed during 32-bit User/Supervisor mode, a reserved instruction exception occurs.

CEIL.L.fmt Floating-point Ceiling To Long Fixed-point Format (continued) CEIL.L.fmt

Operation:

```
32, 64     T:     StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt) , fmt, L) )
```

Exceptions:

Coprocessor unusable exception
Floating-point exception
Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

Floating-Point Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact operation exception
Overflow exception

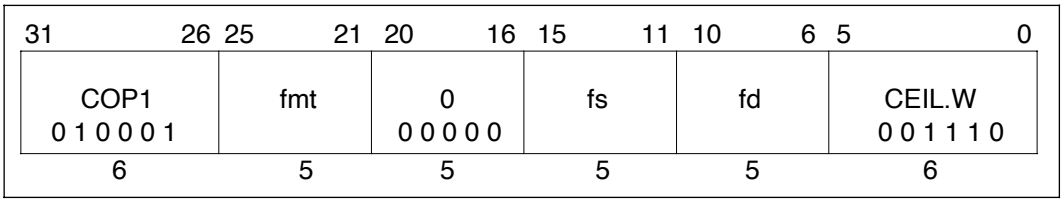
★ **Restrictions:**

An unimplemented operation exception will occur in the following cases.

- If an overflow occurs during conversion to integer format
- If the source operand is an infinite number
- If the source operand is NaN

Essentially, if any of bits 53 to 62 of the result of conversion from a floating-point format to a fixed-point format is 1, an unimplemented operation exception will occur. This includes cases when there is an overflow during conversion.

CEIL.W.fmt Floating-point Ceiling To Single Fixed-point Format CEIL.W.fmt



Format:

CEIL.W.fmt fd, fs

Description:

The contents of floating-point register fs are arithmetically converted into a 32-bit fixed-point format, and the result is stored to floating-point register *fd*. The source operand is processed in the floating-point format *fmt*.

The result of the conversion is rounded toward the +∞ direction, regardless of the current rounding mode.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the *Status* register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as a floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit of the *Status* register is 1, both the odd and even register numbers are valid.

If the source operand is infinite or NaN, and if the rounded result is outside the range of $2^{31} - 1$ to -2^{31}, the invalid operation exception occurs. If the invalid operation exception is not enabled, the exception does not occur, and $2^{31} - 1$ is returned.

CEIL.W.fmt Floating-point Ceiling To Single Fixed-point Format (continued) CEIL.W.fmt

Operation:

```
32, 64    T:    StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt) , fmt, W) )
```

Exceptions:

Coprocessor unusable exception
Floating-point exception

Floating-Point Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact operation exception
Overflow exception

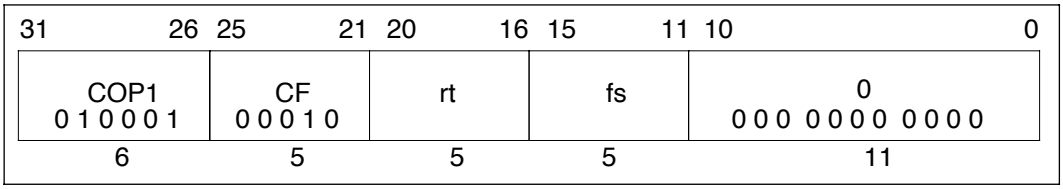
★ **Restrictions:**

An unimplemented operation exception will occur in the following cases.

- If an overflow occurs during conversion to integer format
- If the source operand is an infinite number
- If the source operand is NaN

Essentially, if any of bits 53 to 62 of the result of conversion from a floating-point format to a fixed-point format is 1, an unimplemented operation exception will occur. This includes cases when there is an overflow during conversion.

CFC1 Move Control Word From FPU (Coprocessor 1) CFC1



Format:

CFC1 rt, fs

Description:

The contents of the floating-point control register *fs* are loaded into general purpose register *rt*.

This instruction is only defined when *fs* equals 0 or 31.

The contents of general purpose register *rt* are undefined while the instruction immediately following this load instruction is being executed.

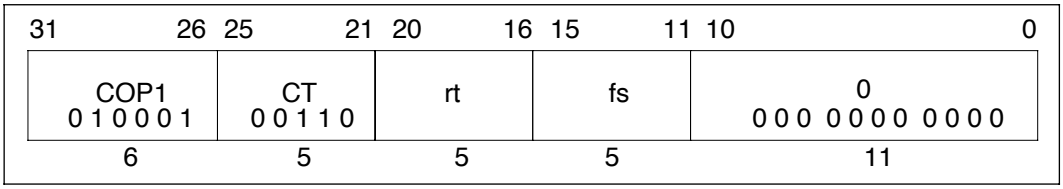
Operation:

```
32      T:    temp ← FCR[fs]
        T+1:  GPR[rt] ← temp
64      T:    temp ← FCR[fs]
        T+1:  GPR[rt] ← (temp31)32 || temp
```

Exceptions:

Coprocessor unusable exception

CTC1 Move Control Word To FPU (Coprocessor 1) CTC1



Format:

CTC1 rt, fs

Description:

The contents of general purpose register *rt* are loaded to floating-point register *fs*.

This instruction is defined if fs is 0 or 31.

If the cause bit of the floating-point control/status register (FCR31) and the corresponding enable bit are set by writing data to FCR31, the floating-point exception occurs. Write the data to the register before the exception occurs.

The contents of the floating-point control register *fs* are undefined while the instruction immediately following this instruction is executed.

Operation:

```
32      T:      temp ← GPR[rt]
        T+1:    FCR[fs] ← temp
                COC[1] ← FCR[31]23
64      T:      temp ← GPR[rt]31...0
        T+1:    FCR[fs] ← temp
                COC[1] ← FCR[31]23
```

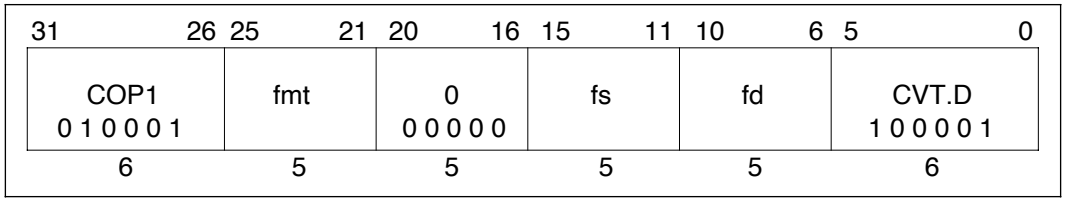
Exceptions:

Coprocessor unusable exception
Floating-point exception

Floating-Point Exceptions:

Invalid operation exception
Unimplemented operation exception
Division by zero exception
Inexact operation exception
Overflow exception
Underflow exception

CVT.D.fmt Floating-point Convert To Double Floating-point Format CVT.D.fmt



Format:

CVT.D.fmt fd, fs

Description:

The contents of floating-point register fs are arithmetically converted into a double-precision floating-point format, and the result is stored to floating-point register *fd*. The source operand is processed in the floating-point format *fmt*.

This instruction is valid only for conversion from the single-precision floating-point format, and 32-bit or 64-bit fixed floating-point format.

In the single-precision floating-point format or 32-bit fixed point format, this conversion operation is executed correctly without the accuracy becoming degraded.

If the *FR* bit of the *Status* register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as a floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit of the *Status* register is 1, both the odd and even register numbers are valid.

Operation:

```
32, 64    T:    StoreFPR (fd, D, ConvertFmt (ValueFPR (fs, fmt) , fmt, D) )
```

Exceptions:

Coprocessor unusable exception
Floating-point exception

Floating-Point Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact operation exception

CVT.D.fmt Floating-point Convert To Double Floating-point Format (continued) CVT.D.fmt

★ **Restrictions:**

An unimplemented operation exception will occur in the following cases.

- If an overflow occurs during conversion to integer format
- If the source operand is an infinite number
- If the source operand is NaN

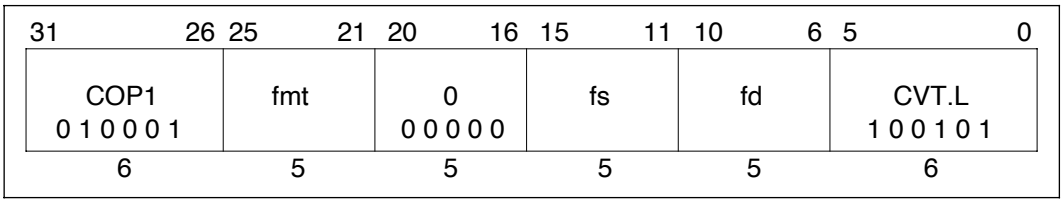
- **Conversion from floating-point format to fixed-point format**

Essentially, if any of bits 53 to 62 of the result of conversion from a floating-point format to a fixed-point format is 1, an unimplemented operation exception will occur. This includes cases when there is an overflow during conversion.

- **Conversion from fixed-point format to floating-point format**

Essentially, if 64-bit fixed-point format data in which any of bits 55 to 62 is 1 is converted to floating-point format data, an unimplemented operation exception will occur.

CVT.L.fmt Floating-point Convert To Long Fixed-point Format CVT.L.fmt



Format:

CVT.L.fmt fd, fs

Description:

The contents of floating-point register fs are arithmetically converted into a 64-bit fixed-point format, and the result is stored to floating-point register *fd*. The source operand is processed in the floating-point format *fmt*.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the *Status* register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as a floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit of the *Status* register is 1, both the odd and even register numbers are valid.

If the source operand is infinite or NaN, and if the rounded result is outside the range of $2^{63}-1$ to -2^{63}, the invalid operation exception occurs. If the invalid operation exception is not enabled, the exception does not occur, and $2^{63}-1$ is returned.

This operation is defined in the 64-bit mode and 32-bit Kernel mode. If this instruction is executed during 32-bit User/Supervisor mode, a reserved instruction exception occurs.

CVT.L.fmt Floating-point Convert To Long Fixed-point Format (continued) CVT.L.fmt

Operation:

```
64      T:   StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt) , fmt, L) )
```

Remark Same operation in the 32-bit Kernel mode.

Exceptions:

Coprocessor unusable exception
Floating-point exception
Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

Floating-Point Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact operation exception
Overflow exception

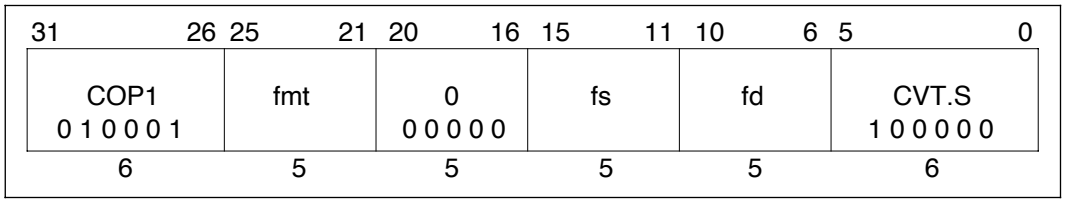
★ **Restrictions:**

An unimplemented operation exception will occur in the following cases.

- If an overflow occurs during conversion to integer format
- If the source operand is an infinite number
- If the source operand is NaN

Essentially, if any of bits 53 to 62 of the result of conversion from a floating-point format to a fixed-point format is 1, an unimplemented operation exception will occur. This includes cases when there is an overflow during conversion.

CVT.S.fmt Floating-point Convert To Single Floating-point Format CVT.S.fmt



Format:

CVT.S.fmt fd, fs

Description:

The contents of floating-point register fs are arithmetically converted into a single-precision floating-point format, and the result is stored to floating-point register *fd*. The source operand is processed in the floating-point format *fmt*. The result of the conversion is rounded according to the current rounding mode.

This instruction is valid only for conversion from the double-precision floating-point format, and 32-bit or 64-bit fixed floating-point format.

If the *FR* bit of the *Status* register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as a floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit of the *Status* register is 1, both the odd and even register numbers are valid.

Operation:

```
32, 64    T:    StoreFPR (fd, S, ConvertFmt (ValueFPR (fs, fmt) , fmt, S) )
```

Exceptions:

Coprocessor unusable exception
Floating-point exception

Floating-Point Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact operation exception
Overflow exception
Underflow exception

CVT.S.fmt Floating-point Convert To Single Floating-point Format (continued) CVT.S.fmt

★ **Restrictions:**

An unimplemented operation exception will occur in the following cases.

- If an overflow occurs during conversion to integer format
- If the source operand is an infinite number
- If the source operand is NaN

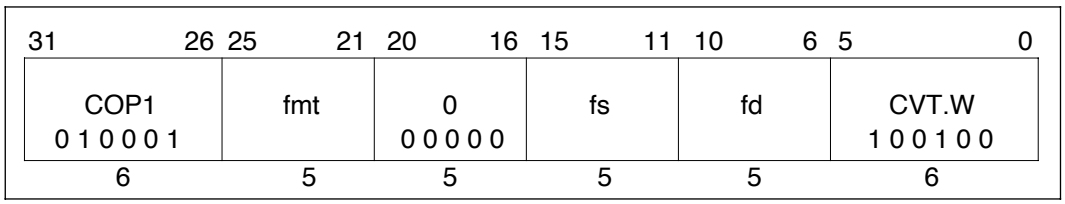
- **Conversion from floating-point format to fixed-point format**

Essentially, if any of bits 53 to 62 of the result of conversion from a floating-point format to a fixed-point format is 1, an unimplemented operation exception will occur. This includes cases when there is an overflow during conversion.

- **Conversion from fixed-point format to floating-point format**

Essentially, if 64-bit fixed-point format data in which any of bits 55 to 62 is 1 is converted to floating-point format data, an unimplemented operation exception will occur.

CVT.W.fmt Floating-point Convert To Single Fixed-point Format CVT.W.fmt



Format:

CVT.W.fmt fd, fs

Description:

The contents of floating-point register fs are arithmetically converted into a 32-bit fixed-point format, and the result is stored to floating-point register *fd*. The source operand is processed in the floating-point format *fmt*.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the *Status* register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as a floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit of the *Status* register is 1, both the odd and even register numbers are valid.

If the source operand is infinite or NaN, and if the rounded result is outside the range of $2^{31}-1$ to -2^{31}, the invalid operation exception occurs. If the invalid operation exception is not enabled, the exception does not occur, and $2^{31}-1$ is returned.

CVT.W.fmt Floating-point Convert To Single Fixed-point Format (continued) CVT.W.fmt

Operation:

```
32, 64    T:    StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt) , fmt, W) )
```

Exceptions:

Coprocessor unusable exception
Floating-point exception

Floating-Point Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact operation exception
Overflow exception

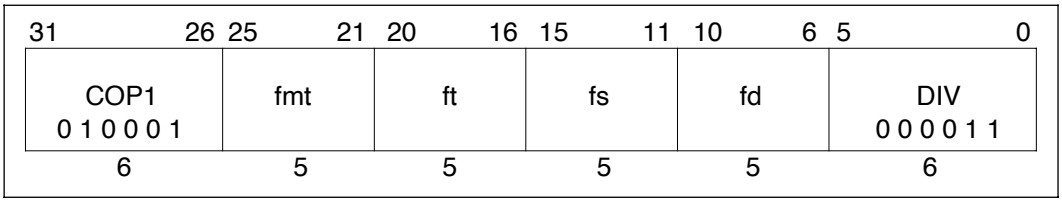
★ **Restrictions:**

An unimplemented operation exception will occur in the following cases.

- If an overflow occurs during conversion to integer format
- If the source operand is an infinite number
- If the source operand is NaN

Essentially, if any of bits 53 to 62 of the result of conversion from a floating-point format to a fixed-point format is 1, an unimplemented operation exception will occur. This includes cases when there is an overflow during conversion.

DIV.fmt Floating-point Divide DIV.fmt



Format:

DIV.fmt fd, fs, ft

Description:

The contents of floating-point register fs are divided by those of floating-point register *ft*, and the result are stored to floating-point register *rd*. The operand is processed in the floating-point format *fmt*. The operation is executed as if the accuracy were infinite, and the result is rounded according to the current rounding mode.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the *Status* register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as a floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit of the *Status* register is 1, both the odd and even register numbers are valid.

Operation:

```
32, 64      T:      StoreFPR (fd, fmt, ValueFPR (fs, fmt)/ValueFPR (ft, fmt) )
```

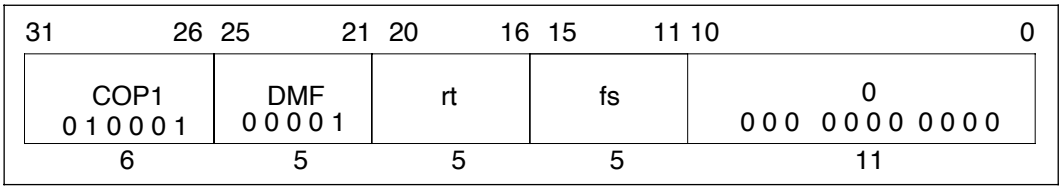
Exceptions:

Coprocessor unusable exception
Floating-point exception

Floating-Point Exceptions:

Unimplemented operation exception
Invalid operation exception
Division-by-zero exception
Inexact operation exception
Overflow exception
Underflow exception

DMFC1 Doubleword Move From FPU (Coprocessor 1) DMFC1



Format:

DMFC1 rt, fs

Description:

The contents of Floating-Point General Purpose register *fs* are stored into CPU general purpose register *rt*.

The contents of general purpose register *rt* are undefined while the instruction immediately following this instruction is being executed.

The *FR* bit of the *Status* register indicates whether all the 32 registers of the processor can be specified. If the *FR* bit is 0, and the least-significant bit of *fs* is 1, this instruction is undefined.

The operation is undefined if an odd number is specified when the *FP* bit of the status register is 0. If the *FR* bit is 1, both the odd-numbered and even-numbered registers are valid.

This operation is defined in 64-bit mode or 32-bit Kernel mode.

DMFC1 Doubleword Move From FPU (Coprocessor 1) (continued) DMFC1

Operation:

```
64      T:      if   SR26 = 1 then
                     data ← FGR [fs]
                else
                if   fs0 = 0 then
                     data ← FGR [fs + 1] || FGR [fs]
                else
                     data ← undefined64
                endif
        T+1:    GPR[rt] ← data
```

Remark Same operation in the 32-bit Kernel mode.

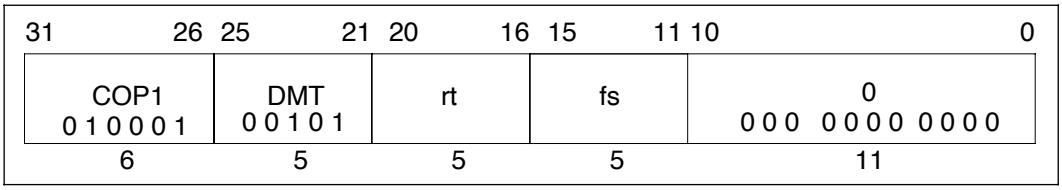
Exceptions:

Coprocessor unusable exception
Floating-point exception
Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

Floating-Point Exceptions:

Unimplemented operation exception

DMTC1 Doubleword Move To FPU (Coprocessor 1) DMTC1



Format:

DMTC1 rt, fs

Description:

The contents of general purpose register *rt* are loaded into Floating-Point General Purpose register *fs*.

The contents of *fs* are undefined while the instruction immediately following this instruction is being executed.

The *FR* bit of the *Status* register indicates whether all the 32 registers of the processor can be specified. If the *FR* bit is 0, and the least-significant bit of *fs* is 1, this instruction is undefined.

The operation is undefined if an odd number is specified when the *FR* bit of the status register is 0. If the *FR* bit is 1, both the odd-numbered and even-numbered registers are valid.

This operation is defined in 64-bit mode or 32-bit Kernel mode.

DMTC1 Doubleword Move To FPU (Coprocessor 1) (continued) DMTC1

Operation:

```
64      T:      data ← GPR[rt]
        T+1:    if  SR26 = 1 then
                   FGR [fs] ← data
                else
                if fs0 = 0 then
                   FGR [fs+1] ← data63..32
                   FGR [fs] ← data31..0
                else
                   undefined_result
                endif
```

Remark Same operation in the 32-bit Kernel mode.

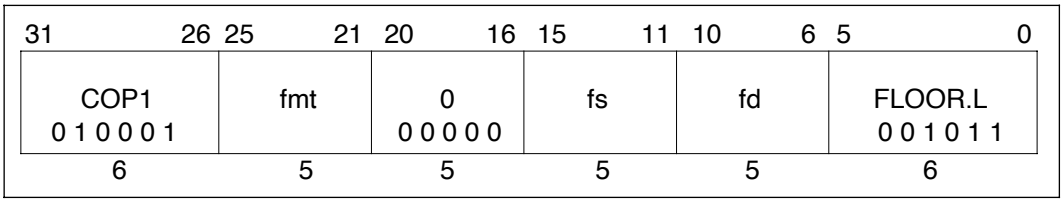
Exceptions:

Coprocessor unusable exception
Floating-point exception
Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

Floating-Point Exceptions:

Unimplemented operation exception

FLOOR.L.fmt Floating-point Floor To Long Fixed-point Format FLOOR.L.fmt



Format:

FLOOR.L.fmt fd, fs

Description:

The contents of floating-point register fs are arithmetically converted into a 64-bit fixed-point format, and the result is stored to floating-point register *fd*. The source operand is processed in the floating-point format *fmt*.

The result of the conversion is rounded toward the $-\infty$ direction, regardless of the current rounding mode.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the *Status* register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as a floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit of the *Status* register is 1, both the odd and even register numbers are valid.

If the source operand is infinite or NaN, and if the rounded result is outside the range of $2^{63}-1$ to -2^{63}, the invalid operation exception occurs. If the invalid operation exception is not enabled, the exception does not occur, and $2^{63}-1$ is returned.

This operation is defined in the 64-bit mode and 32-bit Kernel mode. If this instruction is executed during 32-bit User/Supervisor mode, a reserved instruction exception occurs.

FLOOR.L.fmt Floating-point Floor To Long Fixed-point Format (continued) FLOOR.L.fmt

Operation:

```
64      T:    StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt) , fmt, L) )
```

Remark Same operation in the 32-bit Kernel mode.

Exceptions:

Coprocessor unusable exception
Floating-point exception
Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

Floating-Point Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact operation exception
Overflow exception

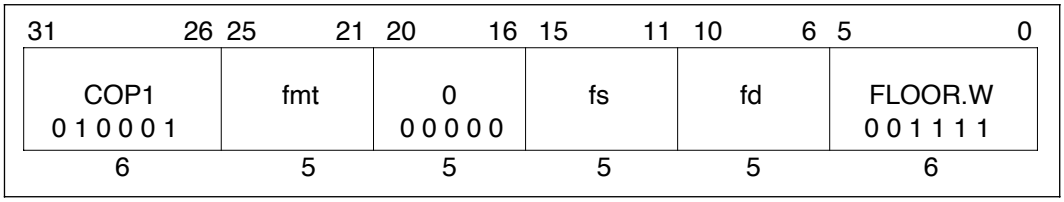
★ **Restrictions:**

An unimplemented operation exception will occur in the following cases.

- If an overflow occurs during conversion to integer format
- If the source operand is an infinite number
- If the source operand is NaN

Essentially, if any of bits 53 to 62 of the result of conversion from a floating-point format to a fixed-point format is 1, an unimplemented operation exception will occur. This includes cases when there is an overflow during conversion.

FLOOR.W.fmt Floating-point Floor To Single Fixed-point Format FLOOR.W.fmt



Format:

FLOOR.W.fmt fd, fs

Description:

The contents of floating-point register fs are arithmetically converted into a 32-bit fixed-point format, and the result is stored to floating-point register *fd*. The source operand is processed in the floating-point format *fmt*.

The result of the conversion is rounded toward the $-\infty$ direction, regardless of the current rounding mode.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the *Status* register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as a floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit of the *Status* register is 1, both the odd and even register numbers are valid.

If the source operand is infinite or NaN, and if the rounded result is outside the range of $2^{31}-1$ to -2^{31}, the invalid operation exception occurs. If the invalid operation exception is not enabled, the exception does not occur, and $2^{31}-1$ is returned.

FLOOR.W.fmt Floating-point Floor To Single Fixed-point Format (continued) FLOOR.W.fmt

Operation:

```
32, 64      T:      StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt) , fmt, W) )
```

Exceptions:

Coprocessor unusable exception
Floating-point exception

Floating-Point Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact operation exception
Overflow exception

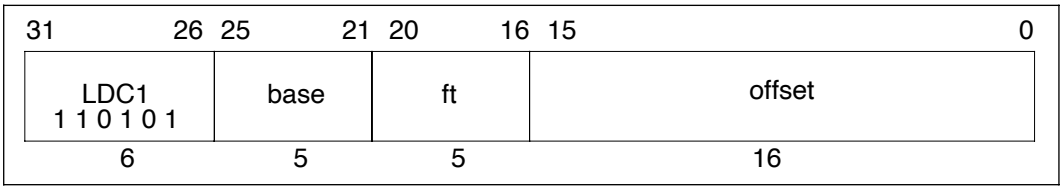
★ **Restrictions:**

An unimplemented operation exception will occur in the following cases.

- If an overflow occurs during conversion to integer format
- If the source operand is an infinite number
- If the source operand is NaN

Essentially, if any of bits 53 to 62 of the result of conversion from a floating-point format to a fixed-point format is 1, an unimplemented operation exception will occur. This includes cases when there is an overflow during conversion.

LDC1 Load Doubleword To FPU (Coprocessor 1) LDC1



Format:

LDC1 ft, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general purpose register *base* to form a virtual address.

If the *FR* bit of the *Status* register is 0, the contents of the doubleword at the memory location specified by the virtual address are loaded to floating-point registers ft and ft+1. At this time, the high-order 32 bits of the doubleword are stored to an odd-numbered register specified by ft+1, and the low-order 32 bits are stored to an even-numbered register specified by *ft*. The operation is undefined if the least significant bit in the *ft* field is not 0.

If the *FR* bit is 1, the contents of the doubleword at the memory location specified by the virtual address are loaded to floating-point register *ft*.

If any of the low-order three bits of the address are not zero, an address error exception occurs.

LDC1 Load Doubleword To FPU (Coprocessor 1) (continued) LDC1

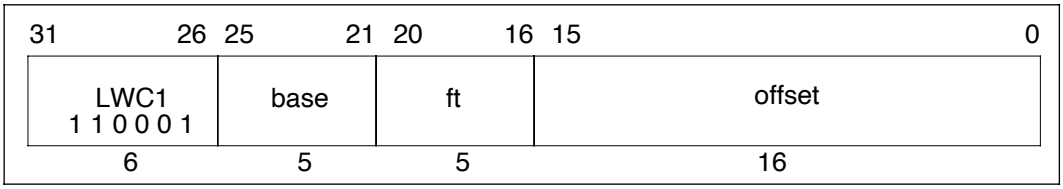
Operation:

```
32      T:  vAddr ← ( (offset15)^16 || offset15...0) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            data ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
            if SR26 = 1 then
                FGR [ft] ← data
            elseif ft0 = 0 then
                FGR [ft+1] ← data63...32
                FGR [ft] ← data31...0
            else
                undefined_result
            endif
64      T:  vAddr ← ( (offset15)^48 || offset15...0) + GPR[base]
            (pAddr, uncached) ← Address Translation (vAddr, DATA)
            data ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
            if SR26 = 1 then
                FGR [ft] ← data
            elseif ft0 = 0 then
                FGR [ft+1] ← data63...32
                FGR [ft] ← data31...0
            else
                undefined_result
            endif
```

Exceptions:

Coprocessor unusable
TLB miss exception
TLB invalid exception
Bus error exception
Address error exception

LWC1 Load Word To FPU (Coprocessor 1) LWC1



Format:

LWC1 ft, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general purpose register *base* to form a virtual address. The contents of the word at the memory location specified by the virtual address are loaded to floating-point register *ft*.

If the *FR* bit of the *Status* register is 0 and if the least-significant bit in the *ft* field is 0, the contents of the word are stored to the low-order 32 bits of floating-point register *ft*. If the least-significant bit in the ft area is 1, the contents of the word are stored to the high-order 32 bits of floating-point register ft-1.

If the *FR* bit is 1, all the 64-bit floating-point registers can be accessed; therefore, the contents of the word are stored to floating-point register *ft*. The value of the high-order 32 bits is undefined.

If either of the low-order two bits of the address is not zero, an address error exception occurs.

LWC1 Load Word To FPU (Coprocessor 1) (continued) LWC1

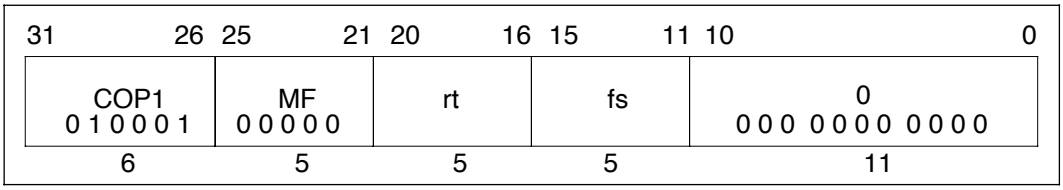
Operation:

```
32      T:      vAddr ← ( (offset15)^16 || offset15...0) + GPR[base]
                (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
                data ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
                if SR26 = 1 then
                    FGR [ft] ← undefined^32 || data
                else
                    FGR [ft] ← data
                endif
64      T:      vAddr ← ( (offset15)^48 || offset15...0) + GPR[base]
                (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
                data ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
                if SR26 = 1 then
                    FGR [ft] ← undefined^32 || data
                else
                    FGR [ft] ← data
                endif
```

Exceptions:

Coprocessor unusable exception
TLB miss exception
TLB invalid exception
Bus error exception
Address error exception

MFC1 Move Word From FPU (Coprocessor 1) MFC1



Format:

MFC1 rt, fs

Description:

The contents of floating-point general purpose register *fs* are stored to the general purpose register *rt* of the CPU register *rt*.

The contents of general purpose register *rt* are undefined while the instruction immediately following this instruction is being executed.

If the *FR* bit of the *Status* register is 0 and if the least-significant bit in the *ft* field is 0, the low-order 32 bits of floating-point register *ft* are stored to the *general purpose* register *rt*. If the least-significant bit in the ft area is 1, the high-order 32 bits of floating-point register ft-1 are stored to the *general purpose* register *rt*.

If the *FR* bit is 1, all the 64-bit floating-point registers can be accessed; therefore, the low-order 32 bits of floating-point register *ft* are stored to the *general purpose* register *rt*.

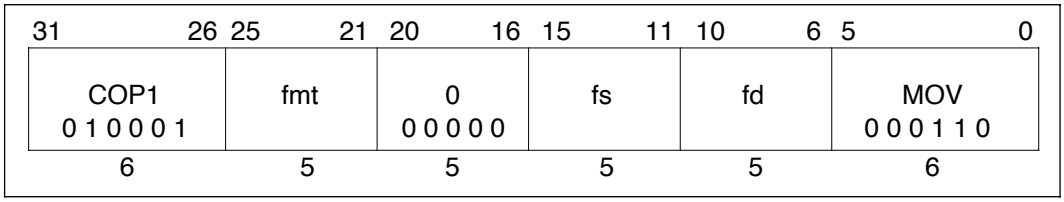
Operation:

```
32      T:      data ← FGR [fs]31...0
        T+1:    GPR [rt] ← data
64      T:      data ← FGR [fs]31...0
        T+1:    GPR[rt] ← (data31)32 || data
```

Exceptions:

Coprocessor unusable exception

MOV.fmt Floating-point Move MOV.fmt



Format:

MOV.fmt fd, fs

Description:

The contents of floating-point register fs are stored to floating-point register *fd*. The operand is processed in the floating-point format *fmt*.

This instruction is not executed arithmetically, and the IEEE754 exception does not occur.

This instruction is valid only in the single- and double-precision floating-point formats.

If the *FR* bit of the status register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as a floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit of the status bit is 1, both the odd and even register numbers are valid.

Operation:

```
32, 64      T:      StoreFPR (fd, fmt, ValueFPR (fs, fmt) )
```

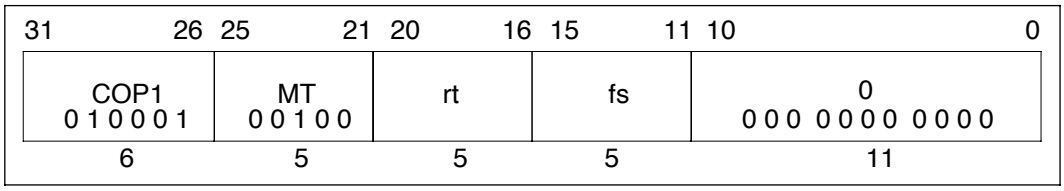
Exceptions:

Coprocessor unusable exception
Floating-point exception

Floating-Point Exceptions:

Unimplemented operation exception

MTC1 Move To FPU (Coprocessor 1) MTC1



Format:

MTC1 rt, fs

Description:

The contents of general purpose of the CPU register *rt* are loaded into the floating-point general purpose register *fs*.

The contents of floating-point register *fs* is undefined while the instruction immediately following this instruction is being executed.

The *FR* bit of the *Status* register specifies the method of access to the Floating-Point General Purpose registers.

If *FR* bit equals zero, all 32 Floating-Point General Purpose registers can be accessed. Access an odd-numbered register for the high-order 32 bits and an even-numbered register for the low-order 32 bits in the format of the floating-point operation instruction when transferring double-precision data.

If the *FR* bit is 1, all the 32 floating-point general purpose registers can be accessed, but the low-order 32 bits of the register are accessed for data.

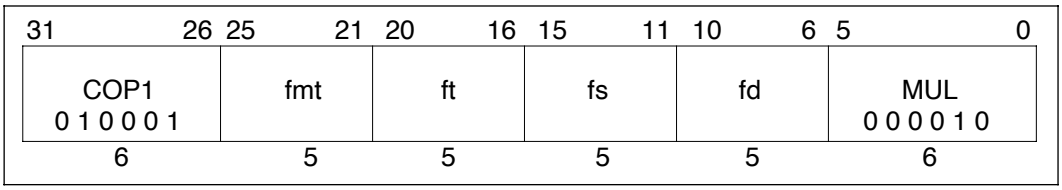
Operation:

```
32, 64  T:      data ← GPR [rt]2_31..0
        T+1:    if SR_26= 1 then
                    FGR [fs] ← undefined^32 II data
                else
                    FGR [fs] ← data
                endif
```

Exceptions:

Coprocessor unusable exception

MUL.fmt Floating-point Multiply MUL.fmt



Format:

MUL.fmt fd, fs, ft

Description:

The contents of floating-point register *fs* are multiplied by those of floating-point register *ft*, and the result is stored to floating-point register *fd*. The operand is processed in the floating-point format *fmt*.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the *Status* register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as a floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit of the *Status* register is 1, both the odd and even register numbers are valid.

Operation:

```
32, 64      T:      StoreFPR (fd, fmt, ValueFPR (fs, fmt) * ValueFPR (ft, fmt) )
```

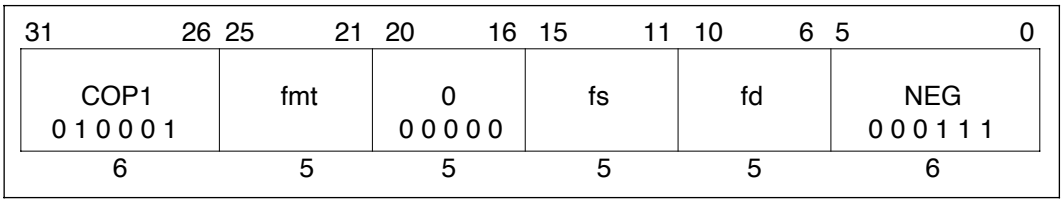
Exceptions:

Coprocessor unusable exception
Floating-point exception

Floating-Point Exceptions:

Unimplemented operation exception
Invalid operation exception
Inexact operation exception
Overflow exception
Underflow exception

NEG.fmt Floating-point Negate NEG.fmt



Format:

NEG.fmt fd, fs

Description:

The sign of the contents of floating-point register *fs* is inverted and the result to floating-point register *fd* is stored. The operand is processed in the floating-point format *fmt*.

The sign is inverted arithmetically. Therefore, the instruction is invalid if NaN is specified as the operand.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the *Status* register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as a floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit of the *Status* register is 1, both the odd and even register numbers are valid.

Operation:

```
32, 64    T:    StoreFPR (fd, fmt, Negate (ValueFPR (fs, fmt) ) )
```

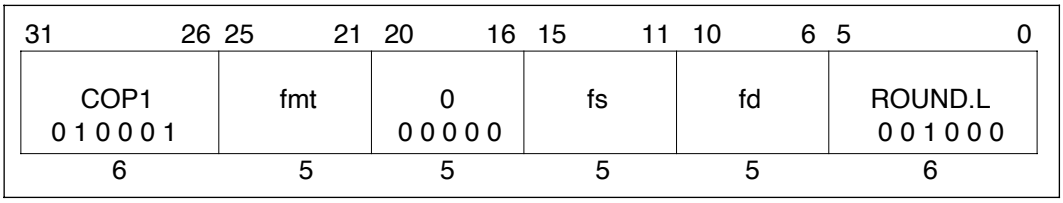
Exceptions:

Coprocessor unusable exception
Floating-point exception

Floating-Point Exceptions:

Unimplemented operation exception
Invalid operation exception

ROUND.L.fmt Floating-point Round To Long Fixed-point Format ROUND.L.fmt



Format:

ROUND.L.fmt fd, fs

Description:

The contents of floating-point register fs are converted into the 64-bit fixed-point format, and the result is stored to floating-point register *fd*. The source operand is processed in the floating-point format *fmt*.

The result of the conversion is rounded to the closest value or even number regardless of the current rounding mode.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the *Status* register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as a floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit of the *Status* register is 1, both the odd and even register numbers are valid.

If the source operand is infinite or NaN, and if the rounded result is outside the range of $2^{63} - 1$ to -2^{63}, the invalid operation exception occurs. If the invalid operation exception is not enabled, the exception does not occur, and $2^{63} - 1$ is returned.

This operation is defined in the 64-bit mode and 32-bit Kernel mode. If this instruction is executed during 32-bit User/Supervisor mode, a reserved instruction exception occurs.

ROUND.L.fmt Floating-point Round To Long Fixed-point Format (continued) ROUND.L.fmt

Operation:

```
64      T:    StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt) , fmt, L) )
```

Remark Same operation in the 32-bit Kernel mode.

Exceptions:

Coprocessor unusable exception
Floating-point exception
Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

Floating-Point Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact operation exception
Overflow exception

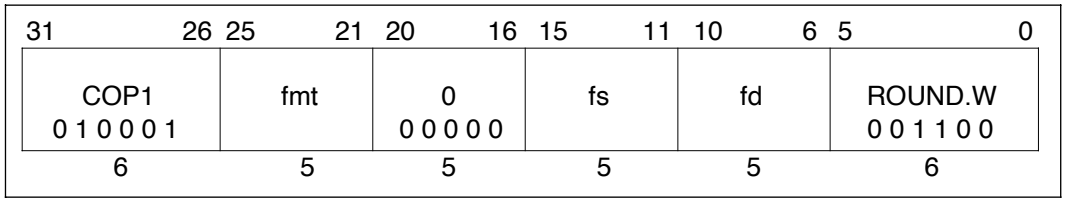
★ **Restrictions:**

An unimplemented operation exception will occur in the following cases.

- If an overflow occurs during conversion to integer format
- If the source operand is an infinite number
- If the source operand is NaN

Essentially, if any of bits 53 to 62 of the result of conversion from a floating-point format to a fixed-point format is 1, an unimplemented operation exception will occur. This includes cases when there is an overflow during conversion.

ROUND.W.fmt Floating-point Round To Single Fixed-point Format ROUND.W.fmt



Format:

ROUND.W.fmt fd, fs

Description:

The contents of floating-point register fs are converted into the 32-bit fixed-point format, and the result is stored to floating-point register *fd*. The source operand is processed in the floating-point format *fmt*.

The result of the conversion is rounded to the closest value or even number regardless of the current rounding mode.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the *Status* register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as a floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit of the *Status* register is 1, both the odd and even register numbers are valid.

If the source operand is infinite or NaN, and if the rounded result is outside the range of $2^{31} - 1$ to -2^{31}, the invalid operation exception occurs. If the invalid operation exception is not enabled, the exception does not occur, and $2^{31} - 1$ is returned.

ROUND.W.fmt Floating-point Round To Single Fixed-point Format (continued) ROUND.W.fmt

Operation:

```
32, 64      T:      StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt) , fmt, W) )
```

Exceptions:

Coprocessor unusable exception
Floating-point exception

Floating-Point Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact operation exception
Overflow exception

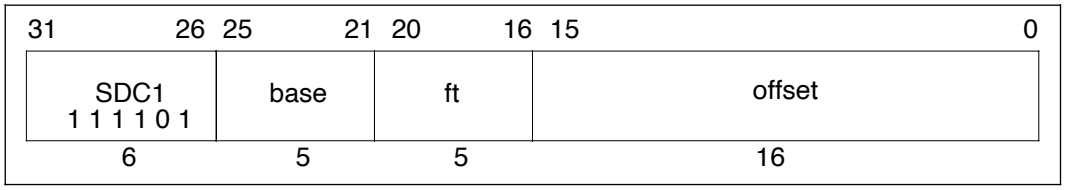
★ **Restrictions:**

An unimplemented operation exception will occur in the following cases.

- If an overflow occurs during conversion to integer format
- If the source operand is an infinite number
- If the source operand is NaN

Essentially, if any of bits 53 to 62 of the result of conversion from a floating-point format to a fixed-point format is 1, an unimplemented operation exception will occur. This includes cases when there is an overflow during conversion.

SDC1 Store Doubleword From FPU (Coprocessor 1) SDC1



Format:

SDC1 ft, offset(base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general purpose register *base* to form a virtual address.

The contents of floating-point registers *ft* and ft+1 are stored to the memory position specified by the virtual address as a doubleword if the *FR* bit of the *Status* register is 0. At this time, the contents of the odd-numbered register specified by ft+1 correspond to the high-order 32 bits of the doubleword, and the contents of the even-numbered register specified by ft correspond to the low-order 32 bits.

If the least significant bit in the ft field is not 0, this instruction is not defined.

If the *FR* bit is 1, the contents of floating-point register *ft* are stored to the memory location specified by the virtual address as a doubleword.

If any of the low-order three bits of the address are not zero, an address error exception occurs.

SDC1 Store Doubleword From FPU (Coprocessor 1) (continued) SDC1

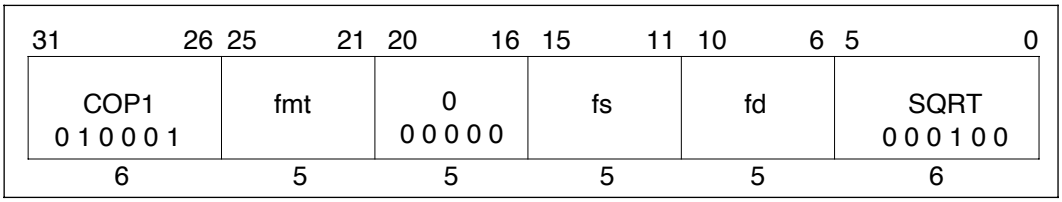
Operation:

```
32    T:  vAddr ← ( (offset15)^16 || offset15...0) + GPR [base]
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          if SR26 = 1
              data ← FGR [ft]63...0
          elseif ft0 = 0 then
              data ← FGR [ft+1]31...0 || FGR [ft]31...0
          else
              data ← undefined^64
          endif
          StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
64    T:  vAddr ← ( (offset15)^48 || offset15...0) + GPR [base]
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          if SR26 = 1
              data ← FGR [ft]63...0
          elseif ft0 = 0 then
              data ← FGR [ft+1]31...0 || FGR [ft]31...0
          else
              data ← undefined^64
          endif
          StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
```

Exceptions:

Coprocessor unusable
TLB miss exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception

SQRT.fmt Floating-point Square Root SQRT.fmt



Format:

SQRT.fmt fd, fs

Description:

The positive arithmetic square root of the contents of floating-point register *fs* is calculated and the result is stored to floating-point register *fd*. The operand is processed in the floating-point format *fmt*. The result is rounded as if calculated to infinite precision and then rounded according to the current rounding mode. If the value of the source operand is –0, the result will be –0. The result is placed in the floating-point register specified by *fd*.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the *Status* register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as a floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit of the *Status* register is 1, both the odd and even register numbers are valid.

Operation:

```
32, 64     T:     StoreFPR (fd, fmt, SquareRoot (ValueFPR (fs, fmt) ) )
```

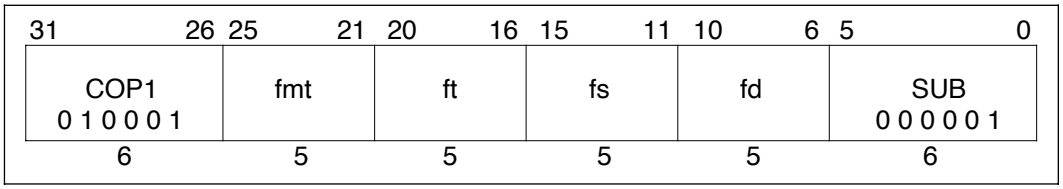
Exceptions:

Coprocessor unusable exception
Floating-point exception

Floating-Point Exceptions:

Unimplemented operation exception
Invalid operation exception
Inexact operation exception

SUB.fmt Floating-point Subtract SUB.fmt



Format:

SUB.fmt fd, fs, ft

Description:

The contents of floating-point register *ft* from those of floating-point register *fs*, and the result is stored to floating-point register *fd*. The result is rounded as if calculated to infinite precision and then rounded according to the current rounding mode.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the *Status* register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as a floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit of the *Status* register is 1, both the odd and even register numbers are valid.

Operation:

```
32, 64    T:    StoreFPR (fd, fmt, ValueFPR (fs, fmt) – ValueFPR (ft, fmt) )
```

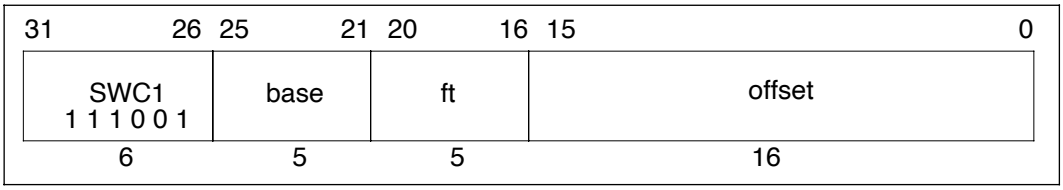
Exceptions:

Coprocessor unusable exception
Floating-point exception

Floating-Point Exceptions:

Unimplemented operation exception
Invalid operation exception
Inexact operation exception
Overflow exception
Underflow exception

SWC1 Store Word From FPU (Coprocessor 1) SWC1



Format:

SWC1 ft, offset (base)

Description:

The 16-bit *offset* is sign-extended and added to the contents of general purpose register *base* to form a virtual address. The contents of the floating-point general purpose register *ft* are stored at the memory location of the specified address.

If the *FR* bit of the *Status* register is 0 and the least-significant bit in the *ft* field is 0, the contents of the low-order 32 bits of floating-point register *ft* are stored. If the least-significant bit in the *ft* field is 1, the contents of the high-order 32 bits of floating-point register ft-1 are stored.

If the *FR* bit is 1, all the 64-bit floating-point registers can be accessed; therefore, the contents of the low-order 32 bits in the ft field are stored.

If either of the low-order two bits of the address are not zero, an address error exception occurs.

SWC1 Store Word From FPU (Coprocessor 1) (continued) SWC1

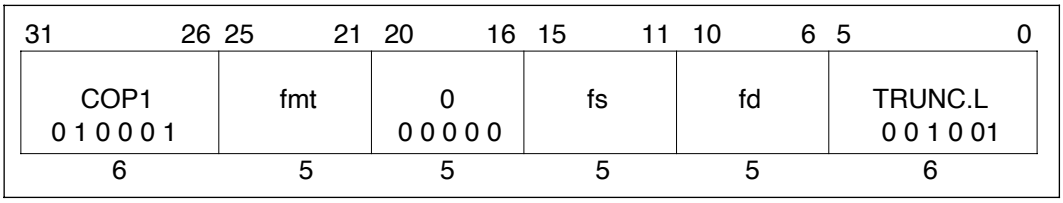
Operation:

32	T:	vAddr ← ((offset$_{15}$)^{16} ll offset$_{15...0}$) + GPR[base] (pAddr, uncached) ← AddressTranslation (vAddr, DATA) data ← FGR [ft]$_{31...0}$ StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
64	T:	vAddr ← ((offset$_{15}$)^{48} ll offset$_{15...0}$) + GPR[base] (pAddr, uncached) ← AddressTranslation (vAddr, DATA) data ← FGR [ft]$_{31...0}$ StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)

Exceptions:

Coprocessor unusable
TLB miss exception
TLB invalid exception
TLB modification exception
Bus error exception
Address error exception

TRUNC.L.fmt Floating-point Truncate To Long Fixed-point Format TRUNC.L.fmt



Format:

TRUNC.L.fmt fd, fs

Description:

The contents of floating-point register fs are converted into the 64-bit fixed-point format, and the result is stored to floating-point register *fd*. The source operand is processed in the floating-point format *fmt*.

The result of the conversion is rounded toward the 0 direction, regardless of the current rounding mode.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the *Status* register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as a floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit of the *Status* register is 1, both the odd and even register numbers are valid.

If the source operand is infinite or NaN, and if the rounded result is outside the range of $2^{63} - 1$ to -2^{63}, the invalid operation exception occurs. If the invalid operation exception is not enabled, the exception does not occur, and $2^{63} - 1$ is returned.

This operation is defined in the 64-bit mode and 32-bit Kernel mode. If this instruction is executed during 32-bit User/Supervisor mode, a reserved instruction exception occurs.

TRUNC.L.fmt Floating-point Truncate To Long Fixed-point Format (continued) TRUNC.L.fmt

Operation:

```
64      T:    StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt) , fmt, L) )
```

Remark Same operation in the 32-bit Kernel mode.

Exceptions:

Coprocessor unusable exception
Floating-point exception
Reserved instruction exception (V_R4300 in 32-bit User or Supervisor mode)

Floating-Point Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact operation exception
Overflow exception

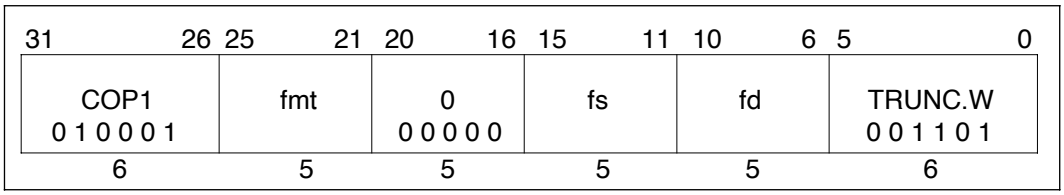
★ **Restrictions:**

An unimplemented operation exception will occur in the following cases.

- If an overflow occurs during conversion to integer format
- If the source operand is an infinite number
- If the source operand is NaN

Essentially, if any of bits 53 to 62 of the result of conversion from a floating-point format to a fixed-point format is 1, an unimplemented operation exception will occur. This includes cases when there is an overflow during conversion.

TRUNC.W.fmt Floating-point Truncate To Single Fixed-point Format TRUNC.W.fmt



Format:

TRUNC.W.fmt fd, fs

Description:

The contents of floating-point register fs are arithmetically converted into a 32-bit fixed-point single format, and the result is stored to floating-point register *fd*. The source operand is processed in the floating-point format *fmt*.

The result of the conversion is rounded toward the 0 direction, regardless of the current rounding mode.

This instruction is valid only for conversion from the single- or double-precision floating-point format.

If the *FR* bit of the *Status* register is 0, only an even number can be specified as a register number because adjacent even-numbered and odd-numbered registers are used in pairs as a floating-point registers. If an odd number is specified, the operation is undefined. If the *FR* bit of the *Status* register is 1, both the odd and even register numbers are valid.

If the source operand is infinite or NaN, and if the rounded result is outside the range of $2^{31} - 1$ to -2^{31}, the invalid operation exception occurs. If the invalid operation exception is not enabled, the exception does not occur, and $2^{31} - 1$ is returned.

TRUNC.W.fmt Floating-point Truncate To Single Fixed-point Format (continued) TRUNC.W.fmt

Operation:

```
32, 64    T:    StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt) , fmt, W) )
```

Exceptions:

Coprocessor unusable exception
Floating-point exception

Floating-Point Exceptions:

Invalid operation exception
Unimplemented operation exception
Inexact operation exception
Overflow exception

★ **Restrictions:**

An unimplemented operation exception will occur in the following cases.

- If an overflow occurs during conversion to integer format
- If the source operand is an infinite number
- If the source operand is NaN

Essentially, if any of bits 53 to 62 of the result of conversion from a floating-point format to a fixed-point format is 1, an unimplemented operation exception will occur. This includes cases when there is an overflow during conversion.

17.6 FPU Instruction Opcode Bit Encoding

Figure 17-3 lists the Bit Encoding for FPU instructions.

Opcode

31...29 \ 28...26	0	1	2	3	4	5	6	7
0								
1								
2		COP1						
3								
4								
5								
6		LWC1				LDC1		
7		SWC1				SDC1		

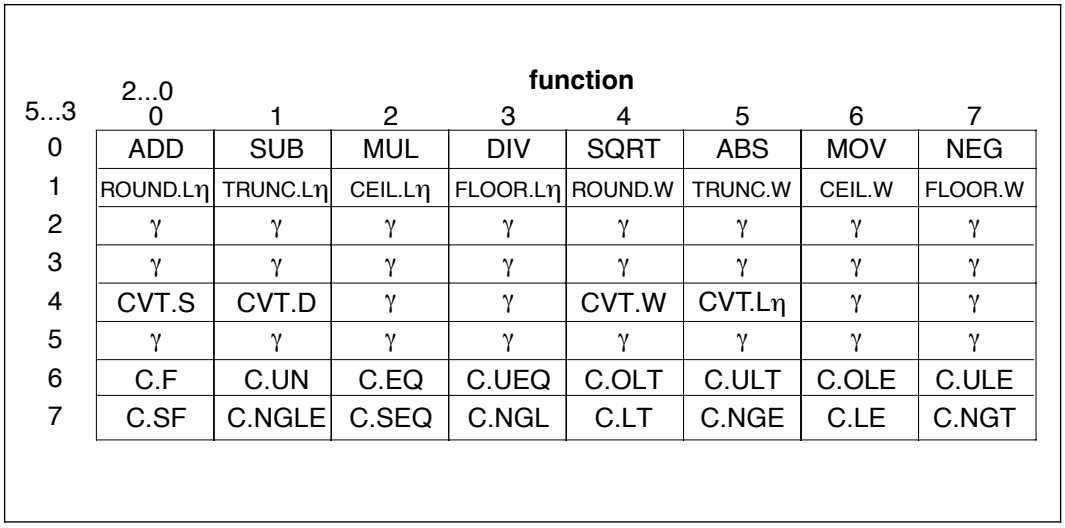
sub

25...24 \ 23...21	0	1	2	3	4	5	6	7
0	MF	DMFη	CF	γ	MT	DMTη	CT	γ
1	BC	γ	γ	γ	γ	γ	γ	γ
2	S	D	γ	γ	W	Lη	γ	γ
3	γ	γ	γ	γ	γ	γ	γ	γ

br

20...19 \ 18...16	0	1	2	3	4	5	6	7
0	BCF	BCT	BCFL	BCTL	*	*	*	*
1	*	*	*	*	*	*	*	*
2	*	*	*	*	*	*	*	*
3	*	*	*	*	*	*	*	*

Figure 17-3 Bit Encoding for FPU Instructions (1/2)



5...3 \ 2...0	function 0	1	2	3	4	5	6	7
0	ADD	SUB	MUL	DIV	SQRT	ABS	MOV	NEG
1	ROUND.Lη	TRUNC.Lη	CEIL.Lη	FLOOR.Lη	ROUND.W	TRUNC.W	CEIL.W	FLOOR.W
2	γ	γ	γ	γ	γ	γ	γ	γ
3	γ	γ	γ	γ	γ	γ	γ	γ
4	CVT.S	CVT.D	γ	γ	CVT.W	CVT.Lη	γ	γ
5	γ	γ	γ	γ	γ	γ	γ	γ
6	C.F	C.UN	C.EQ	C.UEQ	C.OLT	C.ULT	C.OLE	C.ULE
7	C.SF	C.NGLE	C.SEQ	C.NGL	C.LT	C.NGE	C.LE	C.NGT

Figure 17-3 Bit Encoding for FPU Instructions (2/2)

Key:

* When the operation code marked with an asterisk is executed, the reserved instruction exception occurs. This code is reserved for future expansion.

γ Operation codes marked with a gamma cause unimplemented operation exceptions in all current implementations and are reserved for future expansion.

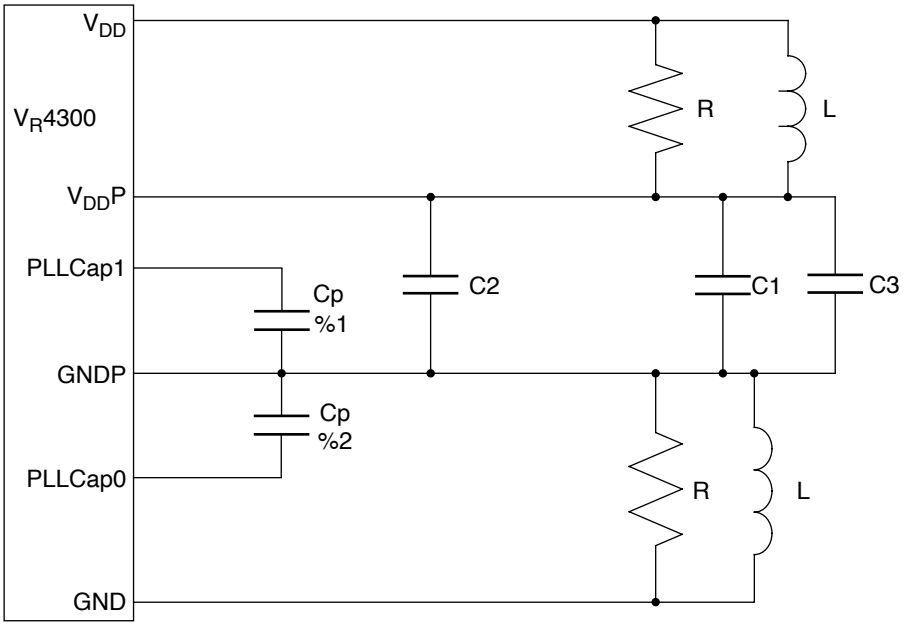
η When the operation code marked with an eta is executed, the result is valid only when use of the MIPS III instruction set is enabled. If the operation code is executed when use of the instruction set is disabled (in the 32 bit User/Supervisor mode), the unimplemented operation exception occurs.

PLL Passive Elements

18

Connect several passive elements externally to the V_R4300 so that the processor can operate normally. Connect the elements to the PLLCap0, PLLCap1, V_{DD}P, and GNDP pins.

Figure 18-1 shows the connections of the passive elements for PLL.



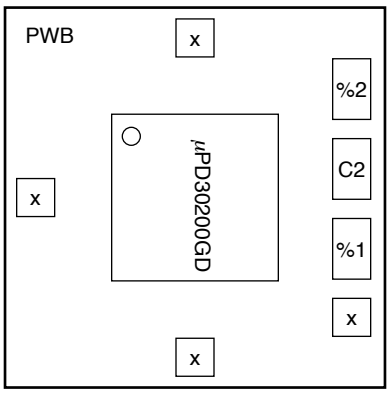
Remarks 1. C1, C2, C3, Cp%1, Cp%2, R, and L are mounted on the board.

2. Either R or L may do in a system where it has been confirmed through experiment that noise is not superimposed on V_{DD}P and GNDP.

3. The value of each element differs depending on the system. Find the appropriate values for each system through experiment.

Figure 18-1 Connection Example of PLL Passive Elements

Figure 18-2 shows a layout example of 120-pin plastic QFP and capacitor on PWB.



Remarks x : GND-V_{DD} Bypass Capacitors
C2 : GNDP-V_{DD}P Bypass Capacitors
%1, %2 : PLL Capacitors

Figure 18-2 Layout Example of QFP and Capacitor on PWB

Separate the wiring of the power (V_{DD}P) and ground (GNDP) for PLL from the normal power (V_{DD}) and ground (GND) wiring. Here is an example of the value of each element.

R = 5Ω C1 = 1 nF C2 = 82 nF

C3 = 10 μF Cp = 470 pF

Because the optimum values of filter elements differ depending on the application and noise environment of the system. Therefore, the above values are given for reference only. Find the optimum values for users' application through trial and error. A choke element (inductor: L) may be used instead of the resistor (R) used as a power filter.

[MEMO]

Coprocessor 0 Hazards

19

If a conflict of internal resources takes place between instructions (such as when the contents of the destination register are used as the source for the next instruction), the V_R4300 interlocks the pipeline to prevent conflict of internal resources. Therefore, it is not necessary to insert a NOP instruction between instructions.

However, the *CP0* register and TLB are not interlocked. When developing a program that uses the *CP0* register and TLB, therefore, take conflict of the internal resources into consideration. CP0 hazard defines the number of NOP instructions to be inserted between instructions to avoid conflict of internal resources, or the number of instructions independent of the conflict. This chapter explains this CP0 hazard.

The value of V_R4300 CP0 hazards is equivalent or less than those of the V_R4400; Table 19-1 lists the V_R4300 CP0 hazards. Code which complies with these hazards will run without modification on the V_R4400 or V_R4200.

When the data of the *CP0* register or bit is defined in the Source column in the following table, that data can be used as a source. If data is stored in the *CP0* register or bit shown in the Destination column, that data is used as the destination.

The number of NOP instructions between the instructions related to the *CP0* register and TLB, or the number of the instructions independent of the conflict can be calculated from the following expression, using this table.

```
(Number of destination hazards of instruction A) -

{(Number of source hazards of instruction B) +1}
```

As an example, to find the number of instructions required between an MTC0 and a subsequent MFC0 instruction, this is:

```
(7) - (4 + 1) = 2 instructions
```

Caution The hazard related to CP0 does not generate the interlock of the pipeline. Therefore, control the number of required instructions by program.

Table 19-1 Coprocessor 0 Hazards

Operation	Source		Destination	
	Name	Number of Hazard	Name	Number of Hazard
MTC0			cpr rd	7
MFC0	cpr rd	4		
TLBR	Index, TLB	5-7	PageMask, EntryHi EntryLo0, EntryLo1	8
TLBWI TLBWR	Index or Random PageMask, EntryHi, EntryLo0, EntryLo1	5-8	TLB	8
TLBP	PageMask, EntryHi	3-6	Index	7
ERET	EPC or ErrorEPC, Status, TLB	4 γ	Status.EXL, Status.ERL	4-8 α
			LLbit	7
CACHE Index Load Tag			TagLo, TagHi, ECC	8 β
CACHE Index Store Tag	TagLo, TagHi, ECC	7		
CACHE Hit ops.			Status.CH	8
Coprocessor usable test	Status.CU, Status.KSU Status.EXL, Status.ERL	2		
Instruction fetch	EntryHi.ASID Status.KSU, Status.EXL, Status.ERL, Status.RE, Config.K0	0		
	TLB	2		
Instruction fetch exception			EPC, Status	8
			Cause, BadVAddr, Context	3
Interrupt	Cause.IP, Status.IM Status.IE, Status.EXL Status.ERL	3		
Load/Store	EntryHi.ASID Status.KSU, Status.EXL, Status.ERL, Status.RE, Config.K0, TLB	4		
	WatchHi, WatchLo	4-5		
Load/Store exception			EPC, Status, Cause	8
			BadVAddr, Context	
TLB shutdown			Status.TS	7

Remarks 1. A hazard is associated when an instruction related to the bit specified by the source or destination is executed. For example, if CP1 is enabled by setting Status.C to 1 by the MTC0 instruction, all the instructions using CP1 (FPU) are subject to hazard.

2. α Status.EXL and Status.ERL are cleared in stage 8, but the effect of clearing them is visible at the time of an instruction fetch starting at the beginning of stage 4.

β One instruction to separate Index Load Tag and MFC0 Tag will do, even though α above would imply three instructions.

- The instruction following a MTC0 instruction must not be a MFC0 instruction.
- The five instructions following a MTC0 instruction to *Status* register that changes KSU bit and sets EXL or ERL bits may be executed in the new mode, and not in the Kernel mode. This can be avoided by setting EXL bit first, leaving KSU bit set to Kernel, and later changing KSU bit.
- There must be two non-load, non-CACHE instructions between a store instruction and a CACHE instruction directed to the same cache line as the store destination.

γ An ERET instruction following an MTC0 instruction that sets the ERL bit in the *Status* register (Status.ERL) must be separated from the MTC0 instruction by three instructions.

Cautions 1. If the K0 bit of the config register is changed to the non-cache mode by using the MTC0 instruction, the non-cache area is set when the instruction fetch two instructions after the MTC0 instruction is executed.

2. If a jump or branch instruction is executed immediately after the ITS bit of the *Status* register has been set, a stall lasting for several instructions will occur.

The status in which CP0 hazard must be taken into consideration when each instruction is executed is explained below.

(1) MTC0

Destination: Completion of writing to *destination* register (CP0) by MTC0 instruction

(2) MFC0

Source: Determination of *source* register (CP0) of MFC0 instruction

(3) TLBR

Source: Determination of TLB status and *Index* register before execution of TLBR instruction

(4) TLBWI, TLBWR

Source: Determination of *source* register of TLBWI and TLBWR instructions and register used for TLB entry specification
Destination: Completion of writing to TLB by TLBWI and TLBWR instructions

(5) TLBP

Source: Determination of *PageMask* register and *EntryHi* register before execution of TLBP instruction
Destination: Completion of writing result of TLBP instruction execution to *Index* register

(6) ERET

Source: Determination of register holding information necessary for ERET instruction execution
Destination: Completion of processor status transition due to ERET instruction execution

(7) CACHE Index Load Tag

Destination: Completion of writing execution of this instruction to each register

(8) CACHE Index Store Tag

Source: Determination of register holding information necessary for execution of this instruction

(9) Coprocessor use test

Source: Determination of mode set by bit value of *CP0* register in Source column

Examples 1. When accessing the *CP0* register in the user mode after changing the content of the *Status.CU0* bit or when executing an instruction using the resources of *CP0* (such as TLB instruction, Cache instruction, or branch instruction)

2. When accessing the *CP0* register in the operating mode used after the contents of the *Status.KSU*, *EXL*, and *ERL* bits have been changed

3. When using the FPU (CP1) after the content of the *Status.CU1* bit has been changed

(10) Instruction fetch

Source: Determination of operating mode and TLB necessary for instruction fetch

Examples 1. When fetching instructions after the mode has been changed from User to Kernel after the contents of the *Status.KSU*, *EXL*, and *ERL* bits have been changed

2. When rewriting TLB and fetching an instruction by using its TLB entry

(11) Instruction fetch exception

Destination: Completion of writing to each register holding information related to an exception when the exception has occurred as a result of instruction fetch

(12) Interrupt

Source: Determination of each register that identifies an exception generation condition when an interrupt cause occurs

(13) Load/store

Source: Determination of operating mode related to address generation by load/store instruction, determination of TLB entry, determination of cache mode set by the *Config.K0* bit, and determination of a register that sets a watch exception generation condition

Example When executing the load/store instruction in the kernel area after the mode has been changed from User to Kernel

(14) Load/store exception

Destination: Completion of writing to each register holding information related to an exception when the exception occurs as a result of a load/store operation

(15) TLB shut down

Destination: Completion of writing to *Status.TS* bit when TLB shut down occurs

Table 19-2 shows examples of calculating the number of CP0 hazards and the number of instructions to be inserted.

Table19-2 Example of Calculating Number of CP0 Hazards and Number of Instructions Inserted

Destination	Source	Conflicting Internal Resources	Number of Instructions Inserted	Expression
TLBWR/TLBWI	TLBP	TLB Entry	3	8–(4+1)
TLBWR/TLBWI	Load/store using newly rewritten TLB	TLB Entry	3	8–(4+1)
TLBWR/TLBWI	Instruction fetch using newly rewritten TLB	TLB Entry	5	8–(2+1)
MTC0, Status [CU]	Coprocessor instruction requiring setting of CU	Status [CU]	4	7–(2+1)
TLBWR	MFC0 EntryHi	EntryHi	3	8–(4+1)
MTC0 EntryLo0	TLBWR/TLBWI	EntryLo0	1	7–(5+1)
TLBP	MFC0 Index	Index	2	7–(4+1)
MTC0 EntryHi	TLBP	EntryHi	1	7–(5+1)
MTC0 EPC	ERET	EPC	2	7–(4+1)
MTC0 Status	ERET	Status	3	7–(3+1)
MTC0 Status [IE]*	Instruction causing interrupt	Status [IE]	3	7–(3+1)

* The number of hazards is undefined if the execution sequence is changed by an exception. In this case, the minimum number of hazards until the value of the *IE* bit is determined and the maximum number of hazards until a pending and enabled interrupt occurs may be the same.

[MEMO]

Differences Between the V_R4300, V_R4305, and V_R4310

A

The following table describes the differences between the V_R4300, V_R4305, and V_R4310.

Table A-1 Differences Between the V_R4300, V_R4305, and V_R4310

Parameter		V_R4300	V_R4305	V_R4310
System bus	Write data transfer	Two buses (D/D××)		
	Initial value setting pins at reset time	DivMode (1:0) (Can be set on power application only)		DivMode (2:0) (Can be set on power application only)
	Block write access	Sequential ordering		
	State after final data write	Final data retained in transfer rate setting		
	Non-cache high-speed write	Provided		
Integer operation unit	Corresponding instructions	MIPS I, II, and III instruction sets		
Cache memory	Data protection	None		
JTAG interface		Provided		
SyncOut-Syncln path		Provided		
Clock interface	Input vs. internal multiplication rate	1.5[*1], 2, 3, 4[*2]	1, 2, 3	2, 2.5[*3], 3, 4, 5, 6
	Internal vs. bus frequency division rate	1.5[*1], 2, 3, 4[*2]	1, 2, 3	2, 2.5[*3], 3, 4, 5, 6
★ Power mode	Low power mode	Pipeline/system bus operated at a quarter of the normal rate[*4]		None
	Wait mode	None		
PRId register		Imp = 0 × 0B		

***1.** The 1.5 times frequency setting is allowed with the 100 MHz model only. (With the 133 MHz model, this setting is reserved.)

***2.** The 4 times frequency setting is allowed with the 133 MHz model only. (With the 100 MHz model, this setting is reserved.)

***3.** The 2.5 times frequency setting is allowed with the 167 MHz model only. (With the 133 MHz model, this setting is reserved.)

***4.** The 133 MHz model of the V_R4300 is not supported.

Differences from V_R4400

B

The V_R4300 is slightly different from the V_R4400 in terms of system design and software. This Appendix describes the differences between the V_R4300 and V_R4400.

The major differences lie in cache handling. This is because the V_R4300 does not support a secondary cache control function and a multi-processing function and because it employs a 32-bit external bus interface.

B.1 Differences in Software

The logical differences in software are the specifications of the CP0 registers. These differences are shown in Table B-1.

B.1.1 CACHE Instruction

Up to 4 MB of a secondary cache memory can be connected to the V_R4400. By contrast, the V_R4300 does not support a secondary cache. Therefore, the operations of the CACHE instructions that reference SD (secondary data cache) and SI (secondary instruction cache) are undefined.

All write back processing is transfer from the primary cache to the main memory.

The CACHE instruction Hit Set Virtual that is used to access the SD and SI with the V_R4400 is undefined with the V_R4300.

The Dirty bit (W bit of the V_R4400) of the data cache can be cleared by the CACHE instruction Hit_Write_Back.

The V_R4300 has a cache state bit. The V_R4400 has two cache state bits to support multi-processing. To manipulate this bit of the V_R4300, write the bit 7 of the TagLo register using a CACHE instruction (Index_Store_Tag_D). With the V_R4400, the bits 6 and 7 of the TagLo register are written.

B.1.2 Cache Parity

Because the V_R4300 does not check the cache data by using a parity, the cache error register (27) always outputs 0, and writing this register is ignored. The parity error register (26) can be used for only self-diagnosis and cannot be used to manipulate the cache.

B.1.3 Status Register

The bit specifications of the status registers are slightly different between the V_R4300 and V_R4400.

The fixed bits (bits 24 and 27) of the status register of the V_R4400 function as an instruction trace support (ITS) bit (bit 24) and low power mode* (RP) bit (bit 27) with the V_R4300.

★ * The low power mode is supported only in the 100 MHz model of the V_R4300 and the V_R4305. Fix the RP bit of the 133 MHz model of the V_R4300 and the V_R4310 to 0.

The CH bit of the V_R4300 can be written only by software. With the V_R4400, however, this bit is set or cleared by hardware when a secondary cache instruction is executed.

The *CE* and *DE* bits of the *Status* register of the V_R4300 are used to manipulate the parity and do not affect the operation.

For details, refer to **6.3.5 Status Register (12)**.

B.1.4 Config Register

The *Config* register of the V_R4300 only supports part of the bit functions of the *Config* register of the V_R4400.

For details, refer to **5.4.6 Config Register (16)**.

B.1.5 Status of FCR31 on Occurrence of Unimplemented Operation Exception

If the floating-point unimplemented operation exception occurs with the V_R4400, the cause bits of the FCR31 for the floating-point operation exception other than the unimplemented operation exception bit (E) are undefined. The exception handler for the unimplemented operation should ignore the cause bits other than the E bit.

The V_R4300 is more strictly defined. If the unimplemented operation exception occurs, the cause bits of the other floating-point operation exceptions are not set.

B.1.6 Integer Zero Division

If an integer is divided by zero, the result is undefined with MIPS ISA (Instruction Set Architecture). This illegal operation returns the following values to the registers of the V_R4300 and V_R4400.

Processor	Dividend	Lo Register	Hi Register
V_R4400	≥ 0	0xFFFF FFFF	Dividend
	< 0	0x0000 0001	Dividend
V_R4300	≥ 0	0x7FFF FFFF	Dividend
	< 0	0x8000 0001	Dividend

B.1.7 Cache Parity Error Exception

Because the V_R4300 does not check data by using a cache parity, a parity error exception does not occur.

Table B-1 Differences in Software

Function \ Product Name		V_R4300	V_R4400
CACHE instruction	Secondary cache	Not supported	Supported
	Parity	None	Provided
Status register	Bit 27	Low power mode*	0
	Bit 24	Instruction trace support	0
	CE and DE bits	Do not affect processor operation	Used for parity
Config register		Only part of bit functions supported	All supported
Unimplemented operation exception		Cause bits other than E bit cleared	Cause bits other than E bit undefined
Integer zero division		Value returned to register differs	
Cache error exception		Does not occur	Always normal operation

★ * 100 MHz model of the V_R4300 and the V_R4305 only

B.2 Differences in System Design

Next, the differences in system design between the V_R4300 and V_R4400 are described. Table B-2 shows these differences.

B.2.1 Initialization of Processor

With the V_R4400, many modes must be set on boot. Setting mode of the V_R4300 is more simple. This is because the V_R4300 sets mode not by software but by using external pins.

The **$\overline{\text{Reset}}$** signal of the V_R4300 may be active or inactive during cold reset. However, do not change the value of this signal during reset sequence.

At soft reset, assert the **$\overline{\text{Reset}}$** signal of the V_R4300 active for the duration of 16MasterClock or longer. With the V_R4400, the **$\overline{\text{Reset}}$** signal must be asserted active for the duration of at least 64MasterClock cycles.

B.2.2 System Interface

The SysAD bus of the V_R4400 is 64 bits wide, but the V_R4300 has a 32-bit SysAD bus without a parity check function.

Multi-Processing Function and Secondary Cache Control Function

The V_R4300 uses the same SysAD bus protocol as the V_R4400. But because the V_R4300 does not support a multi-processing function and a secondary cache control function, its external bus is provided with only part of the SysAD bus specifications.

The operations related to the multi-processing function and secondary cache that are defined for the V_R4400 are undefined with the V_R4300.

Line Size of Cache

The line size of the cache of the V_R4300 is as follows.

Instruction cache : 8 words (32 bytes)

Data cache : 4 words (16 bytes)

Data Transfer Rate

The V_R4400 has nine data rates (D, DDx, DDxx, DxDx, DDxxx, DDxxxx, DxxDxx, DDxxxxx, and DxxxDxxx).

The V_R4300 has two data rates (D and Dxx). These data rates are selected by using the *EP* bit of the *Config* register.

The V_R4400 requires at least 4 cycles as processor request cycles. Consequently, if successive single read request are made, or if write requests and read requests are made successively, two idle cycles are inserted in between two requests, like "ADxxAD".

If write or read are performed successively in the fastest mode (data rate: D) of the V_R4300, however, no idle cycle is needed between write/read cycles, like "ADAD".

When data is input from an external device, the V_R4300 can support any data transfer via the SysAD bus. The V_R4300 can input data at a data rate of "DDDDDDDD", but cannot input a data stream exceeding 8 words (32 bytes).

TClock and RClock

The V_R4400 has two TClock pins.

The V_R4300 has only one TClock pin to reduce the power consumption.

The V_R4400 has RClock as the reception clock of the external agent, but the V_R4300 does not have RClock because it transfers or receives data by using TClock.

Effect of RP Bit

With the V_R4400, SClock and TClock are not affected by the RP bit. The V_R4300, in contrast, can reduce the clock frequencies of SClock and TClock to the 1/4 of the normal level by using the *RP* bit*.

To use this function, if there is an external circuit (such as a DRAM refresh counter) that is affected by changes in the frequency of the clock supplied by the V_R4300 to external devices, incorporate a process that supports frequency conversion of the external circuit into the software.

★ * 100 MHz model of the V_R4300 and the V_R4305 only

Table B-2 Differences in System Design

Function \ Product Name		V_R4300	V_R4400
Initialization of processor		Set by external pins	Set by software
System interface	Bus width	32	64
	Data check	Not performed	Parity/ECC selectable
	Multi-processing and secondary cache	Not supported	Supported
	Line size of cache	Instruction: 8 words Data: 4 words	4/8 words selectable for both instruction/data cache
	Data rate	2 types	9 types
	TClock	1	2
	RClock	None	2
	Effect of RP bit	Reduces frequencies of TClock and SClock to 1/4*	Does not affect TClock and SClock

★ * 100 MHz model of the V_R4300 and the V_R4305 only

B.3 Other Differences

In addition to the above differences, the V_R4300 and V_R4400 differ in the following points. The differences described in this section are summarized in Table B-3.

B.3.1 Cache Size

The specifications of the primary cache of the V_R4000, V_R4400, and V_R4300 are shown in the following table.

Item \ Product Name		V_R4300	V_R4000	V_R4400
Cache capacity	Instruction	16 KB	8 KB	16 KB
	Data	8 KB	8 KB	16 KB
Line size		Instruction: 8 words (32 bytes) Data: 4 words (16 bytes)	4/8 words selectable	
Method		Direct map, virtual index		

To initialize or invalidate, or program each routine of flash, keep in mind the differences in cache size.

B.3.2 TLB

TLB Entry

The V_R4300 has a full-associate TLB with 32 entries. Each entry is mapped to the even/odd page of a page frame number.

The TLB of the V_R4400 is the same as that of the V_R4300 in structure, but has 48 entries.

Interaction between IMT and TLB Manipulations

The operation of the V_R4400 is undefined when the TLB instruction accesses JTLB during the instruction TLB miss (IMT) stall, and consequently, the TLB invalid exception may occur. This exception is likely to occur especially when an entry different from the one that has caused the instruction TLB miss is accessed by software for read/write manipulation (TLBWI, TLBWR, or TLBR).

This does not apply to the V_R4300.

B.3.3 Floating-Point Unit

Floating-Point Data Path

The floating-point operation of the V_R4300 is executed by using the main pipeline and data path of the integer operation unit. While a multicycle instruction of floating-point operation is executed, therefore, the pipeline of integer operation stalls.

The V_R4400 has a dedicated floating-point data path in addition to an integer data path. Therefore, if a program with the floating-point operation instruction and integer operation instruction optimized for the V_R4400 is executed with the V_R4300, not much effect can be expected.

Instruction Execution Time

The V_R4300 completely executes any multicycle instruction that has caused a source exception (exception of the source operand of an instruction) in one cycle. Instead, it issues the default result to the cycle according to the trap enable flag, or notifies occurrence of a trap exception in the next cycle. In addition, calculation such as 0 x 0 can be executed with the fewer cycles than the ordinary calculation.

The V_R4400 always executes each multicycle instruction with the same number of cycles, regardless of whether or not an exception occurs.

Cvt. [s,d] .I Instruction

When converting a 64-bit integer into a single- or double-precision floating-point number, the V_R4400 generates a floating-point unimplemented operation exception unless all the bits 63 through 52 of the integer are 0 or 1.

The V_R4300 generates the floating-point unimplemented operation exception unless all the bits 63 through 55 of a 64-bit integer are 0 or 1.

B.3.4 Pipeline

The V_R4400 uses an 8-stage super pipeline.

The V_R4300 uses a 5-stage pipeline like that of the V_R3000. The pipeline of the V_R4300 is not a super pipeline, but is not different from the super pipeline in terms of functions. However, if the program is optimized, the performance of the pipeline may be influenced.

The number of stall cycles that are generated by the V_R4300 is fewer than that of the V_R4400.

B.3.5 Interrupt

The bit 15 of the cause register of the V_R4300 is dedicated to the timer interrupt that occurs if the value of the counter register coincides with the value of the compare register. Therefore, the V_R4300 is not provided with the $\overline{\text{Int5}}$ pin that is provided to the V_R4400.

Because the V_R4300 does not have bit 5 in the interrupt register*, it does not operate even if data is written to the interrupt register via the system interface.

With the V_R4400, the user can select whether to use the timer interrupt, or the bit 5 of the interrupt register, by using the bit 15 of the cause register.

* This register cannot be directly written by the user via software.

B.3.6 Kernel Physical Address Segment Configuration

The V_R4300 supports two algorithms (uncached and non-coherent) to maintain the coherency of the cache. While the V_R4400 supports a 36-bit physical address space, the V_R4300 supports a 32-bit physical address space. These two points affect the virtual address mapping of the Kernel physical address space segment (xkphys) that does not use the TLB.

Both the V_R4400 and V_R4300 has eight address spaces in this segment, but the size of each area in these spaces is different between the V_R4400 and V_R4300. Each area in the address spaces of the V_R4400 is 64 GB, while that of the V_R4300 is 4 GB.

B.3.7 JTAG

The V_R4300 conforms to IEEE149.1-1990. Consequently, the **JTDO** signal becomes active in the shift IR and shift DR modes.

Because the V_R4400 conforms to the previous version of the IEEE149.1, the JTDO signal is not driven.

Table B-3 Other Differences

Item \ Product Name			V_R4300	V_R4000	V_R4400
Instruction cache size			16 KB	8 KB	16 KB
Data cache size			8 KB	8 KB	16 KB
TLB	TLB size		32 entries	48 entries	
	Interaction between IMT and TLB manipulations		TLB operation is corrected	TLB invalid exception occurs	
Floating-point operation	Data path		Shared with integer operation pipeline	Processed by dedicated pipeline	
	Instruction execution time		All multi-cycle instructions are executed in 1 cycle when source exception occurs.	Each multi-cycle instruction is executed in the same number of cycles regardless of whether exception occurs.	
	Cvt.[s, d].I instruction (checking of floating-point unimplemented operation exception)		All bits 63 to 55 are 1 or 0	All bits 63 to 52 are 1 or 0	
	Effect of RP bit		Reduces operating frequency to 1/4*	Does not affect operating frequency	
Pipeline			5 stages Basic pipeline	8 stages Super pipeline	
Interrupt	Cause register (bit 15)		Dedicated to timer interrupt	Selectable by user	
	Interrupt register (bit 5)		None		
Kernel physical address segment configuration (xkphys)		Physical address space supported	32 bits	36 bits	
		Valid address space	8	5	
JTAG			JTDO active in shift IR and shift DR modes	JTDO not driven in shift IR and shift DR modes	

★ * 100 MHz model of the V_R4300 and the V_R4305 only

[MEMO]

Differences from V_R4200

C

The V_R4300 is slightly different from the V_R4200 in terms of system design and software. This Appendix describes the differences between the V_R4300 and V_R4200.

The major differences are that the V_R4300 employs a new 32-bit system interface and deletes the data check function by parity.

C.1 Differences in Software

The logical differences in software are the specifications of the *CP0* registers. These differences are shown in Table C-1.

C.1.1 Cache Parity

Because the V_R4300 does not check the cache data by using a parity, the *Cache Error* register (27) always outputs 0, and writing this register is ignored. The *Parity Error* register (26) can be used for only self-diagnosis and cannot be used to manipulate the cache.

C.1.2 Status Register

The bit specifications of the *Status* registers are slightly different between the V_R4300 and V_R4200. The *CE* and *DE* bits of the *Status* register of the V_R4300 are used to manipulate the parity and do not affect the operation.

C.1.3 Config Register

The bit specifications slightly differ.

The *BE* bit and EP area of the V_R4200 set information on the external pins BigEndian and **$\overline{\text{DataRate}}$** by hardware on reset which can be read by software.

With the V_R4300, the default values are set to the *BE* bit and EP area at the time of cold reset. The default value of the EP area is 0000 and that of the *BE* bit is 1. After that, the values of these area and bit can be changed by software. Bits 18 and 19 which are 00 with the V_R4200 are 01 with the V_R4300.

For details, refer to **5.4.6 Config Register (16)**.

C.1.4 Cache Parity Error Exception

Because the V_R4300 does not check data by using the cache parity, it does not generate the parity error exception.

The V_R4200 generates the cache parity error exception (DCPE) in the WB stage.

Table C-1 Differences in Software

Function \ Product Name		V_R4300	V_R4200
Cache parity		Not supported	Supported
Status register		CE and DE bits do not function	Used to manipulate parity
Config register	BE bit and EP area	Set default values	Set information on external pins
	Bits 18 and 19	01	00

C.2 Differences in System Design

Next, the differences in system design between the V_R4300 and V_R4200 are described. Table C-2 shows these differences.

C.2.1 System Interface

The system interface of the V_R4200 is a 64-bit bus with a parity check function, but that of the V_R4300 is a 32-bit bus without a parity check function. For details, refer to **Chapter 12 System Interface**.

During block write of an instruction, the V_R4200 executes doubleword data transfer four times with one idle cycle. The V_R4300 executes word data transfer eight times to write the main memory.

During block write of data, the V_R4200 executes doubleword data transfer two times. The V_R4300 executes word data transfer four times to write the main memory.

The V_R4200 has two data rates, "DDx" and "Dxx". The V_R4300 also has two data rates, "D" and "Dxx". The V_R4200 can set a data rate by using the **$\overline{\text{DataRate}}$** pin. The data rate of the V_R4300 is set by software, by using the EP area of the config register. The table below shows the transfer data patterns in the EP area.

EP Area	Transfer Pattern
0000	D
0110	DxxDxx

C.2.2 Clock

The V_R4300 does not output the **MasterOut** and **RClock** signals.

The frequency of the pipeline clock (**PClock**) of the V_R4400 and V_R4200 is usually two times faster than **MasterClock**. The V_R4300 can change the frequency ratio by using the value of DivMode(1:0)***1** pins. (Refer to **Table 2-2 Clock/Control Interface Signals**.) The frequency ratio **PClock**:**MasterClock** can be selected from 2:1, 3:1, 4:1 or 3:2***2**. The V_R4200 usually generates **SClock** and **TClock** by dividing **PClock** by 2. The **PClock** of the V_R4300 is usually at the same frequency as **MasterClock**.

★ In the low power mode***3**, the speeds of **PClock**, **SClock**, and **TClock** of the V_R4300 can be reduced to the 1/4 of the normal level like the V_R4200.

***1.** In V_R4300 and V_R4305. In V_R4310, DivMode(2:0).

***2.** In V_R4300. In V_R4305, the frequency ratio can be set to 1:1, 2:1, or 3:1. In V_R4310, it can be set to 2:1, 3:1 4:1, 5:1, 6:1, or 5:2.

***3.** 100 MHz model of the V_R4300 and the V_R4305 only

C.2.3 Package

The V_R4200 employs a 208-pin plastic QFP. The V_R4300 is housed in a 120-pin plastic QFP.

Table C-2 Differences in System Design

Function \ Product Name		V_R4300	V_R4200
System interface	SysAD bus	No parity, 32 bits	With parity, 64 bits
	Instruction block write	Word data, 8 times	Doubleword data, 4 times
	Data block write	Word data, 4 times	Doubleword data, 2 times
	Data pattern	Set by config register (D, Dxx)	Set by external pins (DDx, Dxx)
Clock	MasterOut, RClock	Not output	Output
	PClock	Frequency ratio to MasterClock variable	Frequency two times higher than normal MasterClock
	TClock	Same frequency as normal MaterClock	PClock divided by two
Package		120-pin plastic QFP	208-pin plastic QFP

C.3 Other Differences

In addition to the above differences, the V_R4300 and V_R4200 differ in the following points. The differences described in this section are summarized in Table C-3.

C.3.1 Physical Address

The physical address and address space of the V_R4200 are 33 bits wide, and those of the V_R4300 are 32 bits wide. Consequently, the tag of the cache and the page frame number area of the TLB entry are 20 bits each at Hi and Lo sides.

C.3.2 Write Buffer

The write buffer of the V_R4200 is a doubleword buffer with two entries. The V_R4300 has a 4-entry word buffer to improve the performance during uncache write.

C.3.3 Reset

The V_R4200 simultaneously asserts the **$\overline{\text{ColdReset}}$** and **$\overline{\text{Reset}}$** signals active. These signals of the V_R4300 need not to be asserted active at the same time. The **$\overline{\text{Reset}}$** signal of the V_R4300 may be active or inactive during cold reset. However, do not change the value of this signal during reset sequence.The **$\overline{\text{ColdReset}}$** signal of the V_R4300 needs not to be synchronized with the **MasterClock** signal.

C.3.4 Status(3:0) Pins

The **Status(3:0)** pins provided to the V_R4200 are not provided to the V_R4300.

With the V_R4300, when the ITS bit of the status register is set, an instruction cache miss occurs when a branch instruction is executed, and the branch destination address is output to **SysAD(31:0)**. However, because the V_R4300 does not have **Status(3:0)** pins, the internal status of the processor cannot be output.

Table C-3 Other Differences

Product Name / Function	**V_R4300**	**V_R4200**
Physical address	32 bits	33 bits
Write buffer	4-entry Word buffer	2-entry Doubleword buffer
$\overline{\text{ColdReset}}$ signal and **MasterClock**	Need not to be synchronized	Must be synchronized
Status (3:0) pins	Not provided	Provided

★ *Restrictions of V_R4300*

D

An unimplemented operation exception will occur in response to the execution of a type conversion instruction of the floating-point operation instruction in the following cases.

- If an overflow occurs during conversion to integer format
- If the source operand is an infinite number
- If the source operand is NaN

The type conversion instructions affected by this restriction are as follows.

CEIL.L.fmt	fd, fs	FLOOR.L.fmt	fd, fs
CEIL.W.fmt	fd, fs	FLOOR.W.fmt	fd, fs
CVT.D.fmt	fd, fs	ROUND.L.fmt	fd, fs
CVT.L.fmt	fd, fs	ROUND.W.fmt	fd, fs
CVT.S.fmt	fd, fs	TRUNC.L.fmt	fd, fs
CVT.W.fmt	fd, fs	TRUNC.W.fmt	fd, fs

E

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

NEC

Facsimile Message

From:

Name

Company

Tel. FAX

Address

Although NEC has taken all possible steps to ensure that the documentation supplied to our customers is complete, bug free and up-to-date, we readily accept that errors may occur. Despite all the care and precautions we've taken, you may encounter problems in the documentation. Please complete this form whenever you'd like to report errors or suggest improvements to us.

Thank you for your kind support.

North America
NEC Electronics Inc.
Corporate Communications Dept.
Fax: 1-800-729-9288
1-408-588-6130

Europe
NEC Electronics (Europe) GmbH
Technical Documentation Dept.
Fax: +49-211-6503-274

South America
NEC do Brasil S.A.
Fax: +55-11-6465-6829

Hong Kong, Philippines, Oceania
NEC Electronics Hong Kong Ltd.
Fax: +852-2886-9022/9044

Korea
NEC Electronics Hong Kong Ltd.
Seoul Branch
Fax: 02-528-4411

Taiwan
NEC Electronics Taiwan Ltd.
Fax: 02-2719-5951

Asian Nations except Philippines
NEC Electronics Singapore Pte. Ltd.
Fax: +65-250-3583

Japan
NEC Semiconductor Technical Hotline
Fax: 044-435-9608

I would like to report the following error/make the following suggestion:

Document title: _____

Document number: _____ Page number: _____

If possible, please fax the referenced page or drawing.

Document Rating	Excellent	Good	Acceptable	Poor
Clarity	❑	❑	❑	❑
Technical Accuracy	❑	❑	❑	❑
Organization	❑	❑	❑	❑