NEC

Preliminary User's Manual

VR5500™

64/32-Bit Microprocessor

μPD30550

Document No. U16044EJ1V0UM00 (1st edition)
Date Published August 2002 N CP(K)

© NEC Corporation 2002
© MIPS Technologies, Inc. 2001
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.

VR Series, VR4000, VR4000 Series, VR4100 Series, VR4200, VR4300 Series, VR4400, VR5000, VR5000 Series, VR5000A, VR5432, VR5500, and VR10000 are trademarks of NEC Corporation.
MIPS is a registered trademark of MIPS Technologies, Inc. in the United States.
MC68000 is a trademark of Motorola Inc.
IBM370 is a trademark of IBM Corp.
Pentium is a trademark of Intel Corp.
DEC VAX is a trademark of Digital Equipment Corporation.
UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Ltd.

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

- **The information contained in this document is being issued in advance of the production cycle for the device. The parameters for the device may change before final production or NEC Corporation, at its own discretion, may withdraw the device prior to its production.**
- **Not all devices/types available in every country. Please check with local NEC representative for availability and additional information.**
- No part of this document may be copied or reproduced in any form or by any means without the prior written consent of NEC Corporation. NEC Corporation assumes no responsibility for any errors which may appear in this document.
- NEC Corporation does not assume any liability for infringement of patents, copyrights or other intellectual property rights of third parties by or arising from use of a device described herein or any other liability arising from use of such device. No license, either express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC Corporation or others.
- 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 the customer's equipment shall be done under the full responsibility of the customer. NEC Corporation assumes no responsibility for any losses incurred by the customer or third parties arising from the use of these circuits, software, and information.
- While NEC Corporation has been making continuous effort to enhance the reliability of its semiconductor devices, the possibility of defects cannot be eliminated entirely. To minimize risks of damage or injury to persons or property arising from a defect in an NEC semiconductor device, customers must incorporate sufficient safety measures in its design, such as redundancy, fire-containment, and anti-failure features.
- NEC devices are classified into the following three quality grades:
  "Standard", "Special", and "Specific". The Specific quality grade applies only to devices developed based on a customer designated "quality assurance program" for a specific application. The recommended applications of a device depend on its quality grade, as indicated below. Customers must check the quality grade of each device before using it in a particular application.

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

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

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

  The quality grade of NEC devices is "Standard" unless otherwise specified in NEC's Data Sheets or Data Books. If customers intend to use NEC devices for applications other than those specified for Standard quality grade, they should contact an NEC sales representative in advance.

M5D 98. 12

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

NEC Electronics (Europe) GmbH
Duesseldorf, Germany
Tel: 0211-65 03 01
Fax: 0211-65 03 327

- **Sucursal en España**
  Madrid, Spain
  Tel: 091-504 27 87
  Fax: 091-504 28 60

- **Succursale Française**
  Vélizy-Villacoublay, France
  Tel: 01-30-67 58 00
  Fax: 01-30-67 58 99

- **Filiale Italiana**
  Milano, Italy
  Tel: 02-66 75 41
  Fax: 02-66 75 42 99

- **Branch The Netherlands**
  Eindhoven, The Netherlands
  Tel: 040-244 58 45
  Fax: 040-244 45 80

- **Branch Sweden**
  Taeby, Sweden
  Tel: 08-63 80 820
  Fax: 08-63 80 388

- **United Kingdom Branch**
  Milton Keynes, UK
  Tel: 01908-691-133
  Fax: 01908-670-290

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 Shanghai, Ltd.
Shanghai, P.R. China
Tel: 021-6841-1138
Fax: 021-6841-1137

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

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

J02.4

INTRODUCTION

Readers

This manual is intended for users who wish to understand the functions of the VR5500 (μPD30550) and to develop application systems using this microprocessor.

Purpose

This manual introduces the architecture and hardware functions of the VR5500 to users, following the organization described below.

Organization

This manual consists of the following contents.

- Introduction
- Pipeline operation
- Cache organization and memory management system
- Exception processing
- Floating-point unit operation
- Hardware
- Instruction set details

How to read this manual

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

The VR4400™ in this manual includes the VR4000™.
The VR4000 Series™ in this document indicates the VR4100 Series™, VR4200™,
VR4300 Series™, and VR4400.

To learn in detail about the function of a specific instruction,
→ Read **CHAPTER 3 OUTLINE OF INSTRUCTION SET**, **CHAPTER 7 FLOATING-POINT UNIT, CHAPTER 17 CPU INSTRUCTION SET**, and **CHAPTER 18 FPU INSTRUCTION SET**.

To know about the overall functions of the VR5500:
→ Read this manual in the order of the contents.

To know about electrical specifications of the VR5500:
→ Refer to **Data Sheet** which is separately available.

Conventions

Data significance: Higher digits on the left and lower digits on the right
Active low representation: XXX# (trailing # after pin and signal names)
Note: Footnote for item marked with **Note** in the text
Caution: Information requiring particular attention
Remark: Supplementary information
Numerical representation: Binary... XXXX or XXXX_2
Decimal…XXXX
Hexadecimal ... 0xXXXX

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

K (kilo)	$2^{10} = 1,024$
M (mega)	$2^{20} = 1,024^2$
G (giga)	$2^{30} = 1,024^3$
T (tera)	$2^{40} = 1,024^4$
P (peta)	$2^{50} = 1,024^5$
E (exa)	$2^{60} = 1,024^6$

Related Documents

The related documents indicated in this publication may include preliminary versions. However preliminary versions are not marked as such.

Documents Related to Devices

Document Name	Document No.
μPD30550 (VR5500) Data Sheet	To be prepared
VR5500 User's Manual	This Manual
VR5432™ User's Manual Volume 1	U13751E
VR5432 User's Manual Volume 2	U15397E
VR5000™, VR5000A™ User's Manual	U11761E
VR5000, VR10000™ Instruction User's Manual	U12754E

Application Note

Document Name	Document No.
VR Series™ Programming Guide Application Note	U10710E

CONTENTS

LIST OF FIGURES (1/5)

LIST OF FIGURES (2/5)

LIST OF FIGURES (3/5)

LIST OF FIGURES (4/5)

LIST OF FIGURES (5/5)

LIST OF TABLES (1/4)

LIST OF TABLES (2/4)

LIST OF TABLES (3/4)

LIST OF TABLES (4/4)

CHAPTER 1 GENERAL

This chapter outlines the RISC 64-/32-bit microprocessor VR5500 (μPD30550).

1.1 Features

The VR5500 is one of NEC's VR Series microprocessors. It is a high-performance 64-/32-bit microprocessor employing the RISC (Reduced Instruction Set Computer) architecture developed by MIPS™.

A bus width of 64 bits or 32 bits can be selected for the system interface of the VR5500, which operates with a protocol compatible with the VR5000 Series™ and VR5432.

The VR5500 has the following features.

- Maximum operating frequency: Internal: 400 MHz, 300 MHz, external: 133 MHz
- Internal operating frequency obtained from the external operating clock (input clock and clock for bus interface) through multiplication.

  The multiplication rate can be selected from 2, 2.5, 3, 3.5, 4, 4.5, or 5.5 at reset.
- 64-bit architecture for 64-bit data processing
- 2-way superscalar pipeline

  Parallel processing by six execution units (ALU0, ALU1, FPU, FPU/MAC, BRU, and LSU)
- Employment of out-of-order mechanism
- Branch prediction mechanism

  Branch history table with 4K entries reduces branch delay.
- Virtual address management by high-speed translation lookaside buffer (TLB) (48 double entries)
- Address space Physical: 36 bits (with 64-bit bus)

  32 bits (with 32-bit bus)

  Virtual: 40 bits (in 64-bit mode)

  31 bits (in 32-bit mode)
- Internal cache memory

  2-way set associative with line lock function

  Instruction: 32 KB

  Data: 32 KB, non-blocking structure. Write method can be selected from writeback and write through.
- 64-/32-bit address/data multiplexed bus

  The bus width is selected at reset. Compatible with the bus protocol of existing products

  64-bit bus: Compatible with bus protocol of VR5000 Series

  32-bit bus: Compatible with bus protocol of VR5432 (native mode) or RM523x[Note]

  Out-of-order return mode can be selected for each bus width.

Note Product of PMC-Sierra

- Internal transaction buffer
- Internal floating-point unit
- Hardware debug function (N-Wire)
- Conforms to MIPS I, II, III, and IV instruction sets.
  Also supports sum-of-products instructions, rotate instructions, register scan instructions, and low-power mode instructions.
- Support of standby mode to reduce power consumption during standby
- Supply voltage Core block: V_{DD} = 1.5 V ±5% (300 MHz model), 1.6 to 1.7 V (400 MHz model)
  I/O block: $V_{DD}IO$ = 3.3 V ±5%, 2.5 V ±5%

1.2 Ordering Information

Part Number	Package	Internal Maximum Operating Frequency
μPD30550F2-300-NN1	272-pin plastic BGA (C/D advanced type) (29 × 29)	300 MHz
μPD30550F2-400-NN1	272-pin plastic BGA (C/D advanced type) (29 × 29)	400 MHz

1.3 VR5500 Processor

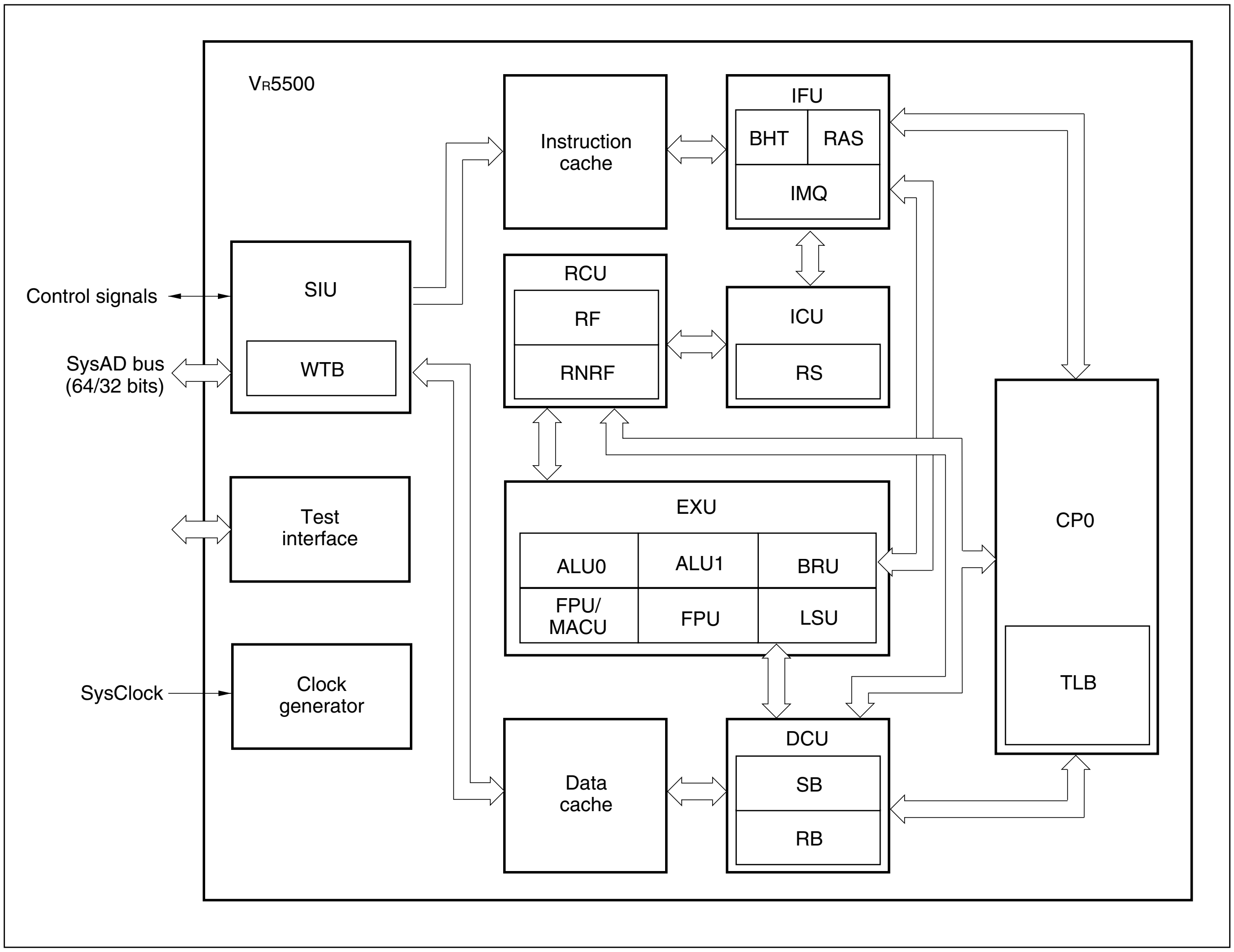
All the internal structures of the VR5500 such as the operation units, register files, and data bus, are 64 bits wide. However, the VR5500 can also execute 32-bit applications even when it operates as a 64-bit microprocessor.

The VR5500 manages instruction execution by using a 2-way superscalar, high-performance pipeline, and realizes out-of-order processing by using six execution units.

“Out-of-order” is a method that executes two or more instructions in a queue according to their execution readiness, independent of the program sequence. The hardware detects the dependency relationship of registers and delay due to load/branch, and locates and processes resources so that there is no gap in the pipeline. The execution result is output (i.e., written back to memory) in the program sequence.

Figure 1-1 shows the internal block diagram of the VR5500. The VR5500 consists of 11 main units.

Figure 1-1. Internal Block Diagram



1.3.1 Internal block configuration

(1) Instruction cache

The instruction cache uses a 2-way set associative, virtual index, physical tag system and enables line-locking. The capacity is 32 KB. The cache is replaced by the LRU (Least Recently Used) method. The line size is 32 bytes (8 words).

(2) Instruction fetch unit (IFU)

This unit fetches an instruction from the instruction cache, stores it once in an instruction management queue (IMQ) of 16 entries, and then transfers it to an instruction control unit (ICU). Up to two instructions are fetched and transferred per cycle.

The IFU also has a branch prediction mechanism and a branch history table (BHT) of 4096 entries so that instructions can continue to be fetched speculatively. Moreover, one return address stack (RAS) entry is provided so that exiting from a subroutine is speculatively processed.

(3) Instruction control unit (ICU)

This unit controls out-of-order execution of instructions. It renames registers to reduce the hazards caused by the dependency relationship of registers, when an instruction is transferred from the IFU. The instruction is then stored in a reservation station (RS) of 20 entries until it is ready for execution. When execution is ready, up to two instructions are taken out from the RS and are transferred to the execution unit (EXU).

(4) Register control unit (RCU)

This unit has a register file (RF) and a renaming register file (RNRF). RF consists of sixty-four 64-bit registers, and RNRF consists of sixteen 64-bit registers. These registers serve as source and destination registers when an instruction is executed. When instruction execution is complete, the RCU transfers the contents of RNRF to RF in accordance with the renaming by the ICU, and completes instruction execution (commits). Up to three instructions can be committed per cycle.

(5) Execution unit (EXU)

This unit consists of the following six sub-units.

- ALU0: 64-bit integer operation unit
- ALU1: 64-bit integer operation unit
- FPU/MAC: 64-/32-bit floating-point operation unit and sum-of-products operation unit (floating-point multiplication and sum-of-products operations, integer multiplication, sum-of-products, and division operations)
- FPU: 64-/32-bit floating-point operation unit
- BRU: Branch unit
- LSU: Load/store unit

(6) Data cache control unit (DCU)

This unit controls transactions to the data cache and replacement of cache lines. It has a refill buffer (RB) and store buffer (SB) with four entries each, and can process a non-blocking cache operation of up to four accesses. The DCU also supports functions such as uncached load/store, completion of transaction in the order of issuance, and data transfer from SB to the data cache by instruction execution commitment.

(7) Data cache

The data cache uses a two-way set associative, virtual index, physical tag system, and enables line-locking. The capacity is 32 KB. The cache is replaced by the LRU (Least Recently Used) method. Write method can be selected from writeback and write through. The line size is 32 bytes (8 words).

(8) Coprocessor 0 (CP0)

CP0 manages memory, processes exceptions, and monitors the performance.

For memory management, it protects access to various operation modes (user, supervisor, and kernel), memory segments, and memory pages.

Virtual addresses are translated by a translation lookaside buffer (TLB). The TLB is a full-associative type and has 48 entries. Each entry can be mapped in page sizes of 4 KB to 1 GB.

The coprocessor performs control when an interrupt or exception occurs as exception processing.

It counts the number of times an event has occurred to check if instruction execution is efficient in order to monitor the performance.

(9) System interface unit (SIU)

The SysAD bus realizes interfacing with an external agent. This bus is a 64-/32-bit address/data multiplexed bus and is compatible with the VR5000 Series.

To enhance the bus efficiency, four 64-bit write transaction buffers (WTBs) are provided.

The SIU also supports an uncached accelerated store operation, so that consecutive single write accesses are combined into one block write access.

(10) Clock generator

The clock generator generates a clock for the pipeline from an externally input clock. The frequency ratio can be selected from 1:2, 1:2.5, 1:3, 1:3.5, 1:4, 1:4.5, 1:5, and 1:5.5.

(11) Test interface

This interface connects an external debugging tool. It conforms to the N-Wire specification and controls testing and debugging of the processor by using JTAG interface signals and debug interface signals.

1.3.2 CPU registers

The VR5500 has the following registers.

- General-purpose registers (GPR): 64 bits × 32

In addition, the processor provides the following special registers.

- PC: Program Counter (64 bits)
- HI register: Contains the integer multiply and divide higher doubleword result (64 bits)
- LO register: Contains the integer multiply and divide lower doubleword result (64 bits)

Two of the general-purpose registers have assigned the following functions.

- r0: Since it is fixed to zero, it 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: The link register used by the JAL/JALR instruction. This register can be used for other instructions. However, be careful that use of the register by the JAL/JALR instruction does not coincide with use of the register for other operations.

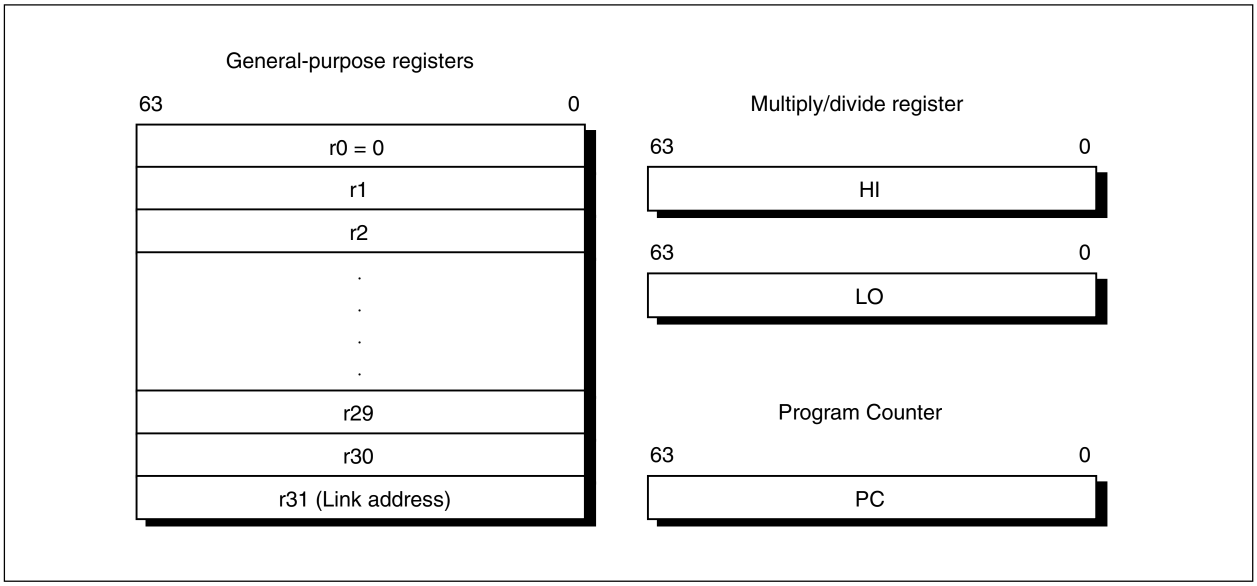
The register group is provided in the CP0 (system control coprocessor), to process exceptions and to manage addresses and in the FPU (floating-point unit) used for the floating-point operation.

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

The operation of the CPU register differs depending on what instructions are executed: 32-bit instructions or MIPS16 instructions.

Figure 1-2 shows the CPU registers.

Figure 1-2. CPU Registers



The VR5500 has no Program Status Word (PSW) register; this is covered by the Status and Cause registers incorporated within the system control coprocessor (CP0). For details of the CP0 registers, refer to **1.3.4 System control coprocessors (CP0).**

1.3.3 Coprocessors

ISA of MIPS defines that up to four coprocessors (CP0 to CP3) can be used. Of these, CP0 is defined as a system control coprocessor, and CP1 is defined as a floating-point unit. CP2 and CP3 are reserved for future expansion.

1.3.4 System control coprocessors (CP0)

CP0 translates virtual addresses to physical addresses, switches the operating mode (kernel, supervisor, or user mode), and manages exceptions. It also controls the cache subsystem to analyze a cause and to return from the error state.

Table 1-1 shows a list of the CP0 registers. For details of the registers related to the virtual system memory, refer to **CHAPTER 5 MEMORY MANAGEMENT SYSTEM**. For details of the registers related to exception handling, refer to **CHAPTER 6 EXCEPTION PROCESSING**.

Table 1-1. CP0 Registers

Register Number	Register Name	Usage	Description
0	Index	Memory management	Programmable pointer to TLB array
1	Random	Memory management	Pseudo-random pointer to TLB array (read only)
2	EntryLo0	Memory management	Lower half of TLB entry for even VPN
3	EntryLo1	Memory management	Lower half of TLB entry for odd VPN
4	Context	Exception processing	Pointer to virtual PTE table in 32-bit mode
5	PageMask	Memory management	Page size specification
6	Wired	Memory management	Number of wired TLB entries
7	–	–	Reserved
8	BadVAddr	Memory management	Display of virtual address where the most recent error occurred
9	Count	Exception processing	Timer count
10	EntryHi	Memory management	Higher half of TLB entry (including ASID)
11	Compare	Exception processing	Timer compare value
12	Status	Exception processing	Operation status setting
13	Cause	Exception processing	Display of cause of the most recent exception occurred
14	EPC	Exception processing	Exception program counter
15	PRId	Memory management	Processor revision ID
16	Config	Memory management	Memory system mode setting
17	LLAddr	Memory management	Display of address of the LL instruction
18	WatchLo	Exception processing	Memory reference trap address lower bits
19	WatchHi	Exception processing	Memory reference trap address higher bits
20	XContext	Exception processing	Pointer to virtual PTE table in 64-bit mode
21 to 24	–	–	Reserved
25	Performance Counter	Exception processing	Count and control of performances
26	Parity Error	Exception processing	Cache parity bits
27	Cache Error	Exception processing	Cache error and status register
28	TagLo	Memory management	Lower half of cache tag
29	TagHi	Memory management	Higher half of cache tag
30	ErrorEPC	Exception processing	Error exception program counter
31	–	–	Reserved

1.3.5 Floating-point unit

The floating-point unit (FPU) executes floating-point operations. The FPU of the VR5500 conforms to ANSI/IEEE Standard 754-1985 “IEEE2 Floating-Point Operation Standard”.

The FPU can perform an operation with both single-precision (32-bit) and double-precision (64-bit) values.

The FPU has the following registers.

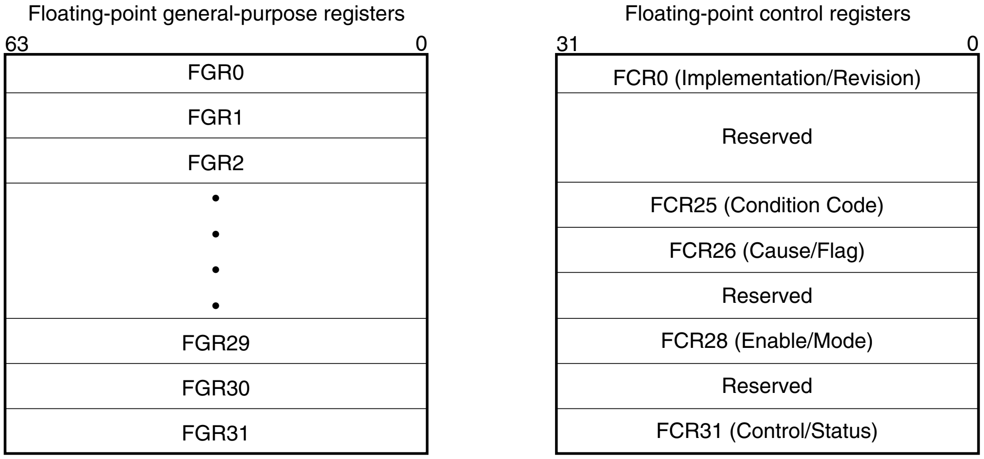
- Floating-point general-purpose register (FGR): 64/32 bits × 32
- Floating-point control register (FCR): 32 bits × 32

The number of bits of the FGR can be changed depending on the setting of the FR bit of the Status register in CP0. If the number of bits is set to 32, sixteen 64-bit FGRs can be used for floating-point operations. If it is set to 64 bits, thirty-two 64-bit registers can be used.

Of the 32 FCRs, only five can be used.

Figure 1-3 shows the FPU registers.

Figure 1-3. FPU Registers



Like the CPU, the FPU uses an instruction set with a load/store architecture. A floating-point operation can be started in each cycle. The load instructions of the FPU include R-type instructions.

For details of the FPU, refer to **CHAPTER 7 FLOATING-POINT UNIT** and **CHAPTER 8 FLOATING-POINT EXCEPTIONS**.

1.3.6 Cache memory

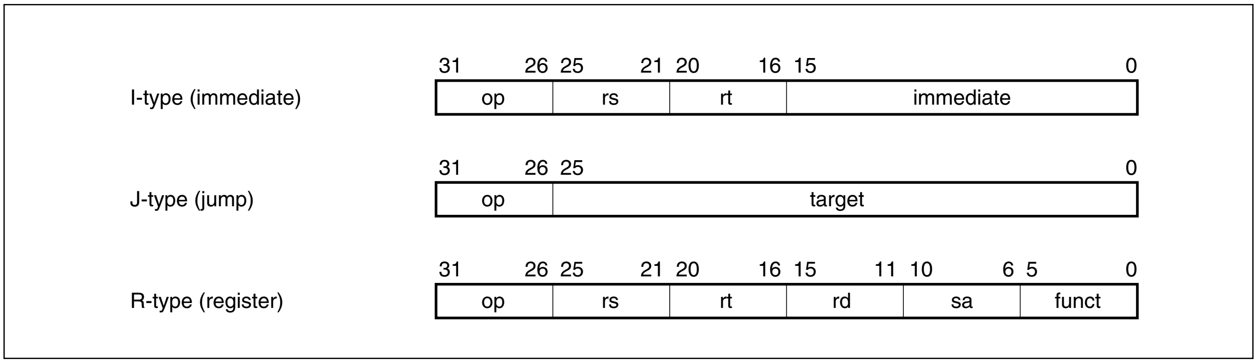
The VR5500 has an internal instruction cache and data cache to enhance the efficiency of the pipeline. The instruction cache and data cache can be accessed in parallel. Both the instruction cache and data cache have a data width of 64 bits and a capacity of 32 KB, and are managed by a two-way set associative method.

For details of the caches, refer to **CHAPTER 11 CACHE MEMORY**.

1.4 Outline of Instruction Set

All the instructions are 32 bits long. The instructions come in three types as shown in Figure 1-4: immediate (I-type), jump (J-type), and register (R-type).

Figure 1-4. Instruction Type



The instructions are classified into the following six groups.

(1) Load/store instructions transfer data between memory and a general-purpose register. Most of these instructions are I-type. The addressing mode is in the format in which a 16-bit signed offset is added to the base register. Some load/store instructions are index-type instructions that use floating-point registers (R-type).

(2) Arithmetic operation instructions execute an arithmetic operation, logical operation, shift manipulation, or multiplication/division on register values. The instruction types of these instructions are R-type (both the operand and the result of the operation are stored in registers) and I-type (one of the operands is a 16-bit signed immediate value).

(3) Jump/branch instructions change the flow of program control. A jump instruction jumps to an address that is generated by combining a 26-bit target address and the higher bits of the program counter (J-type), or to an address indicated by a register (R-type). A branch instruction branches to a 16-bit offset address relative to the program counter (I-type). The Jump and Link instruction saves the return address to register 31.

(4) Coprocessor instructions execute the operations of the coprocessor. The load and store instructions of the coprocessor are I-type instructions. The format of the operation instruction of a coprocessor differs depending on the coprocessor (refer to **CHAPTER 7 FLOATING-POINT UNIT**).

(5) System control coprocessor instructions execute operations on the CP0 register to manage the memory of the processor and to process exceptions.

(6) Special instructions execute system call exceptions and breakpoint exceptions. In addition, they branch to a general-purpose exception processing vector depending on the result of comparison. The instruction types are R-type and I-type.

For each instruction, refer to **CHAPTER 3 OUTLINE OF INSTRUCTION SET**, **CHAPTER 17 CPU INSTRUCTION SET**, and **CHAPTER 18 FPU INSTRUCTION SET**.

1.5 Data Format and Addressing

The VR5500 has the following four types of data formats.

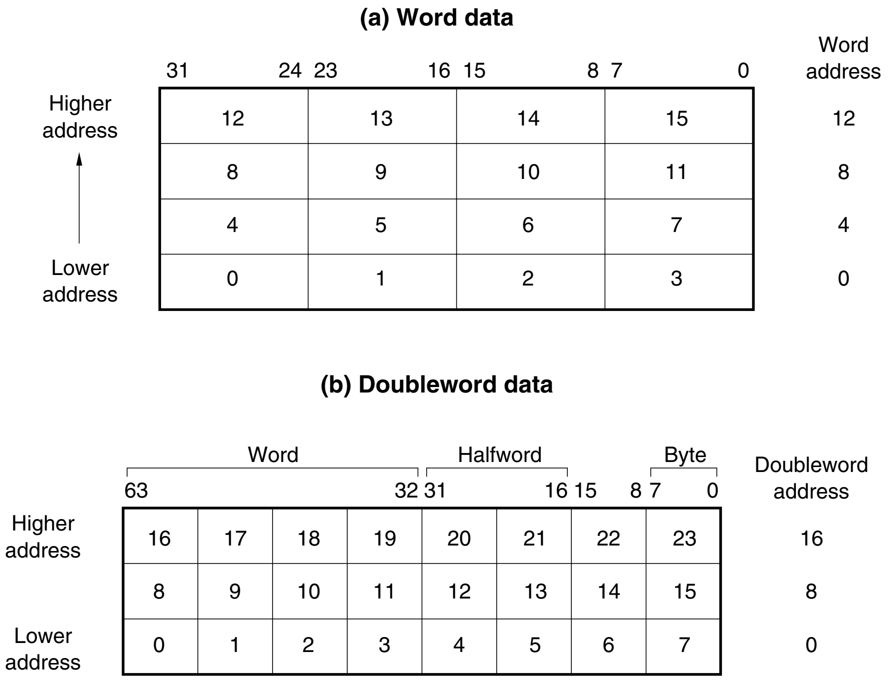
Doubleword (64 bits)
Word (32 bits)
Halfword (16 bits)
Byte (8 bits)

If the data format is doubleword, word, or halfword, the byte order can be set to big endian or little endian by using the BigEndian pin at reset.

The endianness is defined by the position of byte 0 in the data structure of multiple bytes.

In a big-endian system, byte 0 is the most significant byte (leftmost byte). This byte order is compatible with that employed for MC68000™ and IBM370™. Figure 1-5 shows the configuration.

Figure 1-5. Byte Address of Big Endian



(a) Word data

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

(b) Doubleword data

	Word				Halfword			Byte	Doubleword address
	63			32	31	16	15 8	7 0	
Higher address	16	17	18	19	20	21	22	23	16
	8	9	10	11	12	13	14	15	8
Lower address	0	1	2	3	4	5	6	7	0

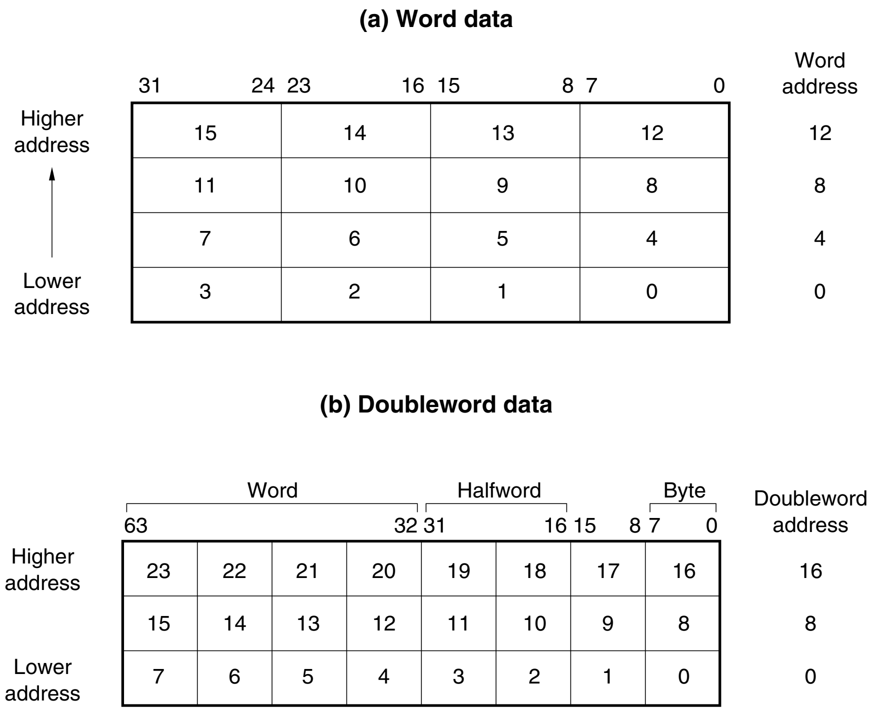
Remarks 1. The most significant byte is at the least significant address.

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

In a little-endian system, byte 0 is the least significant byte (rightmost byte). This byte order is compatible with that employed for Pentium™ and DEC VAX™. Figure 1-6 shows the configuration.

Unless otherwise specified, little endian is used in this manual.

Figure 1-6. Byte Address of Little Endian



(a) Word data

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

(b) Doubleword data

	Word 63			32	Halfword 31		16	15 8	Byte 7 0	Doubleword address
Higher address	23	22	21	20	19	18	17	16	16
	15	14	13	12	11	10	9	8	8
Lower address	7	6	5	4	3	2	1	0	0

Remarks 1. The least significant byte is at the least significant address.

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

The CPU uses the following addresses to access halfwords, words, and doublewors.

- Halfword: Even-byte boundary (0, 2, 4 …)
- Word: 4-byte boundary (0, 4, 8 …)
- Doubleword: 8-byte boundary (0, 8, 16 …)

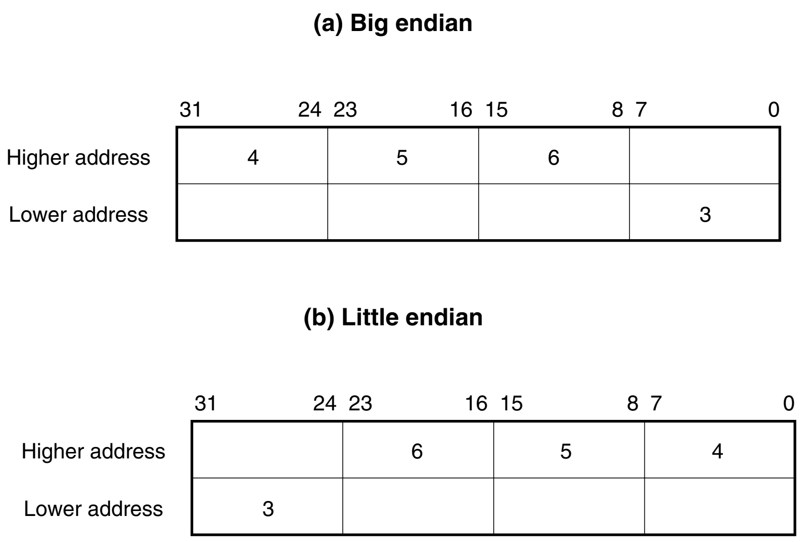
To load/store data that is not aligned at a 4-byte boundary (word) or 8-byte boundary (doubleword), the following dedicated instructions are used.

- Word: LWL, LWR, SWL, SWR
- Doubleword: LDL, LDR, SDL, SDR

These instructions are always used in pairs of L and R.

Figure 1-7 illustrates how the word at byte address 3 is accessed.

Figure 1-7. Byte Address (Unaligned Word)



(a) Big endian

	31 … 24	23 … 16	15 … 8	7 … 0
Higher address	4	5	6	
Lower address				3

(b) Little endian

	31 … 24	23 … 16	15 … 8	7 … 0
Higher address		6	5	4
Lower address	3			

1.6 Memory Management System

The VR5500 can manage a physical address space of up to 64 GB (36 bits). Most systems, however, are provided with a physical memory only in units of 1 GB or lower. Therefore, the CPU translates addresses, allocates them to a vast virtual address space, and supplies the programmer with an extended memory space.

For details of these address spaces, refer to **CHAPTER 5 MEMORY MANAGEMENT SYSTEM**.

1.6.1 High-speed translation lookaside buffer (TLB)

TLB translates a virtual address into a physical address. It is of full-associative method and has 48 entries. Each entry has consecutive two pages of mapping information. The page size can be changed from 4 KB to 1 GB in units of power of 4.

(1) Joint TLB (JTLB)

This TLB holds both instruction addresses and data addresses.

The higher bits of a virtual address (the number of bits depends on the size of the page) and a process identifier are combined and compared with each entry of JLTB. If there is no matching entry in the TLB, an exception occurs, and the entry contents are written by software from a page table on memory to the TLB. The entry is determined by the value of the Random register or Index register.

(2) Micro TLB

This TLB is for address translation in a cache. Two micro TLBs, an instruction micro TLB and a data micro TLB, are available. Each micro TLB has four entries and the contents of an entry can be loaded from the JTLB. However, loading to the micro TLB is performed internally and cannot be monitored by software.

1.6.2 Processor modes

(1) Operating mode

The VR5500 has three operating modes: user, supervisor, and kernel. The memory mapping differs depending on the operating mode. For details, refer to **CHAPTER 5 MEMORY MANAGEMENT SYSTEM**.

(2) Addressing mode

The VR5500 has two addressing modes: 32-bit and 64-bit addressing. The address translation method and memory mapping differ depending on the addressing mode. For details, refer to **CHAPTER 5 MEMORY MANAGEMENT SYSTEM**.

1.7 Instruction Pipeline

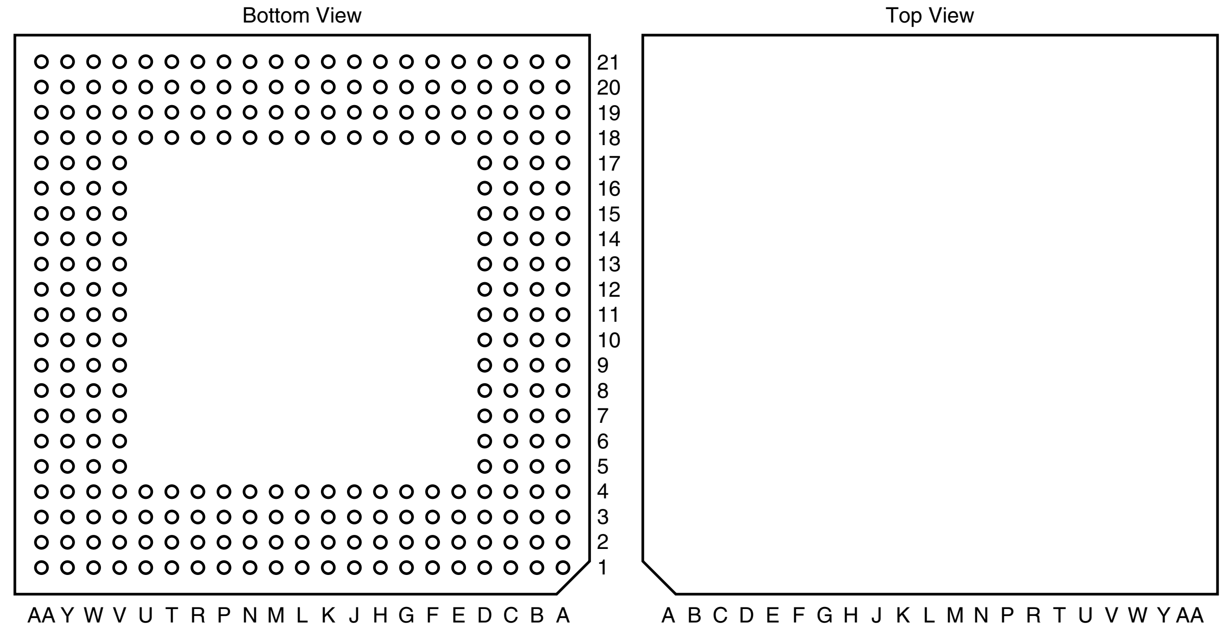
The VR5500 has an instruction pipeline of up to 10 stages. It also has a mechanism that can simultaneously execute two instructions and thus can execute a floating-point operation instruction and an instruction of another type at the same time. For details, refer to **CHAPTER 4 PIPELINE**.

1.7.1 Branch prediction

The VR5500 has an internal branch prediction mechanism that accelerates branching. The branch history is recorded in a branch history table. The branch instruction that has been fetched is executed according to this table. The subsequent instructions are speculatively processed. For operations when branch prediction hits or misses, refer to **CHAPTER 4 PIPELINE**.

2.1 Pin Configuration

- 272-pin plastic BGA (C/D advanced type) (29 × 29)


Bottom View
Top View
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
AA Y W V U T R P N M L K J H G F E D C B A
A B C D E F G H J K L M N P R T U V W Y AA

(1/2)

Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name
A1	V_{SS}	B17	SysAD27	D12	V_{SS}	H4	V_{DD}
A2	V_{SS}	B18	V_{DD}IO	D13	SysAD31	H18	V_{SS}
A3	V_{DD}IO	B19	V_{DD}IO	D14	V_{DD}	H19	V_{SS}
A4	V_{DD}IO	B20	V_{SS}	D15	SysAD60	H20	V_{SS}
A5	Reset#	B21	V_{SS}	D16	V_{SS}	H21	SysAD21
A6	PReq#	C1	V_{DD}IO	D17	SysAD26	J1	SysCmd7
A7	ValidIn#	C2	V_{DD}IO	D18	V_{SS}	J2	SysCmd8
A8	ValidOut#	C3	V_{SS}	D19	V_{SS}	J3	TIntSel
A9	V_{SS}	C4	V_{SS}	D20	V_{DD}IO	J4	Int0#
A10	SysADC7	C5	V_{SS}	D21	V_{DD}IO	J18	SysAD52
A11	SysADC3	C6	V_{DD}	E1	SysCmd0	J19	SysAD20
A12	SysADC1	C7	WrRdy#	E2	DisDValidO#	J20	SysAD51
A13	SysADC4	C8	V_{SS}	E3	DWBTrans#	J21	SysAD19
A14	SysAD62	C9	SysID1	E4	O3Return#	K1	Int1#
A15	SysAD30	C10	V_{DD}	E18	SysAD57	K2	V_{SS}
A16	SysAD28	C11	SysADC2	E19	SysAD25	K3	V_{SS}
A17	SysAD59	C12	V_{SS}	E20	SysAD56	K4	V_{SS}
A18	V_{DD}IO	C13	SysAD63	E21	SysAD24	K18	V_{DD}
A19	V_{DD}IO	C14	V_{DD}	F1	SysCmd1	K19	V_{DD}
A20	V_{SS}	C15	SysAD29	F2	V_{SS}	K20	V_{DD}
A21	V_{SS}	C16	V_{SS}	F3	V_{SS}	K21	V_{DD}
B1	V_{SS}	C17	SysAD58	F4	V_{SS}	L1	Int2#
B2	V_{SS}	C18	V_{DD}IO	F18	V_{DD}	L2	Int3#
B3	V_{DD}IO	C19	V_{SS}	F19	V_{DD}	L3	Int4#
B4	V_{DD}IO	C20	V_{DD}IO	F20	V_{DD}	L4	Int5#
B5	ColdReset#	C21	V_{DD}IO	F21	SysAD55	L18	SysAD17
B6	Release#	D1	V_{DD}IO	G1	SysCmd2	L19	SysAD49
B7	ExtRqst#	D2	V_{DD}IO	G2	SysCmd3	L20	SysAD18
B8	BusMode	D3	V_{SS}	G3	SysCmd4	L21	SysAD50
B9	SysID2	D4	V_{SS}	G4	SysCmd5	M1	RMode#/BKTGIO#
B10	V_{DD}	D5	IC	G18	SysAD23	M2	V_{DD}
B11	SysADC6	D6	V_{DD}	G19	SysAD54	M3	V_{DD}
B12	V_{SS}	D7	RdRdy#	G20	SysAD22	M4	V_{DD}
B13	SysADC0	D8	V_{SS}	G21	SysAD53	M18	V_{SS}
B14	V_{DD}	D9	SysID0	H1	SysCmd6	M19	V_{SS}
B15	SysAD61	D10	V_{DD}	H2	V_{DD}	M20	V_{SS}
B16	V_{SS}	D11	SysADC5	H3	V_{DD}	M21	V_{SS}

Caution Leave the IC pin open.

Remark # indicates active low.

(2/2)

Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name	Pin No.	Pin Name
N1	V_{DD}IO	T21	SysAD12	W2	V_{DD}IO	Y12	V_{DD}
N2	NMI#	U1	NTrcClk	W3	V_{SS}	Y13	SysAD3
N3	V_{DD}IO	U2	NTrcData0	W4	V_{SS}	Y14	V_{SS}
N4	BigEndian	U3	NTrcData1	W5	V_{DD}PA2	Y15	SysAD37
N18	SysAD15	U4	NTrcData3	W6	V_{SS}	Y16	SysAD39
N19	SysAD47	U18	SysAD10	W7	V_{DD}IO	Y17	SysAD40
N20	SysAD16	U19	SysAD42	W8	V_{DD}	Y18	V_{DD}IO
N21	SysAD48	U20	SysAD11	W9	JTDI	Y19	V_{DD}IO
P1	V_{SS}	U21	SysAD43	W10	V_{SS}	Y20	V_{SS}
P2	V_{SS}	V1	NTrcData2	W11	SysAD1	Y21	V_{SS}
P3	V_{SS}	V2	NTrcEnd	W12	V_{DD}	AA1	V_{SS}
P4	V_{SS}	V3	V_{SS}	W13	SysAD35	AA2	V_{SS}
P18	V_{DD}	V4	V_{SS}	W14	V_{SS}	AA3	V_{DD}IO
P19	V_{DD}	V5	V_{SS}PA2	W15	SysAD38	AA4	V_{DD}IO
P20	V_{DD}	V6	V_{SS}	W16	V_{DD}	AA5	V_{DD}PA1
P21	SysAD46	V7	V_{DD}IO	W17	SysAD9	AA6	V_{DD}IO
R1	DivMode0	V8	V_{DD}	W18	V_{SS}	AA7	IC
R2	DivMode1	V9	JTMS	W19	V_{SS}	AA8	JTDO
R3	DivMode2	V10	V_{SS}	W20	V_{DD}IO	AA9	DrvCon
R4	V_{DD}IO	V11	SysAD33	W21	V_{DD}IO	AA10	V_{SS}
R18	SysAD44	V12	V_{DD}	Y1	V_{SS}	AA11	SysAD0
R19	SysAD13	V13	SysAD4	Y2	V_{SS}	AA12	SysAD2
R20	SysAD45	V14	V_{SS}	Y3	V_{DD}IO	AA13	SysAD34
R21	SysAD14	V15	SysAD7	Y4	V_{DD}IO	AA14	SysAD36
T1	V_{DD}	V16	V_{DD}	Y5	V_{SS}PA1	AA15	SysAD5
T2	V_{DD}	V17	SysAD41	Y6	SysClock	AA16	SysAD6
T3	V_{DD}	V18	V_{SS}	Y7	JTRST# (V_{SS})	AA17	SysAD8
T4	V_{DD}	V19	V_{SS}	Y8	V_{DD}	AA18	V_{DD}IO
T18	V_{SS}	V20	V_{DD}IO	Y9	JTCK	AA19	V_{DD}IO
T19	V_{SS}	V21	V_{DD}IO	Y10	V_{SS}	AA20	V_{SS}
T20	V_{SS}	W1	V_{DD}IO	Y11	SysAD32	AA21	V_{SS}

Caution Leave the IC pin open.

Remarks 1. Inside the parentheses indicates the pin name in Ver. 1.x.

2. # indicates active low.

Pin Identification

BigEndian:	Big endian	PReq#:	Processor request
BKTGIO#:	Break/trigger input/output	RdRdy#:	Read ready
BusMode:	Bus mode	Release#:	Release
ColdReset#:	Cold reset	Reset#:	Reset
DisDValidO#:	Disable delay ValidOut#	SysAD(63:0):	System address/data bus
DivMode(2:0):	Divide mode	SysADC(7:0):	System address/data check bus
DrvCon:	Driver control		
DWBTrans#:	Doubleword block transfer	SysClock:	System clock
ExtRqst#:	External request	SysCmd(8:0):	System command/data identifier bus
IC:	Internally connected		
Int(5:0)#:	Interrupt	SysID(2:0):	System bus identifier
JTCK:	JTAG clock	TIntSel:	Timer interrupt selection
JTDI:	JTAG data input	ValidIn#:	Valid input
JTDO:	JTAG data output	ValidOut#:	Valid output
JTMS:	JTAG mode select	V_{DD}:	Power supply for CPU core
JTRST#:	JTAG reset	V_{DD}IO:	Power supply for I/O
NMI#:	Non-maskable interrupt	V_{DD}PA1, V_{DD}PA2:	Quiet V_{DD} for PLL
NTrcClk:	N-Trace clock	V_{SS}:	Ground
NTrcData(3:0):	N-Trace data output	V_{SS}PA1, V_{SS}PA2:	Quiet V_{SS} for PLL
NTrcEnd:	N-Trace end	WrRdy#:	Write ready
O3Return#:	Out-of-Order Return mode		

Remark   # indicates active low.

2.2 Pin Functions

Remark # indicates active low.

2.2.1 System interface signals

These signals are used when the VR5500 is connected to an external device in the system. Table 2-1 shows the functions of these signals.

Table 2-1. System Interface Signals

Pin Name	I/O	Function
SysAD(63:0)	I/O	System address/data bus This is a 64-bit bus that establishes communication between the processor and external agent. The lower 32 bits (SysAD(31:0)) of this bus are used in the 32-bit bus mode.
SysADC(7:0)	I/O	System address/data check bus This is a parity bus for the SysAD bus. It is valid only in the data cycle. The lower 4 bits (SysADC(3:0)) are used in the 32-bit bus mode.
SysCmd(8:0)	I/O	System command/data ID bus This is a 9-bit bus that transfers commands and data identifiers between the processor and external agent.
SysID(2:0)	I/O	System bus protocol ID These signals transfer a request identifier in the out-of-order return mode. The processor drives the valid identifier when the ValidOut# signal is asserted. The external agent must drive the valid identifier when the ValidIn# signal is asserted.
ValidIn#	Input	Valid in This signal indicates that the external agent is driving a valid address or data onto the SysAD bus or a valid command or data identifier onto the SysCmd bus, or a valid request identifier onto the SysID bus in the out-of-order return mode.
ValidOut#	Output	Valid out This signal indicates that the processor is driving a valid address or data onto the SysAD bus or a valid command or data identifier onto the SysCmd bus, or a valid request identifier onto the SysID bus in the out-of-order return mode.
RdRdy#	Input	Read ready This signal indicates that the external agent is ready to acknowledge a processor read request.
WrRdy#	Input	Write ready This signal indicates that the external agent is ready to acknowledge a processor write request.
ExtRqst#	Input	External request This signal is used by the external agent to request the right to use the system interface.
Release#	Output	Release interface This signal indicates that the processor releases the system interface to the slave status.
PReq#	Output	Processor request This signal indicates that the processor has a pending request.

2.2.2 Initialization interface signals

These signals are used by the external device to initialize the operation parameters of the processor. Table 2-2 shows the functions of these signals.

Table 2-2. Initialization Interface Signals (1/2)

Pin Name	I/O	Function
DivMode(2:0)	Input	Division mode These signals set the division ratio of PClock and SysClock. 111: Divided by 5.5 110: Divided by 5 101: Divided by 4.5 100: Divided by 4 011: Divided by 3.5 010: Divided by 3 001: Divided by 2.5 000: Divided by 2 Set the level of these signals before starting a power-on reset, and make sure that the level does not change during operation.
BigEndian	Input	Endian mode This signal sets the byte order for addressing. 1: Big endian 0: Little endian Set the level of these signals before starting a power-on reset, and make sure that the level does not change during operation.
BusMode	Input	Bus mode This signal sets the bus width of the system interface. 1: 64 bits 0: 32 bits Set the level of these signals before starting a power-on reset, and make sure that the level does not change during operation.
TIntSel	Input	Interrupt source select This signal sets the interrupt source to be allocated to the IP7 bit of the Cause register. 1: Timer interrupt 0: Int5# input and external write request (SysAD5) Set the level of these signals before starting a power-on reset, and make sure that the level does not change during operation.
DisDValidO#	Input	ValidOut# delay enable 1: ValidOut# is active even while address cycle is stalled. 0: ValidOut# is active only in the address issuance cycle. Set the level of these signals before starting a power-on reset, and make sure that the level does not change during operation.

Remark 1: High level, 0: Low level

Table 2-2. Initialization Interface Signals (2/2)

Pin Name	I/O	Function
DWBTrans#	Input	Doubleword block transfer enable (valid only in 32-bit mode) 1: Disabled 0: Enabled Set the level of these signals before starting a power-on reset, and make sure that the level does not change during operation.
O3Return#	Input	Out-of-order return mode This signal sets the protocol of the system interface. 1: Normal mode 0: Out-of-order return mode Set the level of these signals before starting a power-on reset, and make sure that the level does not change during operation.
ColdReset#	Input	Cold reset This signal completely initializes the internal status of the processor. Deassert this signal in synchronization with SysClock.
Reset#	Input	Reset This signal logically initializes the internal status of the processor. Deassert this signal in synchronization with SysClock.
DrvCon	Input	Drive control This signal sets the impedance of the external output driver. 1: Weak 0: Normal (recommended) Set the level of these signals before starting a power-on reset, and make sure that the level does not change during operation. **Remark** This signal is used in Ver. 2.0 or later. It is fixed to 0 in Ver. 1.x.

Remark 1: High level, 0: Low level

The O3Return#, DWBTrans#, DisDValidO#, and BusMode signals are used to determine the protocol of the system interface. These signals select the protocol as follows.

Protocol	O3Return#	DWBTrans#	DisDValidO#	BusMode
VR5000-compatible	1	1	1	1
RM523x-compatible	1	1	1	0
VR5432 native mode-compatible	1	0	0	0
Out-of-order return mode	0	Any	Any	Any

Remark 1: High level, 0: Low level
RM523x is a product of PMC-Sierra.

2.2.3 Interrupt interface signals

The external device uses these signals to send an interrupt request to the VR5500. Table 2-3 shows the functions of these signals.

Table 2-3. Interrupt Interface Signals

Pin Name	I/O	Function
Int(5:0)#	Input	Interrupt These are general-purpose processor interrupt requests. The input status of these signals can be checked by the Cause register. Whether Int5# is acknowledged is determined by the status of the TIntSel signal at reset.
NMI#	Input	Non-maskable interrupt This is an interrupt request that cannot be masked.

2.2.4 Clock interface signals

These signals are used to supply or manage the clock. Table 2-4 shows the functions of these signals.

Table 2-4. Clock Interface Signals

Pin Name	I/O	Function
SysClock	Input	System clock Clock signal input to the processor.
VDDPA1 VDDPA2	–	VDD for PLL Power supply for the internal PLL.
VSSPA1 VSSPA2	–	VSS for PLL Ground for the internal PLL.

2.2.5 Power supply

Table 2-5. Power Supply

Pin Name	I/O	Function
VDD	–	Power supply pin for core
VDDIO	–	Power supply pin for I/O
VSS	–	Ground pin

Caution The VR5500 uses two power supplies. Power can be applied to these power supplies in any order. However, do not allow a voltage to be applied to only one of the power supplies for 100 ms or more.

2.2.6 Test interface signal

These signals are used to test the VR5500. They include the JTAG interface signals conforming to IEEE Standard 1149.1 and debug interface signals conforming to the N-Wire specifications. Table 2-6 shows the function of these signals.

Table 2-6. Test Interface Signals

Pin Name	I/O	Function
NTrcData(3:0)	Output	Trace data Trace data output.
NTrcEnd	Output	Trace end This signal delimits (indicates the end of) a trace data packet.
NTrcClk	Output	Trace clock This clock is for the test interface. The same clock as SysClock is output.
RMode#/ BKTGIO#	I/O	Reset mode/break trigger I/O This pin inputs a debug reset mode signal while the JTRST# signal (ColdReset# signal in Ver. 1.x) is active. It inputs/outputs a break or trigger signal during normal operation.
JTDI	Input	JTAG data input Serial data input for JTAG.
JTDO	Output	JTAG data output Serial data output for JTAG. This signal is output at the falling edge of JTCK.
JTMS	Input	JTAG mode select This signal selects the JTAG test mode.
JTCK	Input	JTAG clock input This is a serial clock input signal for JTAG. The maximum frequency is 33 MHz. It is not necessary to synchronize this signal with SysClock.
JTRST#	Input	JTAG reset input This signal is used to initialize the JTAG test module. **Remark** Only Ver. 2.0 or later

2.3 Handling of Unused Pins

2.3.1 System interface pin

(1) 32-bit bus mode

In the VR5500, the width of the SysAD bus can be selected from 64 bits or 32 bits. When the 32-bit bus mode is selected, only the necessary system interface pins are selected and used. In the 32-bit bus mode, therefore, handle the pins that are not used, as follows.

Pin	Handling
SysAD(63:32)	Leave open
SysADC(7:4)	Leave open

(2) Normal mode

The VR5500 in the out-of-order return mode can process read/write transactions regardless of the request issuance sequence. At this time, the SysID(2:0) pins are used to identify the request. These signals are not used in the normal mode and therefore must be handled as follows.

Pin	Handling
SysID(2:0)	Leave open

(3) Parity bus

The VR5500 allows selection of whether to protect data by using parity or not. When parity is used, parity data is output from the processor or external agent to the SysADC bus.

Because whether parity is used or not is selected by software, however, the VR5500 cannot determine the operation of the SysADC bus until the program is started. Therefore, make sure that the SysADC bus is not left open nor goes into a high-impedance state.

When it is known that parity will not be used in the system, it is recommended to connect each pin of the SysADC bus to V_{DD}IO via a high resistance.

2.3.2 Test interface pins

The VR5500 can be tested and debugged with the device mounted on the board. The test interface pins are used to connect an external debugging tool. Therefore, handle the test interface pins as follows when the debugging function is not used and in the normal operating mode.

Pin	Handling
JTCK	Pull up
JTDI	Pull up
JTMS	Pull up
JTRST#[Note]	Pull down
JTDO	Leave open
NTrcClk	Leave open
NTrcData(3:0)	Leave open
NTrcEnd	Leave open
RMode#/BKTGIO#	Pull up

Note Only Ver. 2.0 or later

CHAPTER 3   OUTLINE OF INSTRUCTION SET

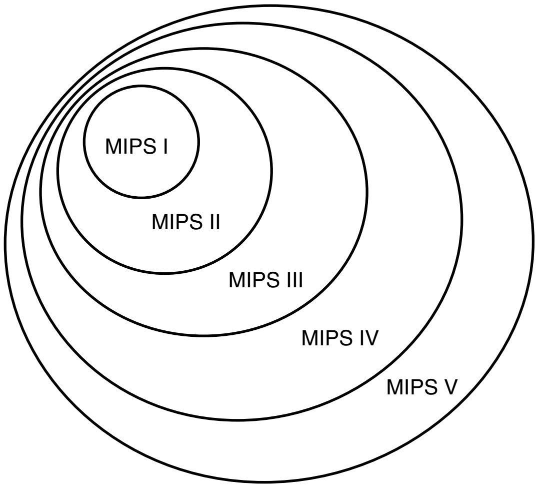
This chapter describes the architecture of the instruction set and outlines the CPU instruction set used for the VR5500.

3.1 Instruction Set Architecture

The VR5500 can execute the MIPS IV instruction set and additional instructions dedicated to the VR5500.

At present, five MIPS instruction set levels, levels I to V, are available. Instruction sets with higher level numbers include instruction sets with lower level numbers (refer to **Figure 3-1**). Therefore, a processor having the MIPS V instruction set can execute the binary program of MIPS I, MIPS II, MIPS III, and MIPS IV without modification.

Figure 3-1. Expansion of MIPS Architecture



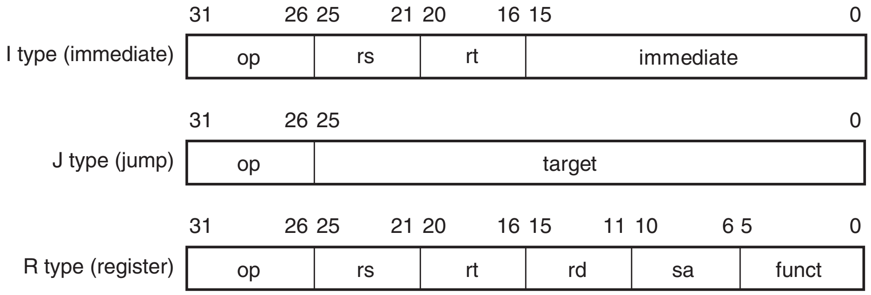
The instructions used in the VR5500 can be classified as follows. For operation details, refer to the corresponding chapter.

- CPU instructions (refer to **3.3 Outline of CPU Instruction Set** and **CHAPTER 17 CPU INSTRUCTION SET**)
- Floating-point (FPU) instructions (Refer to **7.5 Outline of FPU Instruction Set** and **CHAPTER 18 FPU INSTRUCTION SET**)

3.1.1 Instruction format

All instructions are 1-word (32-bit) instructions and are located at the word boundary. Three types of instruction formats are available as shown in Figure 3-2. By simplifying the instruction formats to three, decoding instructions is simplified. Operations and addressing modes that are complicated and not often used are realized by combining two or more instructions with a compiler.

Figure 3-2. Instruction Format



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

3.1.2 Load/store instructions

The load/store instructions transfer data between memory, the CPU, and the general-purpose registers of the coprocessor. These instructions are used to transfer fields of various sizes, treat loaded data as a signed or unsigned integer, access unaligned fields, select the addressing mode, and update the atomic memory (read-modify-write).

A halfword, word, or doubleword address indicates the least significant byte of the bytes generating an object, regardless of the byte order (big endian or little endian). In big endian, this is the most significant byte; it is the least significant byte in little endian.

With some exceptions, the load/store instructions must access an object that is naturally aligned. If an attempt is made to load/store an object at an address that is not even times greater than the size of the object, an address error exception occurs.

New load/store operations have been added at each level of the architecture.

MIPS II

- 64-bit coprocessor transfer
- Atomic update

MIPS III

- 64-bit CPU transfer
- Loading unsigned word to CPU

MIPS V

- Register + register addressing mode of FPU

Remarks 1. The VR5500 does not support an environment where two or more processors operate simultaneously. To maintain compatibility with the other VR Series processors, however, the atomic update instructions of memory defined by MIPS II ISA (such as the load link instruction and conditional store instruction) operate correctly.

The load link bit (LL bit) is set by the LL instruction, cleared by the ERET instruction, and tested by the SC instruction. If the LL bit cannot be set because the cache has become invalid, it can be manipulated only when it is reset from an external source.

2. The SYNC instruction is processed as a NOP instruction. The processor waits until all the instructions issued before the SYNC instruction are committed. Therefore, an LL/SC instruction placed before and after the SYNC instruction can be executed in the program sequence.

Tables 3-1 and 3-2 show the supported load/store instructions and the level of the MIPS architecture at which each instruction is supported first.

Table 3-1. Load/Store Instructions Using Register + Offset Addressing Mode

Data Size	CPU			Coprocessor (Except 0)	
	Signed Load	Unsigned Load	Store	Load	Store
Byte	I	I	I		
Halfword	I	I	I		
Word	I	III	I	I	I
Doubleword	III		III	II	II
Unaligned word	I		I		
Unaligned doubleword	III		III		
Link word (atomic modify)	II		II		
Link doubleword (atomic modify)	III		III		

Table 3-2. Load/Store Instructions Using Register + Register Addressing Mode

Data Size	Floating-Point Coprocessor Only	
	Load	Store
Word	IV	IV
Doubleword	IV	IV

(1) Scheduling load delay slot

The instruction position immediately after a load instruction is called a load delay slot. An instruction that contains a load destination register can be described in the load delay slot, but an interlock is generated for the required number of cycles. Therefore, although any instruction description can be made, it is recommended to schedule the load delay slot from the viewpoints of improving performance and maintaining compatibility with the VR Series. However, because the VR5500 executes instructions by using an out-of-order mechanism, it can resolve a load delay even if scheduling is not made by software.

(2) Definition of access type

The access type is the size of the data the processor loads/stores.

The opcode of a load/store instruction determines the access type. Figure 3-3 shows the access type and the data that is loaded/stored. The address used for a load/store instruction is the least significant byte address (address indicating the least significant byte in little endian), regardless of the access type and byte order (endianness).

The byte order in the doubleword of the accessed data is determined by the access type and the lower 3 bits of the address, as shown in Figure 3-3. Combinations of the access type and the lower bits of the address other than those shown in Figure 3-3 are prohibited (except for the LUXC1 and SUXC1 instructions). If such combinations are used, an address error exception occurs.

Figure 3-3. Byte Specification Related to Load and Store Instructions

Access Type (Value)	Lower Address Bit			Accessed Byte (Big Endian)								Accessed Byte (Little Endian)							
	2	1	0	63							0	63							0
Doubleword (7)	0	0	0	0	1	2	3	4	5	6	7	7	6	5	4	3	2	1	0
7-byte (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	
6-byte (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		
5-byte (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				
3-byte (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							

3.1.3 Operation instructions

Arithmetic operations of 2's complement are executed using integers expressed as 2's complement. Signed addition, subtraction, multiplication, and division instructions are available. "Unsigned" addition and subtraction instructions are also available but these are actually modulo operation instructions that do not detect overflow. Unsigned multiplication and division instructions are also available, as are all shift and logical operation instructions.

MIPS I executes a 32-bit arithmetic operation using 32-bit integers. MIPS III can also execute arithmetic shift instructions using 64-bit operands as 64-bit integers have been added. The logical operations are not affected by the width of the registers.

The operation instructions perform the following operations, using the value of registers.

- Arithmetic operation
- Logical operation
- Shift
- Rotate
- Multiplication
- Division
- Sum-of-products operation
- Counting 0/1 in data

These operations are processed by the following six types of operation instructions.

- ALU immediate instructions
- 3-operand type instructions
- Shift/rotate instructions
- Multiplication/division instructions
- Sum-of-products instructions
- Register scan instructions

Internally, the VR5500 performs processing in 64-bit units. A 32-bit operand can also be used but must be sign-extended. The basic arithmetic and logical instructions such as ADD, ADDU, SUB, SUBU, ADDI, SLL, SRA, and SLLV can support 32-bit operands. If the operand is not correctly sign-extended, however, the operation is undefined. 32-bit data is sign-extended and stored in a 64-bit register.

3.1.4 Jump/branch instructions

All jump and branch instructions always have a delay slot of one instruction. The instruction immediately after a jump/branch instruction (instruction in the delay slot) is executed while the instruction at the destination is being fetched from the cache. The jump/branch instruction cannot be placed in a delay slot. Even if it is placed, however, an error is not detected, and the execution result of the program is undefined.

If execution of the instruction in a delay slot is aborted by the occurrence of an exception or interrupt, the virtual address of the jump/branch instruction immediately before is stored in the EPC register. When the program returns from processing the exception or interrupt, both the jump/branch instruction and the instruction in its delay slot are re-executed. Therefore, do not use register 31 (link address register) as the source register of the Jump and Link, and Branch and Link instructions.

Because an instruction must be placed at the word boundary, use a register in which an address whose lower bits are 0 is stored as the operand of the JR and JALR instructions. If the lower 2 bits of the address are not 0, an address error exception occurs when the destination of the instruction is fetched.

(1) Outline of jump instructions

To call a subroutine described in a high-level language, the J or JAL instruction is usually used. The J and JAL instructions are J-type instructions. This format shifts a 26-bit target address 2 bits to the left and combines the result with the higher 4 bits of the current program counter to generate an absolute address.

Usually, the JR or JALR instruction is used to exit, dispatch, or jump between pages. Both these instructions are R-type and reference the 64-bit byte address of a general-purpose register.

(2) Outline of branch instructions

The branch address of all the branch instructions is 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.

If the branch condition of the Branch Likely instruction is not satisfied, the instruction in the delay slot is invalid. The instruction in the delay slot is executed unconditionally for the other branch instructions.

3.1.5 Special instructions

The special instructions generate an exception by software unconditionally or conditionally. Actually, system call, breakpoint, and trap exceptions occur in the processor. System calls and breakpoints are unconditionally executed, whereas a condition can be specified for a trap.

The SYNC instruction is used to terminate all pending instructions. The VR5500 executes the SYNC instruction as NOP.

3.1.6 Coprocessor instructions

The coprocessor is an alternate execution unit that has a register file separated from the CPU. The MIPS architecture allows allocation of up to four coprocessors, 0 to 3. Each architecture level defines these coprocessors as shown in Table 3-3. Coprocessor 0 is always used for system control, and coprocessor 1 is used as a floating-point unit. The other coprocessors are valid in terms of architecture but have no usage allocated. Some coprocessors are undefined and their opcode is reserved or used for other purposes.

Table 3-3. Definition and Usage of Coprocessors by MIPS Architecture

Coprocessor	MIPS Architecture Level			
	I	II	III	IV
0	System control	System control	System control	System control
1	Floating-point operation	Floating-point operation	Floating-point operation	Floating-point operation
2	Unused	Unused	Unused	Unused
3	Unused	Unused	Undefined	Floating-point operation (COP1X)

A coprocessor has two register sets: coprocessor general-purpose registers and coprocessor control registers. Each register set has up to 32 registers. Depending on the operation instruction of the coprocessor, both the register sets may be changed.

All system control of a MIPS processor is provided as coprocessor 0 (CP0: system control processor). This coprocessor has processor control, memory management, and exception processing functions. The CP0 instructions are peculiar to each CPU.

If the system has an internal floating-point unit, it is used as coprocessor 1 (CP1). With MIPS IV, the FPU uses the opcode space for coprocessor unit 3 as COP1X. For the FPU instructions, refer to **7.5 Outline of FPU Instruction Set** and **CHAPTER 18 FPU INSTRUCTION SET**.

The coprocessor instructions can be classified into the following two major groups.

- Load/store instructions reserved for the main opcode space
- Coprocessor-specific operations that are defined by the coprocessor

(1) Load/store for coprocessor

No load/store instruction is defined for CP0. To read/write a CP0 register, therefore, only an instruction that transfers data to or from the coprocessor can be used.

(2) Coprocessor operation

Up to four coprocessors can be used. To which coprocessor an instruction belongs is indicated by z (z = 0 to 3) suffixed to the mnemonic. In the main opcode, the coprocessor has a coprocessor-specific coded instruction.

3.2 Addition and Modification of VR5500 Instructions

The VR5500 has additional instructions that can be used for multimedia applications, such as sum-of-products instructions and register scan instructions. These additional instructions are not included in the MIPS IV instruction set.

In addition, MIPS ISA makes instructions already defined available again and expands and changes functions.

3.2.1 Integer rotate instructions

Integer rotate instructions have also been added to the VR5500 in the same manner as the VR5432.

These instructions shift the value of a general-purpose register to the right by the number of bits specified by 5 bits of the instruction or by the number of bits specified by a register. The least significant bit that has been shifted is joined to the most significant bit, and the result is stored in the destination register.

Table 3-4. Rotate Instructions

Instruction	Definition
DROR	Doubleword Rotate Right
DROR32	Doubleword Rotate Right + 32
DRORV	Doubleword Rotate Right Variable
ROR	Rotate Right
RORV	Rotate Right Variable

3.2.2 Sum-of-products instructions

Sum-of-products instructions have also been added to the VR5500 in the same manner as the VR5432.

These instructions add a value to the result of multiplication, using the HI register and LO register as an accumulator, and store the result in the destination register. The accumulator is 64 bits long with the lower 32 bits of the HI register as its higher bits and the lower 32 bits of the LO register as its lower bits. No overflow or underflow occurs as a result of executing these instructions. Therefore, no exception occurs.

In addition to the MACC instruction added to the VR5432, the VR5500 also has a sum-of-products instruction that does not store the result in a general-purpose register, and a multiplication instruction that does not store the result in the HI or LO register.

Table 3-5. MACC Instructions

Instruction	Definition
MACC	Multiply, Accumulate, and Move LO
MACCHI	Multiply, Accumulate, and Move HI
MACCHIU	Unsigned Multiply, Accumulate, and Move HI
MACCU	Unsigned Multiply, Accumulate, and Move LO
MSAC	Multiply, Negate, Accumulate, and Move LO
MSACHI	Multiply, Negate, Accumulate, and Move HI
MSACHIU	Unsigned Multiply, Negate, Accumulate, and Move HI
MSACU	Unsigned Multiply, Negate, Accumulate, and Move LO
MUL	Multiply and Move LO
MULHI	Multiply and Move HI
MULHIU	Unsigned Multiply and Move HI
MULS	Multiply, Negate, and Move LO
MULSHI	Multiply, Negate, and Move HI
MULSHIU	Unsigned Multiply, Negate, and Move HI
MULSU	Unsigned Multiply, Negate, and Move LO
MULU	Unsigned Multiply and Move LO

Table 3-6. Sum-of-Products Instructions

Instruction	Definition
MADD	Multiply and Add Word
MADDU	Multiply and Add Word Unsigned
MSUB	Multiply and Subtract Word
MSUBU	Multiply and Subtract Word Unsigned
MUL64	Multiply

3.2.3 Register scan instructions

Register scan instructions have been added to the VR5500.

These instructions scan the contents of a general-purpose register and store the number of 0s or 1s of the register in the destination register.

Table 3-7. Register Scan Instructions

Instruction	Definition
CLO	Count Leading Ones
CLZ	Count Leading Zeros
DCLO	Count Leading Ones in Doubleword
DCLZ	Count Leading Zeros in Doubleword

3.2.4 Floating-point load/store instructions

These instructions have been added to the VR5500.

They load/store data between a floating-point register and memory regardless of whether data is aligned or not.

Table 3-8. Floating-Point Load/Store Instructions

Instruction	Definition
LUXC1	Load Doubleword Indexed Unaligned
SUXC1	Store Doubleword Indexed Unaligned

3.2.5 Other additional instructions

Coprocessor 0 branch instructions are not supported by the VR5000 Series but they are available in the VR5500 again.

In addition, an instruction that is used to manipulate the contents of the performance counter in coprocessor 0, and a NOP instruction that synchronizes the superscalar pipeline are also provided.

The standby mode instructions supported by the VR5000 are also provided in the VR5500.

Table 3-9. Coprocessor 0 Instructions

Instruction	Definition
BC0T	Branch on Coprocessor 0 True
BC0F	Branch on Coprocessor 0 False
BC0TL	Branch on Coprocessor 0 True Likely
BC0FL	Branch on Coprocessor 0 False Likely
MTPC	Move to Performance Counter
MFPC	Move from Performance Counter
MTPS	Move to Performance Event Specifier
MFPS	Move from Performance Event Specifier

Table 3-10. Special Instructions

Instruction	Definition
SSNOP	Superscalar NOP
WAIT	Wait

3.2.6 Instructions for which functions and operations were changed

Functions and operations have been changed in the following instructions.

Table 3-11. Instruction Function Changes in VR5500

Instruction	Major Changed Points
CACHE	In Fill and Fetch_and_Lock operation, the way to be replaced is selected based on the LRU bit of the cache tag.
TLBP	(Compatible with MIPS64)
TLBR	(Compatible with VR5000 Series)
SC	The LL bit is not changed[Note]
SCD	
SYNC	The SYNC instruction is executed after all the on-going instructions complete the commit stage.

Note In the VR5432, the LL bit is cleared when the SC/SCD instruction is executed.

3.3 Outline of CPU Instruction Set

3.3.1 Load and store instructions

Load and store are I-type instructions that transfer data between memory and general-purpose registers. The only addressing mode that load and store instructions directly support is the mode to add a signed 16-bit signed immediate offset to the base register.

Table 3-12. Load/Store Instructions

Instruction	Format and Description op base rt offset
Load Byte	LB rt, offset (base) The sign-extended *offset* is added to the contents of register *base* to generate an address. The contents of the bytes specified by the address are sign-extended and loaded to register *rt*.
Load Byte Unsigned	LBU rt, offset (base) The sign-extended *offset* is added to the contents of register *base* to generate an address. The contents of the bytes specified by the address are zero-extended and loaded to register *rt*.
Load Halfword	LH rt, offset (base) The sign-extended *offset* is added to the contents of register *base* to generate an address. The contents of the halfword specified by the address are sign-extended and loaded to register *rt*.
Load Halfword Unsigned	LHU rt, offset (base) The sign-extended *offset* is added to the contents of register *base* to generate an address. The contents of the halfword specified by the address are zero-extended and loaded to register *rt*.
Load Word	LW rt, offset (base) The sign-extended *offset* is added to the contents of register *base* to generate an address. The contents of the word specified by the address is loaded to register *rt*. In the 64-bit mode, it is sign-extended.
Load Word Left	LWL rt, offset (base) The sign-extended *offset* is added to the contents of register *base* to generate an address. The word whose address is specified is shifted to the left so that the address-specified byte is at the left-most position of the word. The result is merged with the contents of register *rt* and loaded to register *rt*. In the 64-bit mode, it is sign-extended.
Load Word Right	LWR rt, offset (base) The sign-extended *offset* is added to the contents of register *base* to generate an address. The word whose address is specified is shifted to the right so that the address-specified byte is at the right-most position of the word. The result is merged with the contents of register *rt* and loaded to register *rt*. In the 64-bit mode, it is sign extended.
Store Byte	SB rt, offset (base) The sign-extended *offset* is added to the contents of register *base* to generate an address. The least significant byte of register *rt* is stored in the memory location specified by the address.
Store Halfword	SH rt, offset (base) The sign-extended *offset* is added to the contents of register *base* to generate an address. The least significant halfword of register *rt* is stored in the memory location specified by the address.
Store Word	SW rt, offset (base) The sign-extended *offset* is added to the contents of register *base* to generate an address. The lower word of register *rt* is stored in the memory location specified by the address.
Store Word Left	SWL rt, offset (base) The sign-extended *offset* is added to the contents of register *base* to generate an address. The contents of register *rt* is shifted to the right so that the left-most byte of the word is in the position of the address-specified byte. The result is stored in the lower word in memory.
Store Word Right	SWR rt, offset (base) The sign-extended *offset* is added to the contents of register *base* to generate an address. The contents of register *rt* is shifted to the left so that the right-most byte of the word is in the position of the address-specified byte. The result is stored in the upper word in memory.

Table 3-13. Load/Store Instructions (Extended ISA)

Instruction	Format and Description op base rt offset
Load Doubleword	LD rt, offset (base) The sign-extended *offset* is added to the contents of register *base* to generate an address. The contents of the doubleword specified by the address are loaded to register *rt*.
Load Doubleword Left	LDL rt, offset (base) The sign-extended *offset* is added to the contents of register *base* to generate an address. The doubleword whose address is specified is shifted to the left so that the address-specified byte is at the left-most position of the doubleword. The result is merged with the contents of register *rt* and loaded to register *rt*.
Load Doubleword Right	LDR rt, offset (base) The sign-extended *offset* is added to the contents of register *base* to generate an address. The doubleword whose address is specified is shifted to the right so that the address-specified byte is at the right-most position of the doubleword. The result is merged with the contents of register *rt* and loaded to register *rt*.
Load Linked	LL rt, offset (base) The sign-extended *offset* is added to the contents of register *base* to generate an address. The contents of the word specified by the address are loaded to register *rt* and the LL bit is set to 1.
Load Linked Doubleword	LLD rt, offset (base) The sign-extended *offset* is added to the contents of register *base* to generate an address. The contents of the doubleword specified by the address are loaded to register *rt* and the LL bit is set to 1.
Load Word Unsigned	LWU rt, offset (base) The sign-extended *offset* is added to the contents of register *base* to generate an address. The contents of the word specified by the address are zero-extended and loaded to register *rt*.
Store Conditional	SC rt, offset (base) The sign-extended *offset* is added to the contents of register *base* to generate an address. If the LL bit is set to 1, the contents of the lower word of register *rt* are stored in the memory specified by the address, and register *rt* is set to 1. If the LL bit is set to 0, the store operation is not performed and register *rt* is cleared to 0.
Store Conditional Doubleword	SCD rt, offset (base) The sign-extended *offset* is added to the contents of register *base* to generate an address. If the LL bit is set to 1, the contents of register *rt* are stored in the memory specified by the address, and register *rt* is set to 1. If the LL bit is set to 0, the store operation is not performed and register *rt* is cleared to 0.
Store Doubleword	SD rt, offset (base) The sign-extended *offset* is added to the contents of register *base* to generate an address. The contents of register *rt* are stored in the memory specified by the address.
Store Doubleword Left	SDL rt, offset (base) The sign-extended *offset* is added to the contents of register *base* to generate an address. The contents of register *rt* is shifted to the right so that the left-most byte of the doubleword is in the position of the address-specified byte. The result is stored in the lower doubleword in memory.
Store Doubleword Right	SDR rt, offset (base) The sign-extended *offset* is added to the contents of register *base* to generate an address. The contents of register *rt* is shifted to the left so that the right-most byte of the doubleword is in the position of the address-specified byte. The result is stored in the higher doubleword in memory.

3.3.2 Computational instructions

Computational instructions perform arithmetic, logical, and shift operations on values in registers. Computational instructions can be either in register (R-type) format, in which both operands are registers, or in immediate (I-type) format, in which one operand is a 16-bit immediate.

Computational instructions are classified as:

(1) ALU immediate instructions
(2) Three-operand type instructions
(3) Shift/rotate instructions
(4) Multiply/divide instructions
(5) Sum-of-products instructions
(6) Register scan instructions

Table 3-14. ALU Immediate Instructions

Instruction	Format and Description | op | rs | rt | immediate |
Add Immediate	ADDI rt, rs, immediate The 16-bit *immediate* is sign-extended and added to the contents of register *rs*. The 32-bit result is stored in register *rt*. In the 64-bit mode, it is sign-extended. An exception occurs on the generation of 2's complement overflow.
Add Immediate Unsigned	ADDIU rt, rs, immediate The 16-bit *immediate* is sign-extended and added to the contents of register *rs*. The 32-bit result is stored in register *rt*. In the 64-bit mode, it is sign extended. No exception occurs on the generation of overflow.
Set on Less Than Immediate	SLTI rt, rs, immediate The 16-bit *immediate* is sign-extended and compared to the contents of register *rt* treating both operands as signed integers. If *rs* is less than the *immediate*, 1 is stored in register *rt*; otherwise 0 is stored in register *rt*.
Set on Less Than Immediate Unsigned	SLTIU rt, rs, immediate The 16-bit *immediate* is sign-extended and compared to the contents of register *rt* treating both operands as unsigned integers. If *rs* is less than the *immediate*, 1 is stored in register *rt*; otherwise 0 is stored in register *rt*.
AND Immediate	ANDI rt, rs, immediate The 16-bit *immediate* is zero-extended and ANDed with the contents of the register *rs*. The result is stored in register *rt*.
OR Immediate	ORI rt, rs, immediate The 16-bit *immediate* is zero-extended and ORed with the contents of the register *rs*. The result is stored in register *rt*.
Exclusive OR Immediate	XORI rt, rs, immediate The 16-bit *immediate* is zero-extended and Ex-ORed with the contents of the register *rs*. The result is stored in register *rt*.
Load Upper Immediate	LUI rt, immediate The 16-bit *immediate* is shifted left by 16 bits to set the lower 16 bits of word to 0. The result is stored in register *rt*. In the 64-bit mode, it is sign extended.

Table 3-15. ALU Immediate Instructions (Extended ISA)

Instruction	Format and Description op rs rt immediate
Doubleword Add Immediate	DADDI rt, rs, immediate The 16-bit *immediate* is sign-extended to 64 bits and added to the contents of register *rs*. The 64-bit result is stored in register *rt*. An exception occurs on the generation of integer overflow.
Doubleword Add Immediate Unsigned	DADDIU rt, rs, immediate The 16-bit *immediate* is sign-extended to 64 bits and added to the contents of register *rs*. The 64-bit result is stored in register *rt*. No exception occurs on the generation of overflow.

Table 3-16. Three-Operand Type Instructions

Instruction	Format and Description op rs rt rd sa funct
Add	ADD rd, rs, rt The contents of registers *rs* and *rt* are added. The 32-bit result is stored in register *rd*. In the 64-bit mode, it is sign-extended. An exception occurs on the generation of integer overflow.
Add Unsigned	ADDU rd, rs, rt The contents of registers *rs* and *rt* are added. The 32-bit result is stored in register *rd*. In the 64-bit mode, it is sign-extended. No exception occurs on the generation of integer overflow.
Subtract	SUB rd, rs, rt The contents of register *rt* are subtracted from the contents of register *rs*. The 32-bit result is stored in register *rd*. In the 64-bit mode, it is sign-extended. An exception occurs on the generation of integer overflow.
Subtract Unsigned	SUBU rd, rs, rt The contents of register *rt* are subtracted from the contents of register *rs*. The 32-bit result is stored in register rd. In the 64-bit mode, it is sign-extended. No exception occurs on the generation of integer overflow.
Set on Less Than	SLT rd, rs, rt The contents of registers *rs* and *rt* are compared, treating both operands as signed integers. If the contents of register *rs* are less than those of register *rt*, 1 is stored in register *rd*; otherwise 0 is stored in register *rd*.
Set on Less Than Unsigned	SLTU rd, rs, rt The contents of registers *rs* and *rt* are compared, treating both operands as unsigned integers. If the contents of register *rs* are less than those of register *rt*, 1 is stored in register *rd*; otherwise 0 is stored in register *rd*.
AND	AND rd, rt, rs The contents of register *rs* are ANDed with those of general-purpose register *rt* bit-wise. The result is stored in register *rd*.
OR	OR rd, rt, rs The contents of register *rs* are ORed with those of general-purpose register *rt* bit-wise. The result is stored in register *rd*.
Exclusive OR	XOR rd, rt, rs The contents of register *rs* are Ex-ORed with those of general-purpose register *rt* bit-wise. The result is stored in register *rd*.
NOR	NOR rd, rt, rs The contents of register *rs* are NORed with those of general-purpose register *rt* bit-wise. The result is stored in register *rd*.

Table 3-17. Three-Operand Type Instructions (Extended ISA)

Instruction	Format and Description op rs rt rd sa funct
Doubleword Add	DADD rd, rt, rs The contents of register *rs* and register *rt* are added. The 64-bit result is stored in register *rd*. An exception occurs on the generation of integer overflow.
Doubleword Add Unsigned	DADDU rd, rt, rs The contents of register *rs* and register *rt* are added. The 64-bit result is stored in register *rd*. No exception occurs on the generation of integer overflow.
Doubleword Subtract	DSUB rd, rt, rs The contents of register *rt* are subtracted from those of register *rs*. The 64-bit result is stored in register rd. An exception occurs on the generation of integer overflow.
Doubleword Subtract Unsigned	DSUBU rd, rt, rs The contents of register *rt* are subtracted from those of register *rs*. The 64-bit result is stored in register *rd*. No exception occurs on the generation of integer overflow.

Instruction	Format and Description SPECIAL rs rt rd sa funct
Move Conditional on Not Zero	MOVN rd, rs, rt The contents of register *rs* are stored in register *rd* if register *rt* is not equal to 0.
Move Conditional on Zero	MOVZ rd, rs, rt The contents of register *rs* are stored in register *rd* if register *rt* is equal to 0.

Table 3-18. Shift Instructions

Instruction	Format and Description op rs rt rd sa funct
Shift Left Logical	SLL rd, rs, sa The contents of register *rt* are shifted left by *sa* bits and zeros are inserted into the lower bits. The 32-bit result is stored in register *rd*. In the 64-bit mode, it is sign-extended.
Shift Right Logical	SRL rd, rs, sa The contents of register *rt* are shifted right by *sa* bits and zeros are inserted into the higher bits. The 32-bit result is stored in register *rd*. In the 64-bit mode, it is sign-extended.
Shift Right Arithmetic	SRA rd, rt, sa The contents of register *rt* are shifted right by *sa* bits and the higher bits are sign-extended. The 32-bit result is stored in register *rd*. In the 64-bit mode, it is sign-extended.
Shift Left Logical Variable	SLLV rd, rt, rs The contents of register *rt* are shifted left and zeros are inserted into the lower bits. The number of bits shifted is specified by the lower 5 bits of register *rs*. The 32-bit result is stored in register *rd*. In the 64-bit mode, it is sign-extended.
Shift Right Logical Variable	SRLV rd, rt, rs The contents of register *rt* are shifted right and zeros are inserted into the higher bits. The number of bits shifted is specified by the lower 5 bits of register *rs*. The 32-bit result is stored in register *rd*. In the 64-bit mode, it is sign-extended.
Shift Right Arithmetic Variable	SRAV rd, rt, rs The contents of register *rt* are shifted right and the higher bits are sign-extended. The number of bits shifted is specified by the lower 5 bits of register *rs*. The 32-bit result is stored in register *rd*. In the 64-bit mode, it is sign-extended.

Table 3-19. Shift Instructions (Extended ISA)

Instruction	Format and Description op rs rt rd sa funct
Doubleword Shift Left Logical	DSLL rd, rt, sa The contents of register *rt* are shifted left by *sa* bits and zeros are inserted into the lower bits. The 64-bit result is stored in register *rd*.
Doubleword Shift Right Logical	DSRL rd, rt, sa The contents of register *rt* are shifted right by *sa* bits and zeros are inserted into the higher bits. The 64-bit result is stored in register *rd*.
Doubleword Shift Right Arithmetic	DSRA rd, rt, sa The contents of register *rt* are shifted right by *sa* bits and the higher bits are sign-extended. The 64-bit result is stored in register *rd*.
Doubleword Shift Left Logical Variable	DSLLV rd, rt, rs The contents of register *rt* are shifted left and zeros are inserted into the lower bits. The number of bits shifted is specified by the lower 6 bits of register *rs*. The 64-bit result is stored in register *rd*.
Doubleword Shift Right Logical Variable	DSRLV rd, rt, rs The contents of register *rt* are shifted right and zeros are inserted into the higher bits. The number of bits shifted is specified by the lower 6 bits of register *rs*. The 64-bit result is stored in register *rd*.
Doubleword Shift Right Arithmetic Variable	DSRAV rd, rt, rs The contents of register *rt* are shifted right and the higher bits are sign-extended. The number of bits shifted is specified by the lower 6 bits of register *rs*. The 64-bit result is stored in register *rd*.
Doubleword Shift Left Logical + 32	DSLL32 rd, rt, sa The contents of register *rt* are shifted left by 32 + *sa* bits and zeros are inserted into the lower bits. The 64-bit result is stored in register *rd*.
Doubleword Shift Right Logical + 32	DSRL32 rd, rt, sa The contents of register *rt* are shifted right by 32 + *sa* bits and zeros are inserted into the higher bits. The 64-bit result is stored in register *rd*.
Doubleword Shift Right Arithmetic + 32	DSRA32 rd, rt, sa The contents of register *rt* are shifted right by 32 + *sa* bits and the higher bits are sign-extended. The 64-bit result is stored in register *rd*.

Table 3-20. Rotate Instructions (For VR5500)

Instruction	Format and Description SPECIAL rs rt rd sa funct
Rotate Right	ROR rd, rt, sa The contents of register *rt* are shifted right by *sa* bits and the lower bits shifted out are inserted into the higher bits. The 32-bit result is stored in register *rd*. In the 64-bit mode, it is sign-extended.
Rotate Right Variable	RORV rd, rt, rs The contents of register *rt* are shifted right and the lower bits shifted out are inserted into the higher bits. The number of bits shifted is specified by the lower 5 bits of register *rs*. The 32-bit result is stored in register *rd*. In the 64-bit mode, it is sign-extended.
Doubleword Rotate Right	DROR rd, rt, sa The contents of register *rt* are shifted right by *sa* bits and the lower bits shifted out are inserted into the higher bits. The 64-bit result is stored in register *rd*.
Doubleword Rotate Right + 32	DROR32 rd, rt, sa The contents of register *rt* are shifted right by 32 + *sa* bits and the lower bits shifted out are inserted into the higher bits. The 64-bit result is stored in register *rd*.
Doubleword Rotate Right Variable	DRORV rd, rt, rs The contents of register *rt* are shifted right and the lower bits shifted out are inserted into the higher bits. The number of bits shifted is specified by the lower 5 bits of register *rs*. The 64-bit result is stored in register *rd*.

Table 3-21. Multiply/Divide Instructions

Instruction	Format and Description op rs rt rd sa funct
Multiply	MULT rs, rt The contents of registers *rs* and *rt* are multiplied, treating both operands as 32-bit signed integers. The 64-bit result is stored in special registers HI and LO. In the 64-bit mode, it is sign-extended.
Multiply Unsigned	MULTU rs, rt The contents of registers *rs* and *rt* are multiplied, treating both operands as 32-bit unsigned integers. The 64-bit result is stored in special registers HI and LO. In the 64-bit mode, it is sign-extended.
Divide	DIV rs, rt The contents of register *rs* are divided by those of register *rt*, treating both operands as 32-bit signed integers. The 32-bit quotient is stored in special register LO, and the 32-bit remainder is stored in special register HI. In the 64-bit mode, it is sign-extended.
Divide Unsigned	DIVU rs, rt The contents of register *rs* are divided by those of register *rt*, treating both operands as 32-bit unsigned integers. The 32-bit quotient is stored in special register LO, and the 32-bit remainder is stored in special register HI. In the 64-bit mode, it is sign-extended.
Move from HI	MFHI rd The contents of special register HI are loaded to register *rd*.
Move from LO	MFLO rd The contents of special register LO are loaded to register *rd*.
Move to HI	MTHI rs The contents of register *rs* are loaded to special register HI.
Move to LO	MTLO rs The contents of register *rs* are loaded to special register LO.

Table 3-22. Multiply/Divide Instructions (Extended ISA)

Instruction	Format and Description op rs rt rd sa funct
Doubleword Multiply	DMULT rs, rt The contents of registers *rs* and *rt* are multiplied, treating both operands as signed integers. The 128-bit result is stored in special registers HI and LO.
Doubleword Multiply Unsigned	DMULTU rs, rt The contents of registers *rs* and *rt* are multiplied, treating both operands as unsigned integers. The 128-bit result is stored in special registers HI and LO.
Doubleword Divide	DDIV rs, rt The contents of register *rs* are divided by those of register *rt*, treating both operands as signed integers. The 64-bit quotient is stored in special register LO, and the 64-bit remainder is stored in special register HI.
Doubleword Divide Unsigned	DDIVU rs, rt The contents of register *rs* are divided by those of register *rt*, treating both operands as unsigned integers. The 64-bit quotient is stored in special register LO, and the 64-bit remainder is stored in special register HI.

Table 3-23. MACC Instructions (For VR5500) (1/2)

Instruction	Format and Description SPECIAL | rs | rt | rd | funct
Multiply, Accumulate, and Move LO	MACC rd, rs, rt The contents of registers *rs* and *rt* are multiplied, treating both operands as 32-bit signed integers, and result is added to a value that combines the lower 32 bits of special registers HI and LO. The lower 32 bits of the result are stored in register *rd*.
Unsigned Multiply, Accumulate, and Move LO	MACCU rd, rs, rt The contents of registers *rs* and *rt* are multiplied, treating both operands as 32-bit unsigned integers, and result is added to a value that combines the lower 32 bits of special registers HI and LO. The lower 32 bits of the result are stored in register *rd*.
Multiply, Accumulate, and Move HI	MACCHI rd, rs, rt The contents of registers *rs* and *rt* are multiplied, treating both operands as 32-bit signed integers, and result is added to a value that combines the lower 32 bits of special registers HI and LO. The higher 32 bits of the result are stored in register *rd*.
Unsigned Multiply, Accumulate, and Move HI	MACCHIU rd, rs, rt The contents of registers *rs* and *rt* are multiplied, treating both operands as 32-bit unsigned integers, and result is added to a value that combines the lower 32 bits of special registers HI and LO. The higher 32 bits of the result are stored in register *rd*.
Multiply, Negate, Accumulate, and Move LO	MSAC rd, rs, rt The contents of registers *rs* and *rt* are multiplied, treating both operands as 32-bit signed integers, and result is subtracted from a value that combines the lower 32 bits of special registers HI and LO. The lower 32 bits of the result are stored in register *rd*.
Unsigned Multiply, Negate, Accumulate, and Move LO	MSACU rd, rs, rt The contents of registers *rs* and *rt* are multiplied, treating both operands as 32-bit unsigned integers, and result is subtracted from a value that combines the lower 32 bits of special registers HI and LO. The lower 32 bits of the result are stored in register *rd*.
Multiply, Negate, Accumulate, and Move HI	MSACHI rd, rs, rt The contents of registers *rs* and *rt* are multiplied, treating both operands as 32-bit signed integers, and result is subtracted from a value that combines the lower 32 bits of special registers HI and LO. The higher 32 bits of the result are stored in register *rd*.
Unsigned Multiply, Negate, Accumulate, and Move HI	MSACHIU rd, rs, rt The contents of registers *rs* and *rt* are multiplied, treating both operands as 32-bit unsigned integers, and result is subtracted from a value that combines the lower 32 bits of special registers HI and LO. The higher 32 bits of the result are stored in register *rd*.
Multiply and Move LO	MUL rd, rs, rt The contents of registers *rs* and *rt* are multiplied, treating both operands as 32-bit signed integers. The higher 32 bits of the result is stored in the lower bits of special register HI, and lower 32 bits of the result are stored in lower bits of special register LO and register *rd*.
Unsigned Multiply and Move LO	MULU rd, rs, rt The contents of registers *rs* and *rt* are multiplied, treating both operands as 32-bit unsigned integers. The higher 32 bits of the result is stored in the lower bits of special register HI, and lower 32 bits of the result are stored in lower bits of special register LO and register *rd*.

Table 3-23. MACC Instructions (For VR5500) (2/2)

Instruction	Format and Description SPECIAL rs rt rd funct
Multiply and Move HI	MULHI rd, rs, rt The contents of registers *rs* and *rt* are multiplied, treating both operands as 32-bit signed integers. The higher 32 bits of the result are stored in the lower bits of special register HI and register *rd*, and the lower 32 bits of the result are stored in the lower bits of special register LO.
Unsigned Multiply and Move HI	MULHIU rd, rs, rt The contents of registers *rs* and *rt* are multiplied, treating both operands as 32-bit unsigned integers. The higher 32 bits of the result are stored in the lower bits of special register HI and register *rd*, and the lower 32 bits of the result are stored in the lower bits of special register LO.
Multiply, Negate, and Move LO	MULS rd, rs, rt The contents of registers *rs* and *rt* are multiplied, treating both operands as 32-bit signed integers, and the result is inverted. The higher 32 bits of the result are stored in the lower bits of special register HI, and the lower 32 bits of the result are stored in the lower bits of special register LO and register *rd*.
Unsigned Multiply, Negate, and Move LO	MULSU rd, rs, rt The contents of registers *rs* and *rt* are multiplied, treating both operands as 32-bit unsigned integers, the result is inverted. The higher 32 bits of the result are stored in the lower bits of special register HI, and the lower 32 bits of the result are stored in the lower bits of special register LO and register *rd*.
Multiply, Negate, and Move HI	MULSHI rd, rs, rt The contents of registers *rs* and *rt* are multiplied, treating both operands as 32-bit signed integers, the result is inverted. The higher 32 bits of the result are stored in the lower bits of special register HI and register *rd*, and the lower 32 bits of the result are stored in the lower bits of special register LO.
Unsigned Multiply, Negate, and Move HI	MULSHIU rd, rs, rt The contents of registers *rs* and *rt* are multiplied, treating both operands as 32-bit unsigned integers, the result is inverted. The higher 32 bits of the result are stored in the lower bits of special register HI and register *rd*, and the lower 32 bits of the result are stored in the lower bits of special register LO.

Table 3-24. Sum-of-Products Instructions (For VR5500)

Instruction	Format and Description SPECIAL2 | rs | rt | rd | 0 | funct
Multiply and Add Word	MADD rs, rt The contents of registers *rs* and *rt* are multiplied, treating both operands as 32-bit signed integers, and the result is added to a value that combines the lower 32 bits of special registers HI and LO. The 64-bit result is stored in special registers HI and LO.
Multiply and Add Word Unsigned	MADDU rs, rt The contents of registers *rs* and *rt* are multiplied, treating both operands as 32-bit unsigned integers, and the result is added to a value that combines the lower 32 bits of special registers HI and LO. The 64-bit result is stored in special registers HI and LO.
Multiply and Subtract Word	MSUB rs, rt The contents of registers *rs* and *rt* are multiplied, treating both operands as 32-bit signed integers, and the result is subtracted from a value that combines the lower 32 bits of special registers HI and LO. The 64-bit result is stored in special registers HI and LO.
Multiply and Subtract Word Unsigned	MSUBU rs, rt The contents of registers *rs* and *rt* are multiplied, treating both operands as 32-bit unsigned integers, and the result is subtracted from a value that combines the lower 32 bits of special registers HI and LO. The 64-bit result is stored in special registers HI and LO.
Multiply	MUL64 rd, rs, rt The contents of registers *rs* and *rt* are multiplied, treating both operands as 32-bit signed integers. The lower 32 bits of the result are stored in register *rd*.

Since the VR5500 stalls the entire pipeline when executing an integer multiply/divide instruction, the number of execution cycle increases compared with normal instruction execution. The number of processor cycles (PCycles) required for an integer multiply/divide instruction is shown below.

Table 3-25. Number of Cycles for Multiply and Divide Instructions

Instruction	Number of PCycles	
	When Executed Singly	When Executed Repeatedly
DIV, DIVU	40	40
DDIV, DDIVU	72	72
MACC, MACCHI, MACCHIU, MACCU, MSAC, MSACHI, MSACHIU, MSACU	3	3
MUL, MULHI, MULHIU, MULU, MULS, MULSHI, MULSHIU, MULSU	3	3
MADD, MADDU, MSUB, MSUBU	2	2
MUL64	2	2
MULT, MULTU	3	3
DMULT, DMULTU	3	3

Table 3-26. Register Scan Instructions (For VR5500)

Instruction	Format and Description SPECIAL2 rs rt rd 0 funct
Count Leading Ones	CLO rd, rs The 32-bit contents of register *rs* are scanned from the highest to lowest bit, and the number of 1s is stored in register *rd*.
Count Leading Zeros	CLZ rd, rs The 32-bit contents of register *rs* are scanned from the highest to lowest bit, and the number of 0s is stored in register *rd*.
Count Leading Ones in Doubleword	DCLO rd, rs The 64-bit contents of register *rs* are scanned from the highest to lowest bit, and the number of 1s is stored in register *rd*.
Count Leading Zeros in Doubleword	DCLZ rd, rs The 64-bit contents of register *rs* are scanned from the highest to lowest bit, and the number of 0s is stored in register *rd*.

3.3.3 Jump and branch instructions

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

For instructions involving a link (such as JAL and BLTZAL), the return address is saved in register r31.

Table 3-27. Jump Instruction

Instruction	Format and Description op target
Jump	J target The contents of the 26-bit target address is shifted left by two bits and combined with the higher 4 bits of the PC. The program jumps to this calculated address with a delay of one instruction.
Jump and Link	JAL target The contents of the 26-bit target address is shifted left by two bits and combined with the higher 4 bits of the PC. The program jumps to this calculated address with a delay of one instruction. The address of the instruction following the delay slot is stored in r31 (link register).

Instruction	Format and Description op rs rt rd sa funct
Jump Register	JR rs The program jumps to the address specified in register *rs* with a delay of one instruction.
Jump and Link Register	JALR rs, rd The program jumps to the address specified in register *rs* with a delay of one instruction. The address of the instruction following the delay slot is stored in *rd.*

Table 3-28. Branch Instructions

Instruction	Format and Description op rs rt offset
Branch on Equal	BEQ rs, rt, offset If the contents of register *rs* are equal to those of register *rt*, the program branches to the target address.
Branch on Not Equal	BNE rs, rt, offset If the contents of register *rs* are not equal to those of register *rt*, the program branches to the target address.
Branch on Less Than or Equal to Zero	BLEZ rs, offset If the contents of register *rs* are less than or equal to zero, the program branches to the target address.
Branch on Greater Than Zero	BGTZ rs, offset If the contents of register *rs* are greater than zero, the program branches to the target address.

Instruction	Format and Description REGIMM rs sub offset
Branch on Less Than Zero	BLTZ rs, offset If the contents of register *rs* are less than zero, the program branches to the target address.
Branch on Greater Than or Equal to Zero	BGEZ rs, offset If the contents of register *rs* are greater than or equal to zero, the program branches to the target address.
Branch on Less Than Zero and Link	BLTZAL rs, offset The address of the instruction that follows delay slot is stored in register r31 (link register). If the contents of register *rs* are less than zero, the program branches to the target address.
Branch on Greater Than or Equal to Zero and Link	BGEZAL rs, offset The address of the instruction that follows delay slot is stored in register r31 (link register). If the contents of register *rs* are greater than or equal to zero, the program branches to the target address.

Remark sub: Sub-operation code

Instruction	Format and Description COP0 BC br offset
Branch on Coprocessor 0 True	BC0T offset The 16-bit *offset* (shifted left by two bits and sign-extended) is added to the address of the instruction in the delay slot to calculate the branch target address. If the conditional signal of the coprocessor 0 is true, the program branches to the target address with one-instruction delay.
Branch on Coprocessor 0 False	BC0F offset The 16-bit *offset* (shifted left by two bits and sign-extended) is added to the address of the instruction in the delay slot to calculate the branch target address. If the conditional signal of the coprocessor 0 is false, the program branches to the target address with one-instruction delay.

Remark BC: BC sub-operation code
br: Branch condition identifier

Table 3-29. Branch Instructions (Extended ISA)

Instruction	Format and Description op rs rt offset
Branch on Equal Likely	BEQL rs, rt, offset If the contents of register *rs* are equal to those of register *rt*, the program branches to the target address. If the branch condition is not met, the instruction in the delay slot is discarded.
Branch on Not Equal Likely	BNEL rs, rt, offset If the contents of register *rs* are not equal to those of register *rt*, the program branches to the target address. If the branch condition is not met, the instruction in the delay slot is discarded.
Branch on Less Than or Equal to Zero Likely	BLEZL rs, offset If the contents of register *rs* are less than or equal to zero, the program branches to the target address. If the branch condition is not met, the instruction in the delay slot is discarded.
Branch on Greater Than Zero Likely	BGTZL rs, offset If the contents of register *rs* are greater than zero, the program branches to the target address. If the branch condition is not met, the instruction in the delay slot is discarded.

Instruction	Format and Description REGIMM rs sub offset
Branch on Less Than Zero Likely	BLTZL rs, offset If the contents of register *rs* are less than zero, the program branches to the target address. If the branch condition is not met, the instruction in the delay slot is discarded.
Branch on Greater Than or Equal to Zero Likely	BGEZL rs, offset If the contents of register *rs* are greater than or equal to zero, the program branches to the target address. If the branch condition is not met, the instruction in the delay slot is discarded.
Branch on Less Than Zero and Link Likely	BLTZALL rs, offset The address of the instruction that follows delay slot is stored in register r31 (link register). If the contents of register *rs* are less than zero, the program branches to the target address. If the branch condition is not met, the instruction in the delay slot is discarded.
Branch on Greater Than or Equal to Zero and Link Likely	BGEZALL rs, offset The address of the instruction that follows delay slot is stored in register r31 (link register). If the contents of register *rs* are greater than or equal to zero, the program branches to the target address. If the branch condition is not met, the instruction in the delay slot is discarded.

Remark sub: Sub-operation code

Instruction	Format and Description COP0 BC br offset
Branch on Coprocessor 0 True Likely	BC0TL offset The 16-bit *offset* (shifted left by two bits and sign-extended) is added to the address of the instruction in the delay slot to calculate the branch target address. If the conditional signal of the coprocessor 0 is true, the program branches to the target address with one-instruction delay. If the branch condition is not met, the instruction in the delay slot is discarded.
Branch on Coprocessor 0 False Likely	BC0FL offset The 16-bit *offset* (shifted left by two bits and sign-extended) is added to the address of the instruction in the delay slot to calculate the branch target address. If the conditional signal of the coprocessor 0 is false, the program branches to the target address with one-instruction delay. If the branch condition is not met, the instruction in the delay slot is discarded.

Remark BC: BC sub-operation code
br: Branch condition identifier

3.3.4 Special instructions

Special instructions mainly generate software exceptions.

Table 3-30. Special Instructions

Instruction	Format and Description SPECIAL rs rt rd sa funct
Synchronize	SYNC Completes the load/store instruction executing in the current pipeline before the next load/store instruction starts execution.
System Call	SYSCALL Generates a system call exception, and then transits control to the exception handling program.
Breakpoint	BREAK Generates a break point exception, and then transits control to the exception handling program.

Table 3-31. 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 The contents of register *rs* are compared with those of register *rt*, treating both operands as signed integers. If the contents of register *rs* are greater than or equal to those of register *rt*, an exception occurs.
Trap if Greater Than or Equal Unsigned	TGEU rs, rt The contents of register *rs* are compared with those of register *rt*, treating both operands as unsigned integers. If the contents of register *rs* are greater than or equal to those of register *rt*, an exception occurs.
Trap if Less Than	TLT rs, rt The contents of register *rs* are compared with those of register *rt*, treating both operands as signed integers. If the contents of register *rs* are less than those of register *rt*, an exception occurs.
Trap if Less Than Unsigned	TLTU rs, rt The contents of register *rs* are compared with those of register *rt*, treating both operands as unsigned integers. If the contents of register *rs* are less than those of register *rt*, an exception occurs.
Trap if Equal	TEQ rs, rt If the contents of registers *rs* and *rt* are equal, an exception occurs.
Trap if Not Equal	TNE rs, rt If the contents of registers *rs* and *rt* are not equal, an exception occurs.

Table 3-31. Special Instructions (Extended ISA) (2/2)

Instruction	Format and Description REGIMM rs sub immediate
Trap if Greater Than or Equal Immediate	TGEI rs, immediate The contents of register *rs* are compared with 16-bit sign-extended *immediate* data, treating both operands as signed integers. If the contents of register *rs* are greater than or equal to 16-bit sign-extended *immediate* data, an exception occurs.
Trap if Greater Than or Equal Immediate Unsigned	TGEIU rs, immediate The contents of register *rs* are compared with 16-bit zero-extended *immediate* data, treating both operands as unsigned integers. If the contents of register *rs* are greater than or equal to 16-bit sign-extended *immediate* data, an exception occurs.
Trap if Less Than Immediate	TLTI rs, immediate The contents of register *rs* are compared with 16-bit sign-extended *immediate* data, treating both operands as signed integers. If the contents of register *rs* are less than 16-bit sign-extended *immediate* data, an exception occurs.
Trap if Less Than Immediate Unsigned	TLTIU rs, immediate The contents of register *rs* are compared with 16-bit zero-extended *immediate* data, treating both operands as unsigned integers. If the contents of register *rs* are less than 16-bit sign-extended *immediate* data, an exception occurs.
Trap if Equal Immediate	TEQI rs, immediate If the contents of register *rs* and *immediate* data are equal, an exception occurs.
Trap if Not Equal Immediate	TNEI rs, immediate If the contents of register *rs* and *immediate* data are not equal, an exception occurs.

Remark sub: Sub-operation code

Instruction	Format and Description op base hint offset
Prefetch	PREF hint, offset (base) Sign-extends a 16-bit *offset* and adds it to register *base* to generate a virtual address. The operation to be performed on that address is indicated by 5-bit *hint*.

Table 3-32. Special Instructions (For VR5500)

Instruction	Format and Description SPECIAL rs rd rt sa funct
Superscalar NOP	SSNOP The processor waits until all preceding instructions have been committed or until writeback to a register by the preceding load instruction has been completed.

3.3.5 Coprocessor instructions

The coprocessor instructions perform the operations of each coprocessor. The coprocessor load and store instructions are I-type instructions. The format of the operation instructions of the coprocessor differs depending on the coprocessor.

Table 3-33. Coprocessor Instructions

Instruction	Format and Description op base rt offset
Load Word to Coprocessor z	LWCz rt, offset (base) Sign-extends an *offset* and adds it to register *base* to generate an address. Loads the contents of a word specified by the address to general-purpose register *rt* of coprocessor z.
Store Word from Coprocessor z	SWCz rt, offset (base) Sign-extends an *offset* and adds it to register *base* to generate an address. Stores the contents of general-purpose register *rt* of coprocessor z in the memory location 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 register rd of coprocessor z.
Move from Coprocessor z	MFCz rt, rd Transfers the contents of 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 coprocessor control register *rd* of coprocessor z.
Move Control from Coprocessor z	CFCz rt, rd Transfers the contents of coprocessor control register *rd* of coprocessor z to CPU register *rt*.

Remark sub: Sub-operation code

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

Remark CO: Sub-operation identifier

Table 3-34. Coprocessor Instructions (Extended ISA)

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

Remark sub: Sub-operation code

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

3.3.6 System control coprocessor (CP0) instructions

System control coprocessor (CP0) instructions perform operations specifically on the CP0 registers to manipulate the memory management and exception handling facilities of the processor.

Table 3-35. System Control Coprocessor (CP0) Instructions (1/2)

Instruction	Format and Description COP0 sub rt rd 0
Move to System Control Coprocessor	MTC0 rt, rd The word data of general register *rt* in the CPU are loaded to general register *rd* in the CP0.
Move from System Control Coprocessor	MFC0 rt, rd The word data of general register *rd* in the CP0 are loaded to general register *rt* in the CPU.
Doubleword Move to System Control Coprocessor 0	DMTC0 rt, rd The doubleword data of general register *rt* in the CPU are loaded to general register *rd* in the CP0.
Doubleword Move from System Control Coprocessor 0	DMFC0 rt, rd The doubleword data of general register *rd* in the CP0 are loaded to general register *rt* in the CPU.

Remark sub: Sub-operation code

Table 3-35. System Control Coprocessor (CP0) Instructions (2/2)

Instruction	Format and Description COP0 CO funct
Read Indexed TLB Entry	TLBR The TLB entry indexed by the Index register is loaded to the EntryHi, EntryLo0, EntryLo1, or PageMask register.
Write Indexed TLB Entry	TLBWI The contents of the EntryHi, EntryLo0, EntryLo1, or PageMask register are loaded to the TLB entry indexed by the Index register.
Write Random TLB Entry	TLBWR The contents of the EntryHi, EntryLo0, EntryLo1, or PageMask register are loaded to the TLB entry indexed by the Random register.
Probe TLB for Matching Entry	TLBP The address of the TLB entry that matches the contents of EntryHi register is loaded to the Index register.
Return from Exception	ERET The program returns from exception, interrupt, or error trap.

Remark CO: Sub-operation identifier

Instruction	Format and Description CACHE base op offset
Cache Operation	CACHE op, offset (base) Sign-extends the 16-bit *offset* and adds to the contents of register *base* to generate a virtual address. This virtual address is translated to physical address with TLB. For this physical address, cache operation that is indicated by 5-bit op is performed.

Table 3-36. System Control Coprocessor (CP0) Instructions (For V_R5500)

Instruction	Format and Description COP0 CO code funct
Wait	WAIT The processor's operating mode is shifted to standby mode.

Remark CO: Sub-operation identifier

Instruction	Format and Description COP0 sub rt rd 0 reg
Move to Performance Counter	MTPC rt, reg The contents of general-purpose register *rt* in the CPU are loaded to performance counter *reg* in the CP0.
Move from Performance Counter	MFPC rt, reg The contents of performance counter *reg* in the CP0 are loaded to general-purpose register *rt* in the CPU.
Move to Performance Event Specifier	MTPS rt, reg The contents of general-purpose register *rt* in the CPU are loaded to performance counter control register *reg* in the CP0.
Move from Performance Event Specifier	MFPS rt, reg The contents of performance counter control register *reg* in the CP0 are loaded to general-purpose register *rt* in the CPU.

Remark sub: Sub-operation code

CHAPTER 4 PIPELINE

This chapter explains the pipeline.

4.1 Overview

The pipeline is one of the instruction execution formats. It divides instruction execution processing into several stages. An instruction has been completely executed when it has gone through all the stages. When one instruction has been processed in one stage, the next instruction enters that stage.

The operating clock of the pipeline is called PClock, and one of its cycles is called PCycle. Each stage of the pipeline is executed in 1 PCycle.

The pipeline of the VR5500 has a two-way superscalar architecture in which two instructions are fetched at a time. The instructions are executed in the pipeline out of order. If the pipeline is completely filled, execution of two instructions can be completed in 1PCycle.

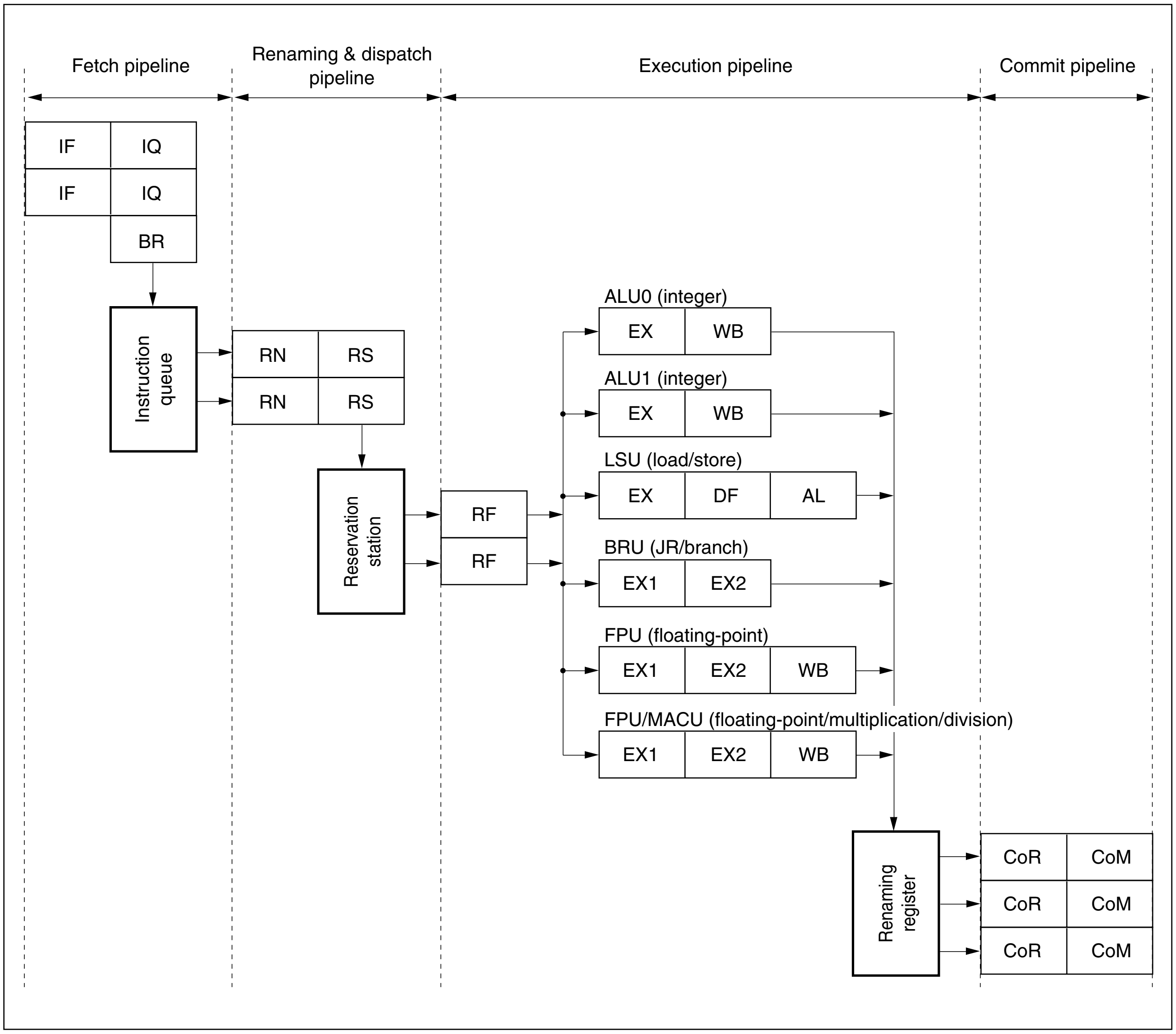
4.1.1 Pipeline stages

The VR5500 has six execution units including integer operation, floating-point operation (including sum-of-products operation), load/store, and branch units. Each of these units operates independently. Therefore, the number of stages of the pipeline differs depending on the instruction. For example, an integer arithmetic operation instruction uses nine stages.

The stages that make up the pipeline include the following.

IF:	Instruction fetch	EX:	Execution
BR:	Branch prediction	DF:	Data fetch
IQ:	Instruction queue	AL:	Data align
RN:	Register renaming	WB:	Writeback
RS:	Reservation station	CoR:	Commit register
RF:	Register fetch	CoM:	Commit memory

Figure 4-1. Pipeline Stages of VR5500 and Instruction Flow



4.1.2 Configuration of pipeline

The pipeline of the VR5500 is divided into four blocks. Each block operates independently.

(1) Fetch pipeline

The fetch pipeline generates a speculative fetch stream in accordance with branch prediction and stores a fetched instruction in a 16-entry instruction queue. It can fetch two instructions per cycle from the 64-bit bus connected to the instruction cache. If the fetched instruction includes a branch or jump instruction, the fetch pipeline immediately calculates the address at the destination by using a branch history table and information on the return address stack, and changes the program flow. As a result, all processing is speculatively issued. Even if the execution pipeline does not execute a branch instruction, therefore, the fetch pipeline continues processing a branch instruction and tracing an instruction stream without stalling, until the instruction queue becomes full.

(2) Renaming & dispatch pipeline

The renaming & dispatch pipeline can receive up to two instructions from the instruction queue per cycle, and assign a renaming register number to the received instructions. At the same time, it overwrites the register number specified as an operand with a renaming register number. The renamed instructions are stored in the reservation station (RS). The VR5500 has an RS dedicated to each execution unit. Four entries each are available for the two ALUs, four entries for LSU, four entries for BRU, two entries for FPU, and two entries for FPU/MACU.

This pipeline continues operating until the instruction queue becomes empty or the RS becomes full.

Each instruction stored in the RS is checked for its dependency upon other instructions and the utilization status of the execution unit necessary for execution. An instruction that has been judged as executable is selected from the RS. Up to two instructions can be selected per cycle. The instruction sequence described in the program is ignored. The two selected instructions are packed into one instruction, like VLIW. The packed instructions are sent to the execution pipeline.

The types of instructions that can be packed are shown below.

Figure 4-2. Combination of Instructions That Can Be Packed

Higher-side instruction	Lower-side instruction	Higher-side instruction	Lower-side instruction	Higher-side instruction	Lower-side instruction
INT	INT	FP	INT	FP	nop
INT	BR	FP	BR	INT	nop
INT	FP	FP	FP	MEM	nop
MEM	INT	MAC	INT	nop	BR
MEM	BR	MAC	BR	nop	MAC
MEM	FP	MAC	FP		

Remark

INT:	Integer operation	BR:	Branch
FP:	Floating-point operation	MEM:	Load/store (memory access)
MAC:	Sum-of-products operation, multiplication/division	nop:	No operation

(3) Execution pipeline

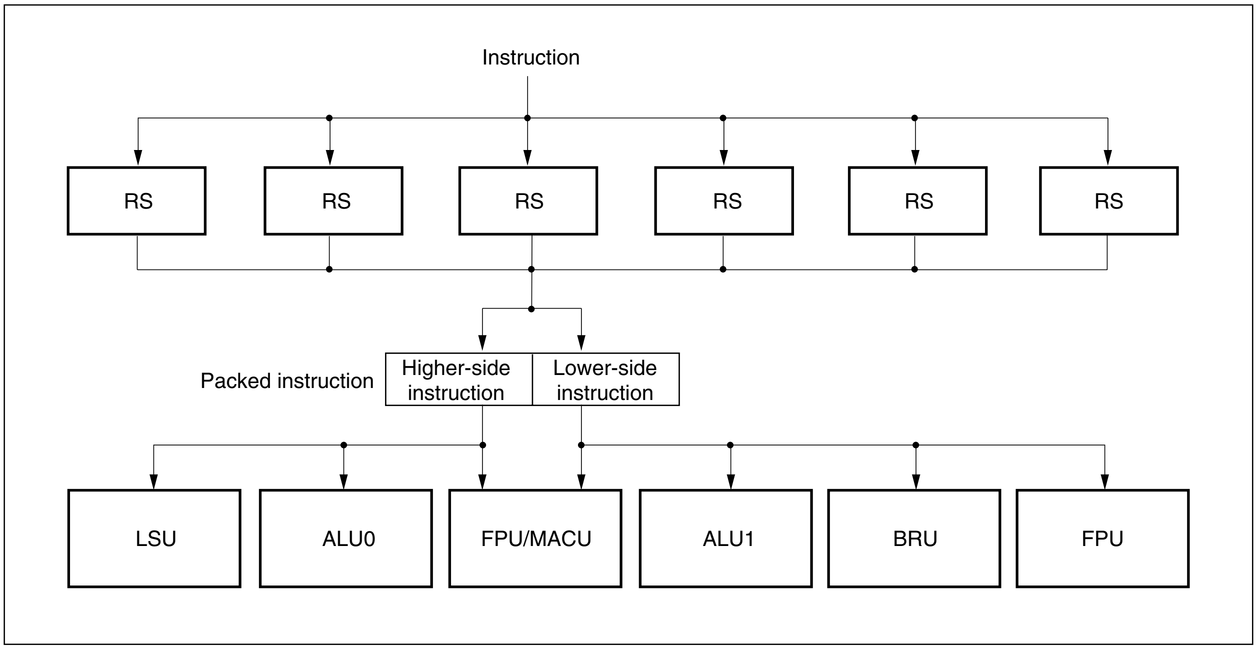
The execution pipeline consists of six execution units. The higher side of the packed instructions is sent to the LSU, ALU0, and FPU/MACU, and is executed by one of these units. The lower side is sent to the FPU/MACU, ALU1, BRU, and FPU, and is executed by one of them.

The FPU/MACU and FPU execute floating-point operations. The FPU/MACU is a FPU with a multiplier/divider added, and can also execute integer multiplication/division.

All the execution results are stored in the renaming register assigned to the instruction along with exception information that has been detected.

Instructions do not stall in the execution pipeline of the VR5500. All dependency relationships and resource conflicts are resolved by the renaming & dispatch pipeline before the execution pipeline. Therefore, the execution pipeline of the VR5500 is not provided with a mechanism for stall detection.

Figure 4-3. Instruction Flow in Execution Pipeline



(4) Commit pipeline

The commit pipeline controls the processor state. The instructions that are executed by the execution pipeline regardless of the program sequence are completed (committed) in the program sequence by this pipeline. The commit pipeline performs the following processing.

- Checking of exception/trap
- Updating store buffer
- Updating processor state

4.2 Branch Delay

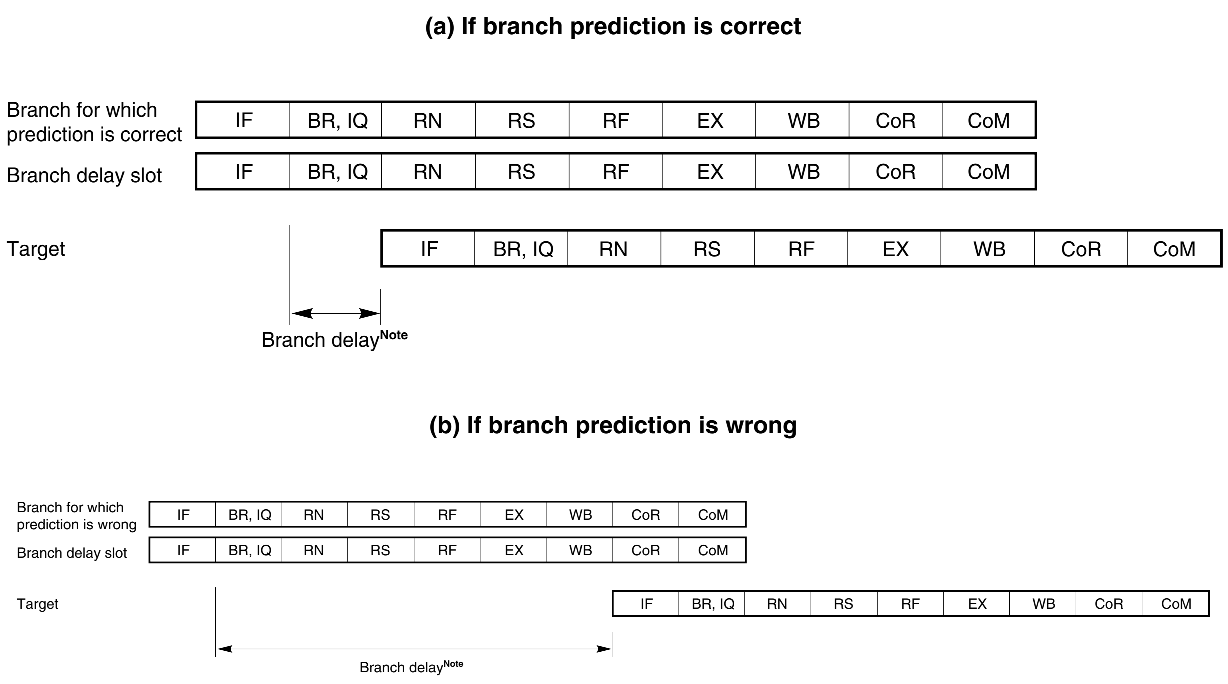
The position of the instruction next to a branch instruction is called the branch delay slot. The instruction in the branch delay slot is executed regardless of whether the condition of the branch instruction (except the Branch Likely instruction) is satisfied or not.

To accelerate branch processing, the VR5500 has a branch prediction mechanism. This mechanism uses a branch history table (BHT) with 4096 entries (2 bits each) to record satisfaction of the condition of branch instructions executed in the past. It also uses a return address stack (RAS) to hold the address to which execution is to return after a function call. The VR5500 predicts the target address of a branch instruction in accordance with the BHT, and speculatively fetches and executes the subsequent instructions.

The pipeline of the VR5500 generates a branch delay of six cycles if branch prediction is wrong. If branch prediction is correct, the branch delay is 1 cycle.

Figure 4-4 shows how branch prediction is performed and the position of the branch delay slot.

Figure 4-4. Branch Delay



(a) If branch prediction is correct

Branch for which prediction is correct
IF | BR, IQ | RN | RS | RF | EX | WB | CoR | CoM
Branch delay slot
IF | BR, IQ | RN | RS | RF | EX | WB | CoR | CoM
Target
IF | BR, IQ | RN | RS | RF | EX | WB | CoR | CoM
Branch delay[Note]

(b) If branch prediction is wrong

Branch for which prediction is wrong
IF | BR, IQ | RN | RS | RF | EX | WB | CoR | CoM
Branch delay slot
IF | BR, IQ | RN | RS | RF | EX | WB | CoR | CoM
Target
IF | BR, IQ | RN | RS | RF | EX | WB | CoR | CoM
Branch delay[Note]

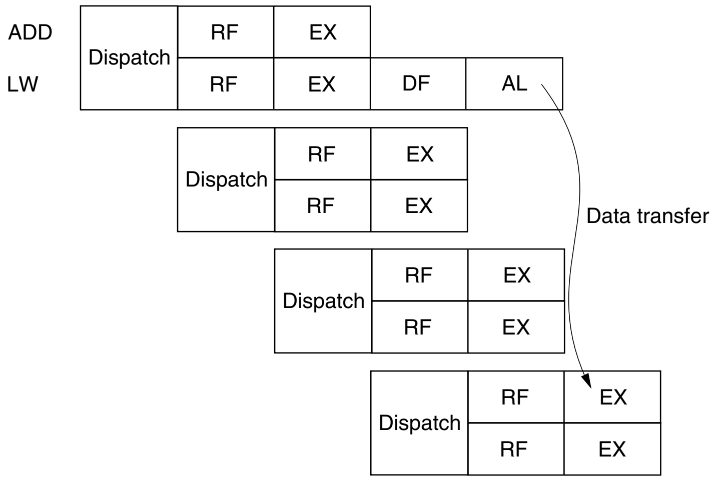
Note The branch delay is covered if there is a valid instruction in the instruction queue.

4.3 Load Delay

The load delay instruction generates a delay until the subsequent instruction can use the result of loading. The processor performs the scheduling necessary for eliminating this delay.

Because the VR5500 uses an out-of-order mechanism to execute instructions, the delay can be covered by executing an instruction that is not dependent upon the load instruction even if a load delay occurs.

Figure 4-5. Load Delay



4.3.1 Non-blocking load

To alleviate the penalty due to a cache miss, the data cache of the VR5500 has a non-blocking mechanism. This allows the VR5500 to continue accessing the cache while holding a cache miss, even if a cache miss occurs as a result of executing a load instruction. This means that the subsequent instructions, including other load instructions, can be consecutively executed if they do not have dependency relationship with the load instruction that has caused the cache miss. Up to four cache misses can be held.

4.4 Exception Processing

If an exception occurs, the instruction that has caused the exception and all the subsequent instructions in the pipeline are canceled.

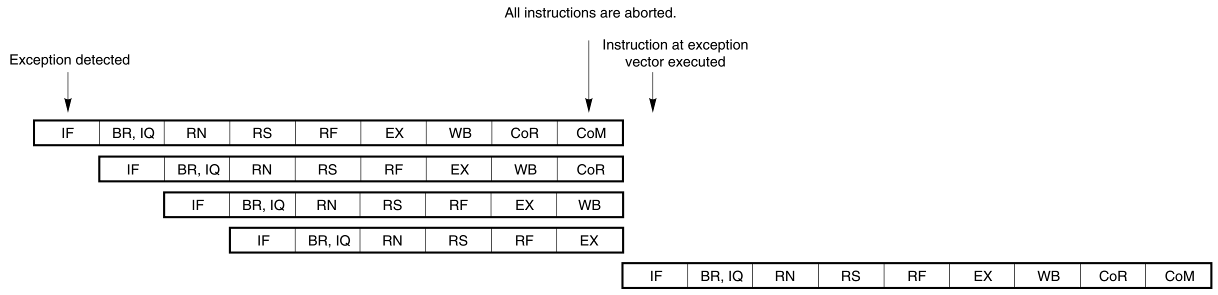
If the instruction responsible for the exception has reached the commit stage, the following three events occur.

- The status and cause of the exception are written to each CP0 register.
- The current PC changes to an appropriate exception vector address.
- The previous exception bit is cleared.

As a result, all the instructions that had been issued before the exception occurred are completed, and all the instructions issued after the instruction responsible for the exception are discarded. Therefore, the EPC indicates the value from which execution can be resumed.

Figure 4-6 shows an example of detecting an exception.

Figure 4-6. Exception Detection


All instructions are aborted.
Exception detected
Instruction at exception vector executed
IF BR, IQ RN RS RF EX WB CoR CoM
IF BR, IQ RN RS RF EX WB CoR
IF BR, IQ RN RS RF EX WB
IF BR, IQ RN RS RF EX
IF BR, IQ RN RS RF EX WB CoR CoM

4.5 Store Buffer

The VR5500 has a 4-entry store buffer (SB) in the DCU so that it can speculatively execute store instructions. The SB temporarily holds the store data of a speculatively executed store instruction, and actually writes data to the cache when that store instruction is committed.

4.6 Write Transaction Buffer

The VR5500 has a write transaction buffer (WTB) that improves the performance of write operations to the external memory. The WTB is used for all transactions of the system interface. The WTB is a four-stage FIFO and can hold data of up to 256 bits. It can therefore hold up to four read requests or one uncached write request or cache line writeback.

The entire WTB is used for writeback data in case of a cache miss that requires writeback, and the processor can perform processing in parallel with memory updating. In the case of storing in an uncached area and a write-through store, processing by the WTB and writing to the memory by the CPU are not executed in parallel. If the WTB is full, the subsequent store operation is stalled until there is a space available.

The WTB cannot be read or written by software.

CHAPTER 5 MEMORY MANAGEMENT SYSTEM

The VR5500 has a memory management unit (MMU) that uses a high-speed translation lookaside buffer (TLB) which translates virtual addresses into physical addresses. This chapter explains in detail the operation of the TLB, the CP0 registers used as a software interface with the TLB, and the memory mapping method used to translate virtual addresses into physical addresses.

5.1 Processor Modes

5.1.1 Operating modes

The VR5500 has the following three operating modes with priority assigned by the system to these modes, starting with the one at the top.

- Kernel mode (highest priority): In this mode, all the registers can be accessed and changed. The nucleus of the operating system operates in the kernel mode.
- Supervisor mode: The priority of this mode is lower than that of the kernel mode. This mode is used for sections assigned a lower importance by the operating system.
- User mode (lowest priority): This mode prevents users from interfering with each other.

The basic operating mode of the processor is the user mode. When the processor processes an error (when the ERL bit is set) or an exception (when the EXL bit is set), it enters the kernel mode.

The operating mode of the processor is set by the KSU field of the Status register and the ERL and EXL bits. Table 5-1 shows the three operating modes, and the setting of the Status register related to the error and exception levels. A blank indicates that any setting is possible.

Table 5-1. Operating Modes

Status Register Bit			Operating Mode
KSU(1:0)	EXL	ERL	
10	0	0	User mode
01	0	0	Supervisor mode
00	0	0	Kernel mode
		1	
	1		

In the case of an exception or error, the EXL and ERL bits are set regardless of the setting of the KSU field. When these bits are set, interrupts are disabled. If the EXL bit is cleared by an exception handler to enable processing of multiple interrupts, for example, the processor enters the mode set by the KSU field from the kernel mode. Therefore, change the KSU field before clearing the EXL bit by an exception handler.

5.1.2 Instruction set modes

The instruction set mode of the processor determines which instructions are enabled. By default, the MIPS IV instruction set architecture (ISA) is implemented. However, MIPS III ISA or MIPS I/II ISA can also be used to maintain compatibility with a conventional machine.

The instruction set mode is set by bits UX, SX, and XX of the Status register. Table 5-2 shows the setting of the Status register related to the instruction set mode. A blank indicates that any setting is possible.

Table 5-2. Instruction Set Modes

Operating Mode	Status Register Bit			Instruction Set Mode		
	UX	SX	XX	MIPS I, II	MIPS III	MIPS IV
User mode	0		0	Can be used	Cannot be used	Cannot be used
	0		1	Can be used	Cannot be used	Can be used
	1		0	Can be used	Can be used	Cannot be used
	1		1	Can be used	Can be used	Can be used
Supervisor mode		0		Can be used	Cannot be used	Can be used
		1		Can be used	Can be used	Can be used
Kernel mode				Can be used	Can be used	Can be used

5.1.3 Addressing modes

The addressing mode of the processor determines whether a 32-bit or 64-bit memory address is to be generated. Refer to Table 5-3 for the settings of the following addressing modes.

- In the kernel mode, 64-bit addressing is enabled by the KX bit. All the instructions are always valid.
- In the supervisor mode, 64-bit addressing and the MIPS III instructions are enabled by the SX bit.
- In the user mode, 64-bit addressing and the MIPS III instructions are enabled by the UX bit. In addition, the MIPS IV instructions are enabled by the XX bit.

Table 5-3. Addressing Modes

Operating Mode	Status Register Bit			Addressing Mode
	UX	SX	KX	
User mode	0			32-bit
	1			64-bit
Supervisor mode		0		32-bit
		1		64-bit
Kernel mode			0	32-bit
			1	64-bit

5.2 Translation Lookaside Buffer (TLB)

Virtual addresses are translated into physical addresses using an on-chip TLB[Note]. The on-chip TLB is a fully-associative memory that holds 48 entries, which provide mapping to odd/even page in pairs for one entry. These pages can have ten different sizes, 4 K, 16 K, 64 K, 256 K, 1 M, 4 M, 16 M, 64 M, 256 M, and 1 G, and can be specified for each entry.

If it is supplied with a virtual address, each TLB entry checks the 48 entries simultaneously to see whether they match the virtual addresses that are provided with the ASID field and saved in the EntryHi register.

If there is a virtual address match (hit) in the TLB, a physical address is created from the physical page number and the offset value.

If no match occurs (miss), an exception is taken and software refills the TLB entry from the page table resident in memory. The software writes to an entry selected using the Index register or a random entry indicated in the Random register.

If more than one entry in the TLB matches the virtual address being translated, the operation is undefined. In this case, the TS bit of the Status register is set to 1, and a TLB refill exception occurs regardless of the valid bit status of the TLB entry. Replace the TLB entry using the exception handler and clear the TS bit to 0.

Note Depending on the address space, virtual addresses may be translated to physical addresses without using a TLB. For example, address translation for the kseg0 or kseg1 address space does not use mapping. The physical addresses of these address spaces are determined by subtracting the base address of the address space from the virtual addresses.

(1) Micro TLB

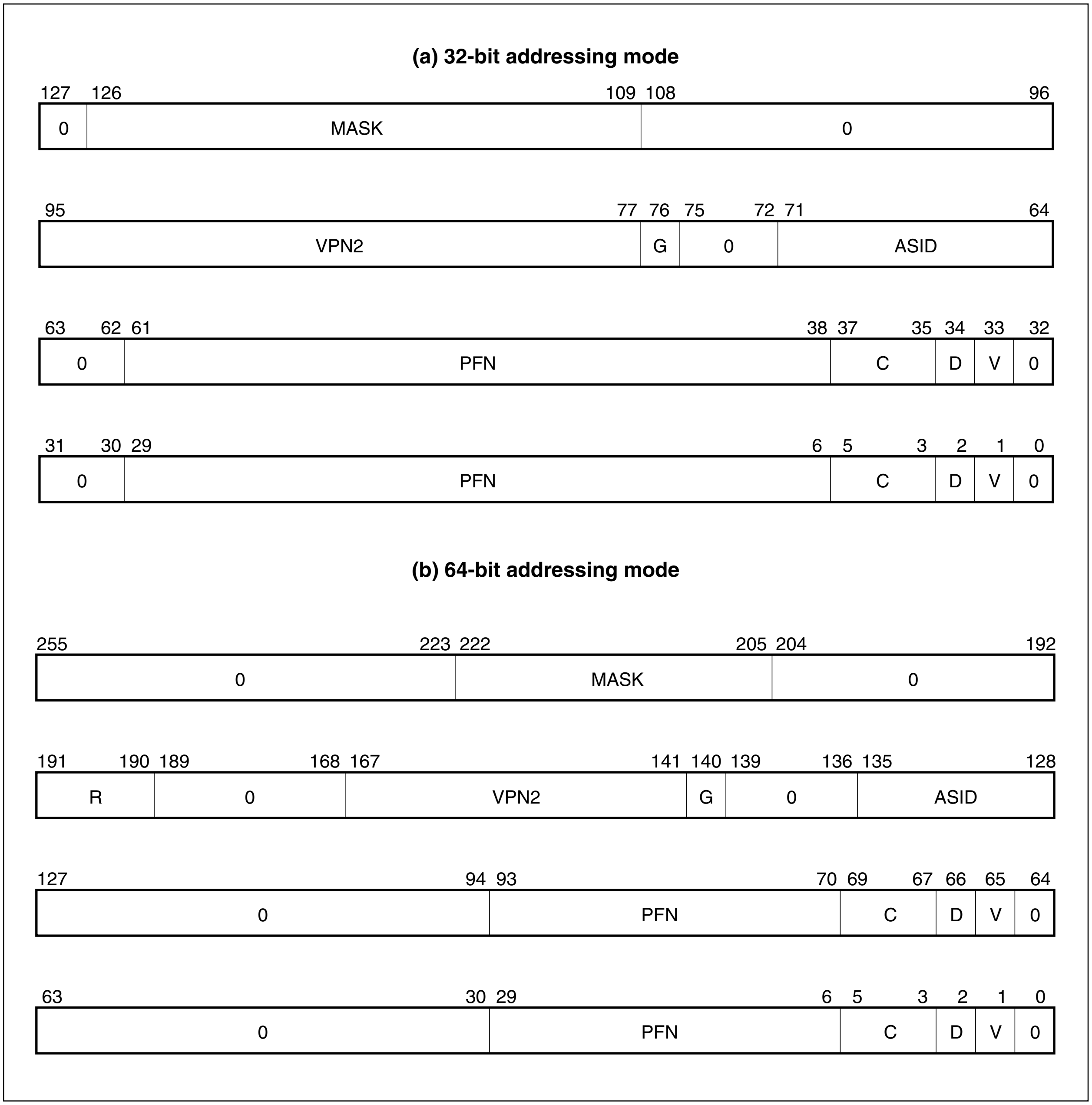
The VR5500 has two 4-entry micro TLBs in addition to a 48-entry TLB. These TLBs are also full-associative memories and are respectively dedicated to the translation of instruction and data addresses.

The micro TLBs are a subset of the TLB, and the page size can be set for each entry in the same manner as the TLB. If a mismatch occurs in a micro TLB, the entries are replaced with new entries from the TLB by using a dummy LRU (Least Recently Used) algorithm. The pipeline stalls while an entry is being transferred from the TLB.

5.2.1 Format of TLB entry

Figure 5-1 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 5-1. Format of TLB Entry



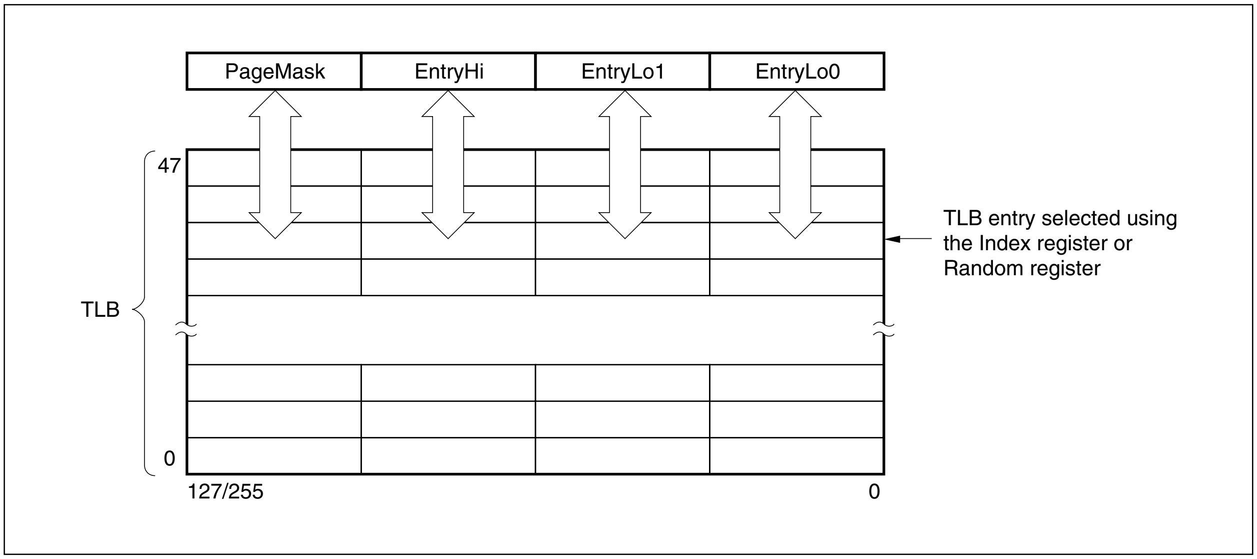
(a) 32-bit addressing mode

(b) 64-bit addressing mode

The format of the EntryHi, EntryLo0, EntryLo1, and PageMask registers is almost the same as a TLB entry. However, the bit at the position corresponding to the TLB G bit is reserved (0) in the EntryHi register. The bit at the position corresponding to the G bit of the EntryLo register is reserved (0) in the TLB. For details of other fields, refer to the description of the relevant registers.

The contents of the TLB entries can be read or written via the EntryHi, EntryLo0, EntryLo1, and PageMask registers using a TLB manipulation instruction, as shown in Figure 5-2. The target entry is either one specified by the Index register, or a random entry indicated by the Random register.

Figure 5-2. Outline of TLB Manipulation


PageMask
EntryHi
EntryLo1
EntryLo0
47
TLB
0
127/255
0
TLB entry selected using the Index register or Random register

5.2.2 TLB instructions

The instructions used for TLB control are described below.

(1) TLBP (Translation lookaside buffer probe)

The TLBP instruction loads the Index register with a TLB entry number that matches the contents of the EntryHi register. If there is no matching TLB entry, the most significant bit of the Index register is set (1).

(2) TLBR (Translation lookaside buffer read)

The TLBR instruction writes the EntryHi, EntryLo0, EntryLo1, and PageMask registers with the contents of the TLB entry indicated by the content of the Index register.

(3) TLBWI (Translation lookaside buffer write index)

The TLBWI instruction writes the contents of the EntryHi, EntryLo0, EntryLo1, and PageMask registers to the TLB entry indicated by the contents of the Index register.

(4) TLBWR (Translation lookaside buffer write random)

The TLBWR instruction writes the contents of the EntryHi, EntryLo0, EntryLo1, and PageMask registers to the TLB entry indicated by the contents of the Random register.

5.2.3 TLB exception

If there is no TLB entry that matches the virtual address, a TLB Refill exception occurs. If the access control bits (D and V) indicate that the access is not valid, a TLB modified or TLB invalid exception occurs.

Refer to **CHAPTER 6 EXCEPTION PROCESSING** for details of TLB exceptions.

5.3 Virtual-to-Physical Address Translation

Translating a virtual address to a physical address begins by comparing the virtual address sent from the processor with the virtual addresses of all entries in the TLB. First, one of the following comparisons is made for the virtual page number (VPN) of the address.

- In 32-bit mode: The higher bits[Note] of the virtual address are compared to the contents of the VPN2 (virtual page number divided by two) of each TLB entry.
- In 64-bit mode: The higher bits[Note] of the virtual address are compared to the contents of the R and the VPN2 (virtual page number divided by two) of each TLB entry.

Note The number of bits differs depending on the page size.
The table below shows examples of the higher bits of the virtual address with page sizes of 16 MB and 4 KB.

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

When there is an entry which has a field with the same contents in this comparison, if either of the following applies, a match occurs.

- The Global bit (G) of the TLB entry is set to 1
- 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.

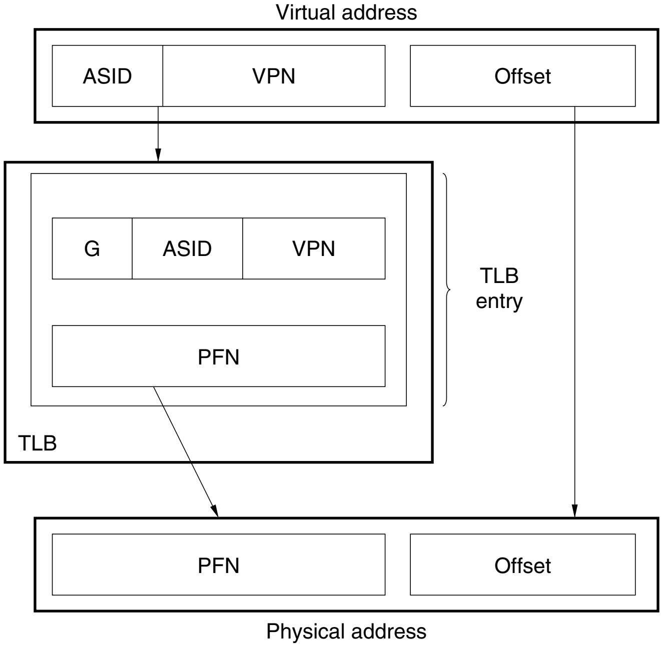
If the matching entry is in the TLB, the physical address and access control bits (C, D, V) are read out from that entry. In order to perform valid address translation, the entry's V bit must be set (1), but this is unrelated to the determination of the matching TLB entry. An offset value is added to the physical address that was read out. The offset indicates an address inside the page frame space. The offset part bypasses the TLB and the lower bits of the virtual address are output as are.

If there is no match, the processor core generates a TLB refill exception and references the page table in the memory in which the virtual addresses and physical addresses have been paired, the contents of which are then written to the TLB via software.

Figure 5-3 shows a summary of address translation, and Figure 5-4 the TLB address translation flowchart.

Figure 5-3. Virtual-to-Physical Address Translation

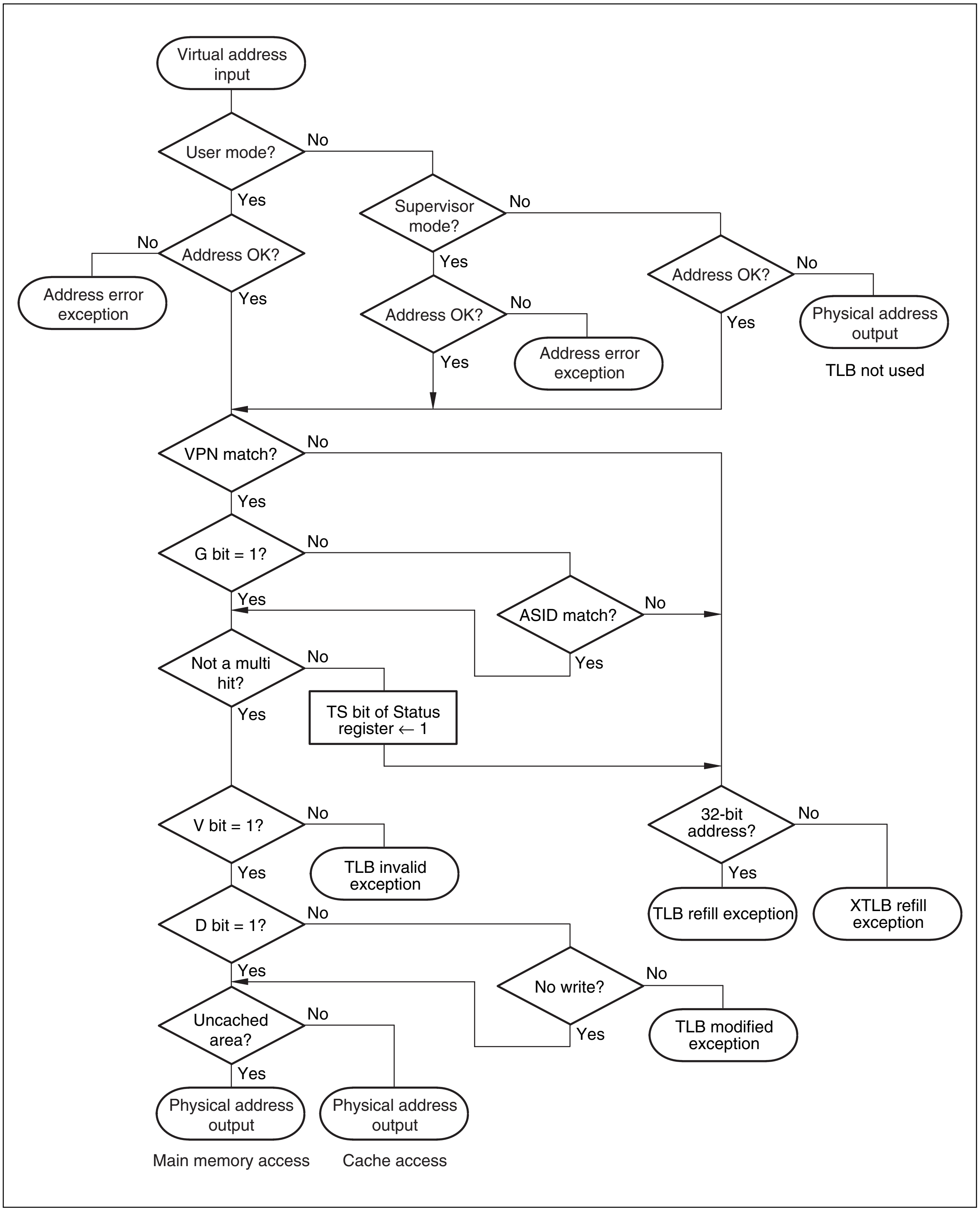
<1> The virtual address page number (VPN, higher bits in the address) and ASID are compared with the corresponding area in the TLB.



<2> If there is an entry matched, the page frame number (PFN) representing the higher bits of the physical address is output from the TLB.

<3> The offset is then added to the PFN, which bypasses the TLB.

Figure 5-4. TLB Address Translation

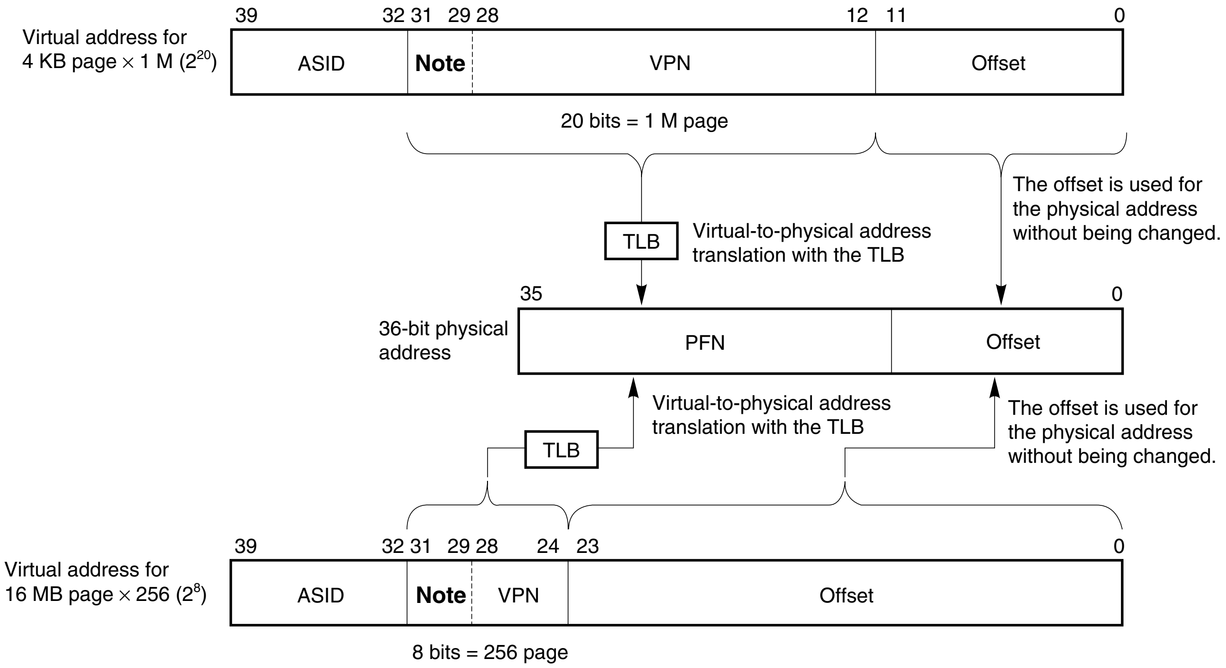


5.3.1 32-bit addressing mode address translation

Figure 5-5 shows the virtual-to-physical address translation in the 32-bit mode addressing mode. The page sizes can be selected from the ten pattern, 4 KB (12 bits) to 1 GB (30 bits) in 4-multiply units.

- Shown at the top of Figure 5-5 is the virtual address space in which the page size is 4 KB and the offset is 12 bits. The 20 bits excluding the ASID field represent the virtual page number (VPN), enabling selection of a page table of 1 M entries.
- Shown at the bottom of Figure 5-5 is the virtual address space in which the page size is 16 MB and the offset is 24 bits. The 8 bits excluding the ASID field represent the VPN, enabling selection of a page table of 256 entries.

Figure 5-5. Virtual Address Translation in 32-Bit Addressing Mode


39
32 31
29 28
12
11
0
Virtual address for
4 KB page × 1 M (2^{20})
ASID
Note
VPN
Offset
20 bits = 1 M page
TLB
Virtual-to-physical address translation with the TLB
The offset is used for the physical address without being changed.
35
0
36-bit physical address
PFN
Offset
Virtual-to-physical address translation with the TLB
TLB
The offset is used for the physical address without being changed.
39
32 31
29 28
24
23
0
Virtual address for
16 MB page × 256 (2^8)
ASID
Note
VPN
Offset
8 bits = 256 page

Note User, supervisor, or kernel address space is selected by bits 31 to 29 of the virtual address.

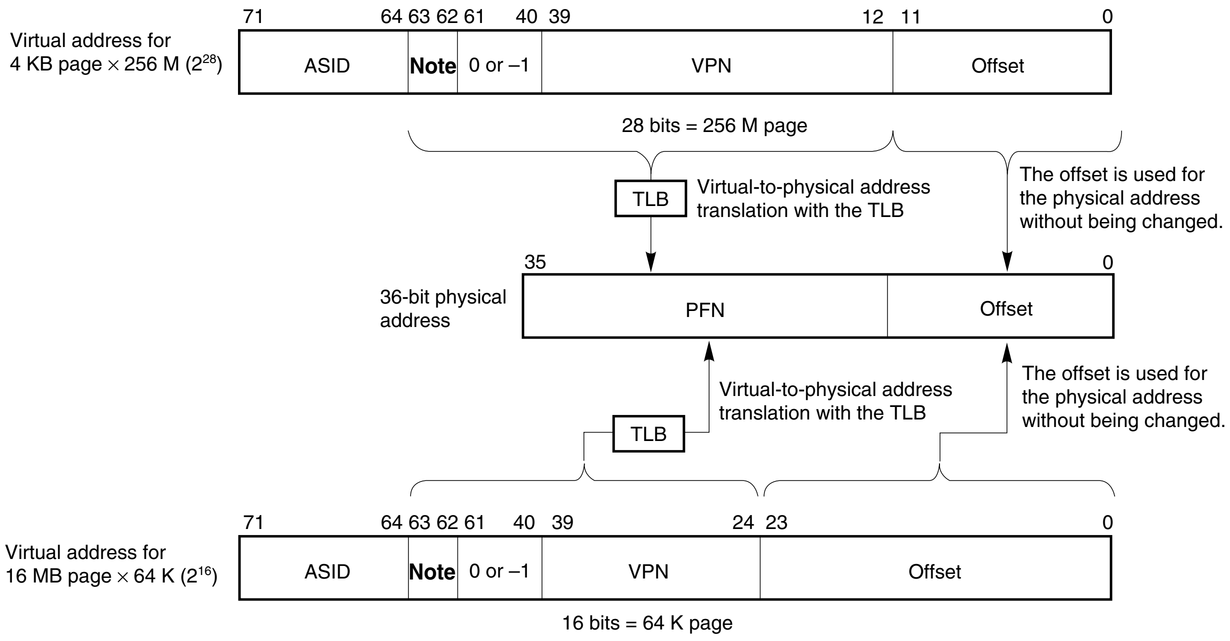
Remark Bits 35 to 32 of the physical address are not output in the 32-bit bus mode.

5.3.2 64-bit addressing mode address translation

Figure 5-6 shows the virtual-to-physical address translation in the 64-bit mode addressing mode. The page sizes can be selected from the ten pattern, 4 KB (12 bits) to 1 GB (30 bits) in 4-multiply units.

- Shown at the top of Figure 5-6 is the virtual address space in which the page size is 4 KB and the offset is 12 bits. The 28 bits excluding the ASID field represent the virtual page number (VPN), enabling selection of a page table of 256 M entries.
- Shown at the bottom of Figure 5-6 is the virtual address space in which the page size is 16 MB and the offset is 24 bits. The 16 bits excluding the ASID field represent the VPN, enabling selection of a page table of 64 K entries.

Figure 5-6. Virtual Address Translation in 64-Bit Addressing Mode


Virtual address for
4 KB page × 256 M (2^{28})
71
64 63 62 61
40 39
12 11
0
ASID
Note
0 or –1
VPN
Offset
28 bits = 256 M page
TLB
Virtual-to-physical address translation with the TLB
The offset is used for the physical address without being changed.
36-bit physical address
35
0
PFN
Offset
Virtual-to-physical address translation with the TLB
TLB
The offset is used for the physical address without being changed.
Virtual address for
16 MB page × 64 K (2^{16})
71
64 63 62 61
40 39
24 23
0
ASID
Note
0 or –1
VPN
Offset
16 bits = 64 K page

Note User, supervisor, or kernel address space is selected by bits 63 and 62 of the virtual address.

Remark Bits 35 to 32 of the physical address are not output in the 32-bit bus mode.

5.4 Virtual Address Space

The address space of the CPU is extended in memory management system, by translating huge virtual memory addresses into physical addresses.

The VR5500 has three types of virtual address spaces: user, supervisor, and kernel. The addressing mode of each of these virtual address spaces can be set to 32-bit or 64-bit mode. In the 32-bit addressing mode, a virtual address is 32 bits wide, and the maximum user area is 2 GB (2^{31} bytes). In the 64-bit addressing mode, the virtual address width is 64 bits and the maximum user area is 1 TB (2^{40} bytes).

The virtual address is extended with an address space identifier (ASID) (refer to **Figures 5-5** and **5-6**), which reduces the frequency of TLB flushing when switching contexts. This 8-bit ASID is in the CP0 EntryHi register, and the Global (G) bit is in the EntryLo0 and EntryLo1 registers, described later in this chapter.

When the system interface is in the 32-bit bus mode, the VR5500 uses 32-bit physical addresses. Consequently, the physical address space is 4 GB. In the 64-bit bus mode, the physical address space is 128 GB because the VR5500 uses 36-bit physical address.

Caution **If the system interface of the VR5500 is in the 32-bit bus mode, an address error exception does not occur and physical addresses are processed with bits 35 to 32 ignored, even if the space is referenced so that bits 35 to 32 of the physical address are a value other than 0.**

5.4.1 User mode virtual address space

In user mode, a 2 GB (2^{31} bytes) virtual address space (useg) can be used in 32-bit addressing mode. In 64-bit addressing mode, a 1 TB (2^{40} bytes) virtual address space (xuseg) can be used.

useg and xuseg can be referenced via the TLB. Whether a cache is used or not is determined for each page by the TLB entry (depending on the C bit setting in the TLB entry).

The user address space can be accessed in supervisor mode and kernel mode.

The user segment starts at address 0 and the current active user process resides in either useg (in 32-bit addressing mode) or xuseg (in 64-bit addressing mode).

The VR5500 operates in user mode when the Status register contains the following bit-values.

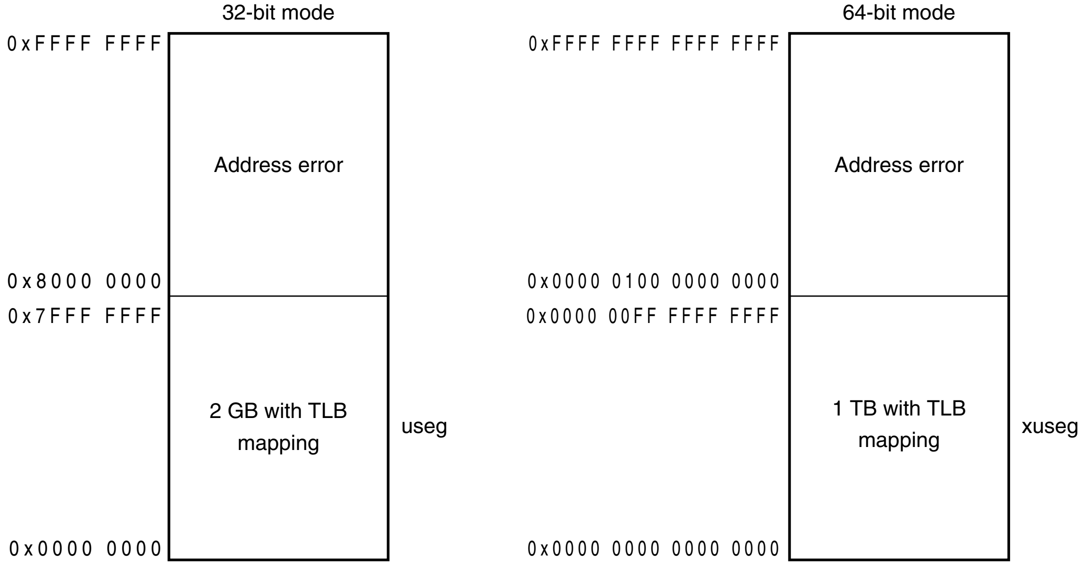
- KSU field = 10
- EXL bit = 0
- ERL bit = 0

In addition, the UX bit in the Status register selects addressing mode as follows.

- When UX bit = 0: 32-bit useg space is selected.
  A TLB mismatch is processed by the 32-bit TLB refill exception handler.
- When UX bit = 1: 64-bit xuseg space is selected.
  A TLB mismatch is processed by the 64-bit XTLB refill exception handler.

Figure 5-7 shows user mode address mapping and Table 5-4 lists the characteristics of the user segments.

Figure 5-7. User Mode Address Space



Remark When a 2's complement overflow occurs in the address calculation, the calculated address is invalid and the result is not defined.

Table 5-4. 32-Bit and 64-Bit User Mode Segments

Addressing Mode	Address Bit Value	Status Register Bit Value				Segment	Address Range	Size
		KSU	EXL	ERL	UX	Name		
32-bit	A31 = 0	Any	0	0	0	useg	0x0000 0000 to 0x7FFF FFFF	2 GB (2^{31} bytes)
64-bit	A(63:40) = 0		0	0	1	xuseg	0x0000 0000 0000 0000 to 0x0000 00FF FFFF FFFF	1 TB (2^{40} bytes)

(1) useg (32-bit mode)

When the UX bit of in the Status register is 0 and the most significant bit of the virtual address is 0, this virtual address space is labeled useg. Any attempt to reference an address with the most-significant bit of 1 causes an address error exception (refer to **CHAPTER 6 EXCEPTION PROCESSING**).

(2) xuseg (64-bit mode)

When the UX bit of the Status register is 1 and bits 63 to 40 of the virtual address are all 0, this virtual address space is labeled xuseg, and 1 terabyte (2^{40} bytes) of the user address space can be used. Any attempt to reference an address with bits 63 to 40 equal to 1 causes an address error exception (refer to **CHAPTER 6 EXCEPTION PROCESSING**).

5.4.2 Supervisor mode virtual address space

Supervisor mode layers the execution of operating systems. Kernel operating systems at the highest layer are executed in kernel mode, and the rest of the operating system is executed in supervisor mode.

suseg, sseg, xsuseg, xsseg, and csseg (all the spaces) can be referenced via the TLB. Whether a cache is used or not is determined for each page by the TLB entry (depending on the C bit setting in the TLB entry).

The supervisor address space can be accessed in kernel mode.

The processor operates in supervisor mode when the Status register contains the following bit-values.

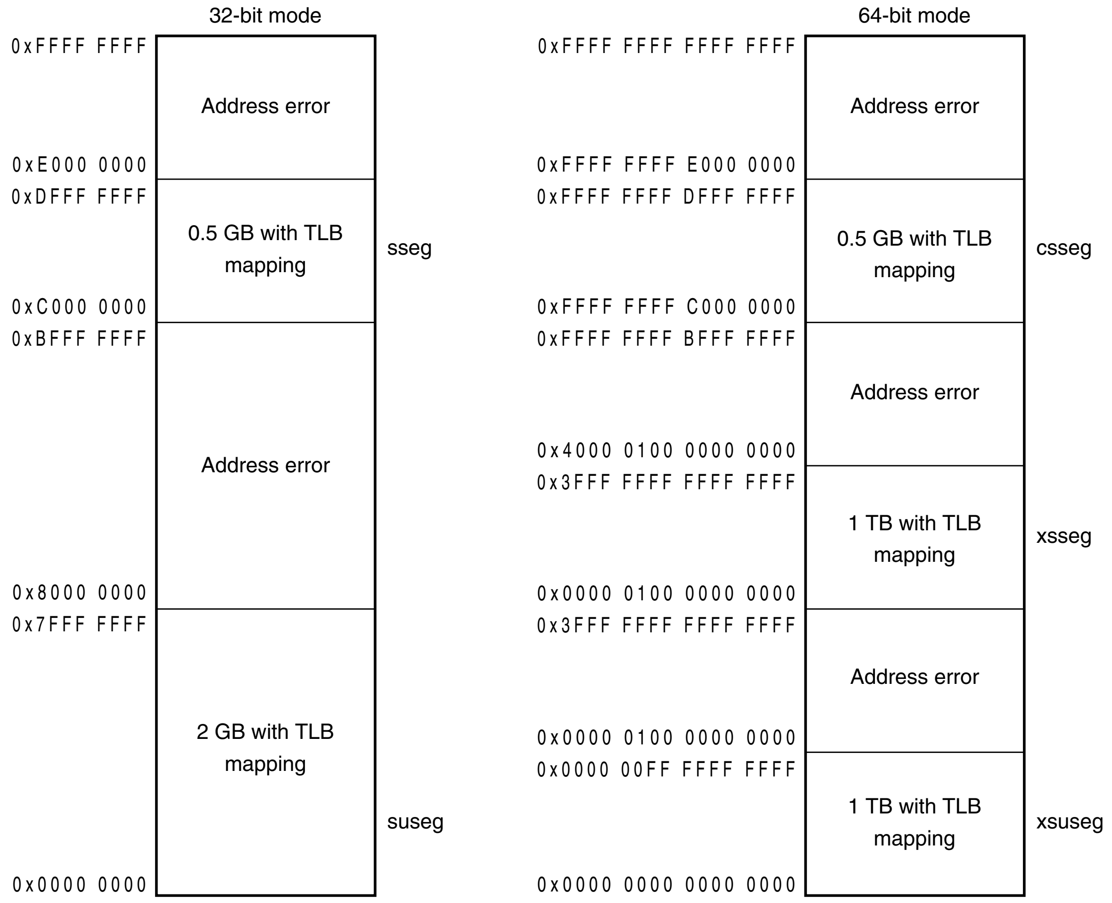
- KSU field = 01
- EXL bit = 0
- ERL bit = 0

In addition, the SX bit in the Status register selects addressing mode as follows.

- When SX bit = 0: 32-bit supervisor space
  A TLB mismatch is processed by the 32-bit TLB refill exception handler.
- When SX bit = 1: 64-bit supervisor space
  A TLB mismatch is processed by the 64-bit XTLB refill exception handler.

Figure 5-8 shows supervisor mode address mapping and Table 5-5 lists the characteristics of the segments in supervisor mode.

Figure 5-8. Supervisor Mode Address Space



Remark When a 2’s complement overflow occurs in the address calculation, the calculated address is invalid and the result is not defined.

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

Addressing Mode	Address Bit Value	Status Register Bit Value				Segment Name	Address Range	Size
		KSU	EXL	ERL	SX			
32-bit	A31 = 0	01 or 00	0	0	0	suseg	0x0000 0000 to 0x7FFF FFFF	2 GB (2^{31} bytes)
	A(31:29) = 110	01 or 00	0	0	0	sseg	0xC000 0000 to 0xDFFF FFFF	512 MB (2^{29} bytes)
64-bit	A(63:62) = 00	01 or 00	0	0	1	xsuseg	0x0000 0000 0000 0000 to 0x0000 00FF FFFF FFFF	1 TB (2^{40} bytes)
	A(63:62) = 01	01 or 00	0	0	1	xsseg	0x4000 0000 0000 0000 to 0x4000 00FF FFFF FFFF	1 TB (2^{40} bytes)
	A(63:62) = 11	01 or 00	0	0	1	csseg	0xFFFF FFFF C000 0000 to 0xFFFF FFFF DFFF FFFF	512 MB (2^{29} bytes)

(1) suseg (32-bit supervisor mode, user space)

When the SX bit of the Status register is 0 and the most-significant bit of the virtual address space is 0, the suseg virtual address space is selected; it covers 2 GB (2^{31} bytes) 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.

(2) sseg (32-bit supervisor mode, supervisor space)

When the SX bit of the Status register is 0 and the higher 3 bits of the virtual address space are 110, the sseg virtual address space is selected; it covers 512 MB (2^{29} bytes) of the current supervisor virtual address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

(3) xsuseg (64-bit supervisor mode, user space)

When the SX bit of the Status register is 1 and bits 63 and 62 of the virtual address space are 00, the xsuseg virtual address space is selected; it covers 1 TB (2^{40} bytes) 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.

(4) xsseg (64-bit supervisor mode, current supervisor space)

When the SX bit of the Status register is 1 and bits 63 and 62 of the virtual address space are 01, the xsseg virtual address space is selected; it covers 1 TB (2^{40} bytes) of the current supervisor virtual address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

(5) csseg (64-bit supervisor mode, separate supervisor space)

When the SX bit of the Status register is 1 and bits 63 and 62 of the virtual address space are 11, the csseg virtual address space is selected; it covers 512 MB (2^{29} bytes) of the separate supervisor virtual address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

5.4.3 Kernel mode virtual address space

If the Status register satisfies any of the following conditions, the processor runs in kernel mode.

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

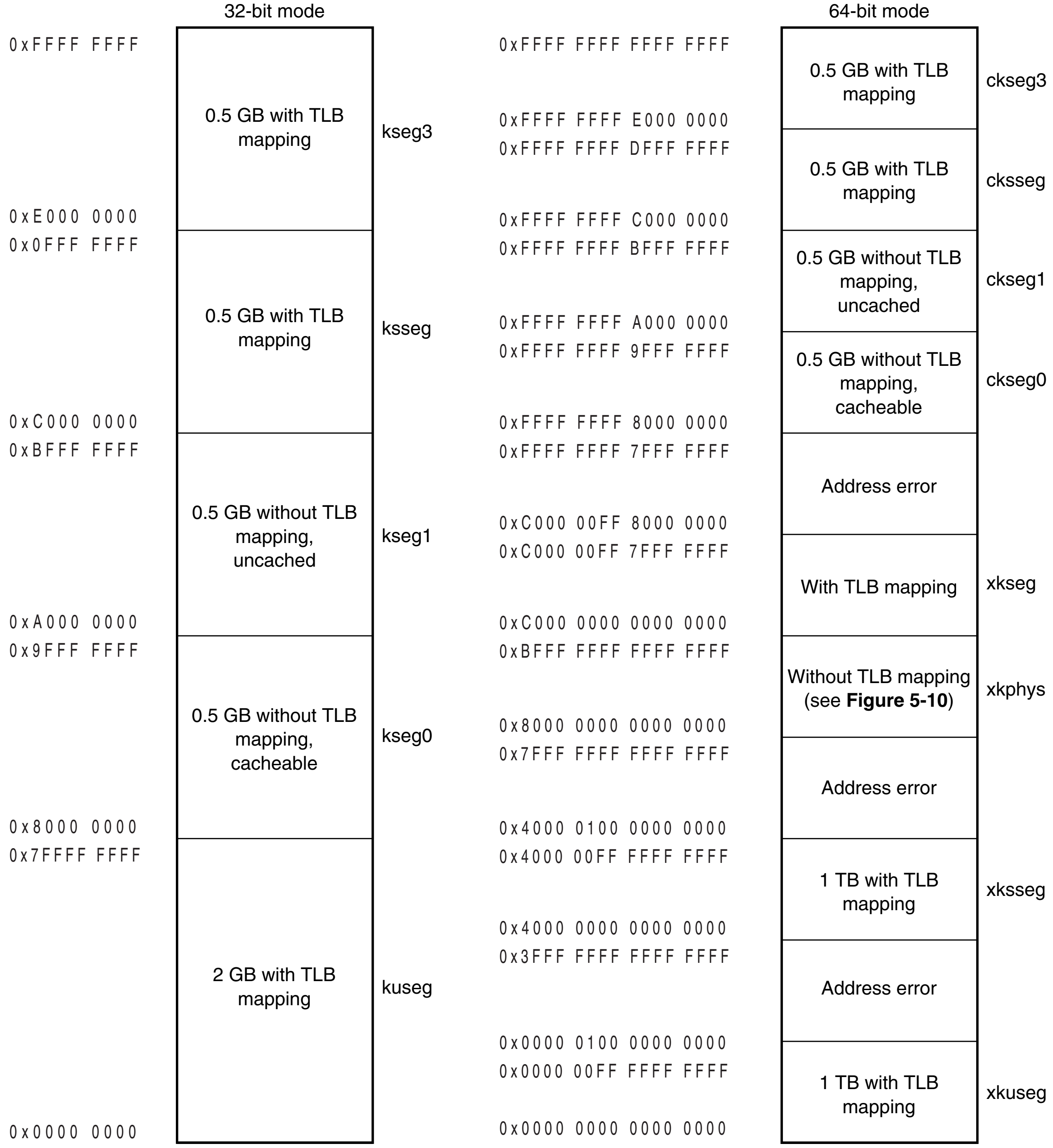
The addressing width in kernel mode varies according to the state of the KX bit of the Status register, as follows.

- When KX = 0: 32-bit kernel space is selected.
  A TLB mismatch is processed by the 32-bit TLB refill exception handler.
- When KX = 1: 64-bit kernel space is selected.
  A TLB mismatch is processed by the 32-bit XTLB refill exception handler.

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

Kernel mode virtual address space is divided into regions differentiated by the higher bits of the virtual address, as shown in Figure 5-9. Table 5-6 lists the characteristics of the 32-bit kernel mode segments, and Table 5-7 lists the characteristics of the 64-bit kernel mode segments.

Figure 5-9. Kernel Mode Address Space



Remark When a 2's complement overflow occurs in the address calculation, the calculated address is invalid and the result is not defined.

Figure 5-10. xkphys Area Address Space

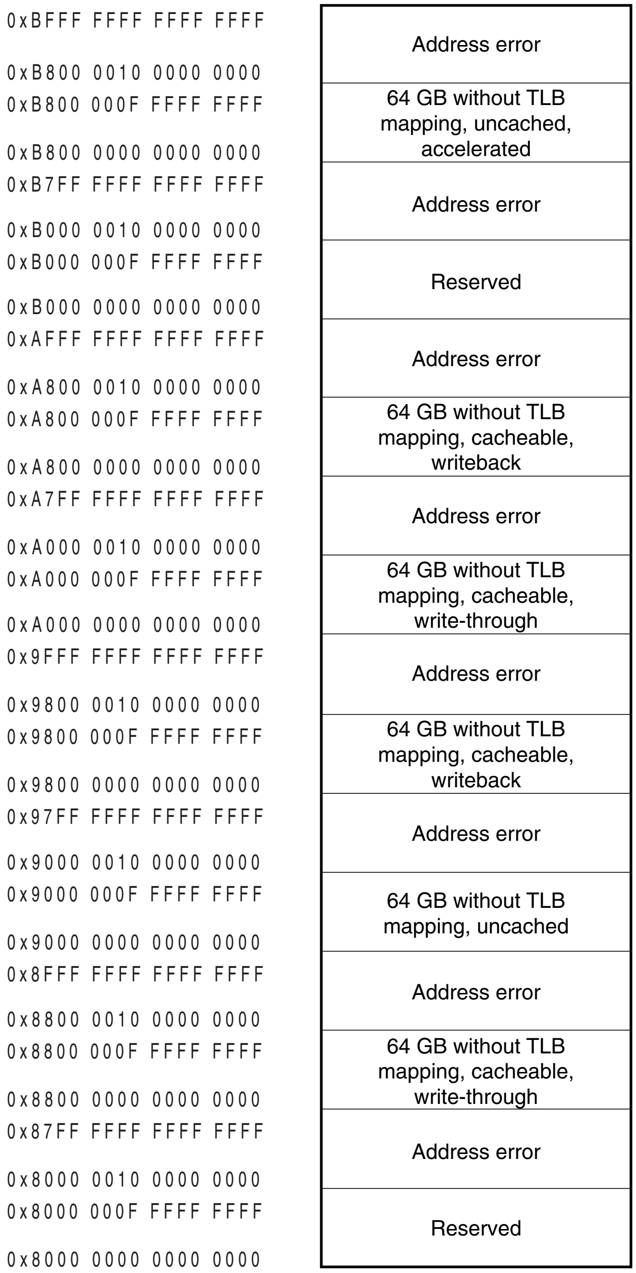


Table 5-6. 32-Bit Kernel Mode Segments

Address Bit Value	Status Register Bit Value				Segment Name	Virtual Address	Physical Address	Size
	KSU	EXL	ERL	KX				
A31 = 0	KSU = 00 or EXL = 1 or ERL = 1			0	kuseg	0x0000 0000 to 0x7FFF FFFF	TLB map	2 GB (2^{31} bytes)
A(31:29) = 100				0	kseg0	0x8000 0000 to 0x9FFF FFFF	0x0000 0000 to 0x1FFF FFFF	512 MB (2^{29} bytes)
A(31:29) = 101				0	kseg1	0xA000 0000 to 0xBFFF FFFF	0x0000 0000 to 0x1FFF FFFF	512 MB (2^{29} bytes)
A(31:29) = 110				0	ksseg	0xC000 0000 to 0xDFFF FFFF	TLB map	512 MB (2^{29} bytes)
A(31:29) = 111				0	kseg3	0xE000 0000 to 0xFFFF FFFF	TLB map	512 MB (2^{29} bytes)

(1) kuseg (32-bit kernel mode, user space)

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

References to kuseg are mapped through TLB. Whether cache can be used or not is determined by bit C of each page's TLB entry.

If the ERL bit of the Status register is 1, the user address space is assigned 2 GB (2^{31} bytes) without TLB mapping and becomes unmapped (with virtual addresses being used as physical addresses) and uncached.

(2) kseg0 (32-bit kernel mode, kernel space 0)

When the KX bit of the Status register is 0 and the higher 3 bits of the virtual address space are 100, the kseg0 virtual address space is selected; it is the current 512 MB (2^{29} bytes) physical space.

References to kseg0 are not mapped through TLB; the physical address selected is defined by subtracting 0x8000 0000 from the virtual address. The K0 field of the Config register controls cacheability (see **5.5.8 Config register (16)**).

(3) kseg1 (32-bit kernel mode, kernel space 1)

When the KX bit of the Status register is 0 and the higher 3 bits of the virtual address space are 101, the kseg1 virtual address space is selected; it is the current 512 MB (2^{29} bytes) physical space.

References to kseg1 are not mapped through TLB; the physical address selected is defined by subtracting 0xA000 0000 from the virtual address. Caches are disabled for accesses to these addresses, and main memory (or memory-mapped I/O device registers) is accessed directly.

(4) ksseg (32-bit kernel mode, supervisor space)

When the KX bit of the Status register is 0 and the higher 3 bits of the virtual address space are 110, the ksseg virtual address space is selected; it is the current 512 MB (2^{29} bytes) virtual address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

References to ksseg are mapped through TLB. Whether cache can be used or not is determined by bit C of each page's TLB entry.

(5) kseg3 (32-bit kernel mode, kernel space 3)

When the KX bit of the Status register is 0 and the higher 3 bits of the virtual address space are 111, the kseg3 virtual address space is selected; it is the current 512 MB (2^{29} bytes) kernel virtual space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

References to kseg3 are mapped through TLB. Whether cache can be used or not is determined by bit C of each page's TLB entry.

Table 5-7. 64-Bit Kernel Mode Segments

Address Bit Value	Status Register Bit Value				Segment	Virtual Address	Physical Address	Size
	KSU	EXL	ERL	KX	Name			
A(63:62) = 00	KSU = 00 or EXL = 1 or ERL = 1			1	xkuseg	0x0000 0000 0000 0000 to 0x0000 00FF FFFF FFFF	TLB map	1 TB (2^{40} bytes)
A(63:62) = 01				1	xksseg	0x4000 0000 0000 0000 to 0x4000 00FF FFFF FFFF	TLB map	1 TB (2^{40} bytes)
A(63:62) = 10				1	xkphys	0x8000 0000 0000 0000 to 0xBFFF FFFF FFFF FFFF	0x0000 0000 0000 to 0x000F FFFF FFFF	2^{36} bytes (see **(8)**)
A(63:62) = 11				1	xkseg	0xC000 0000 0000 0000 to 0xC000 00FF 7FFF FFFF	TLB map	2^{40} to 2^{31} bytes
A(63:62) = 11, A(63:31) = −1				1	ckseg0	0xFFFF FFFF 8000 0000 to 0xFFFF FFFF 9FFF FFFF	0x0000 0000 to 0x1FFF FFFF	512 MB (2^{29} bytes)
A(63:62) = 11, A(63:31) = −1				1	ckseg1	0xFFFF FFFFA000 0000 to 0xFFFF FFFF BFFF FFFF	0x0000 0000 to 0x1FFF FFFF	512 MB (2^{29} bytes)
A(63:62) = 11, A(63:31) = −1				1	cksseg	0xFFFF FFFF C000 0000 to 0xFFFF FFFF DFFF FFFF	TLB map	512 MB (2^{29} bytes)
A(63:62) = 11, A(63:31) = −1				1	ckseg3	0xFFFF FFFF E000 0000 to 0xFFFF FFFF FFFF FFFF	TLB map	512 MB (2^{29} bytes)

(6) xkuseg (64-bit kernel mode, user space)

When the KX bit of the Status register is 1 and bits 63 and 62 of the virtual address space are 00, the xkuseg virtual address space is selected; it is the 1 TB (2^{40} bytes) current user address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

References to xkuseg are mapped through TLB. Whether cache can be used or not is determined by bit C of each page's TLB entry.

If the ERL bit of the Status register is 1, the user address space is assigned 2 GB (2^{31} bytes) without TLB mapping and becomes unmapped (with virtual addresses being used as physical addresses) and uncached.

(7) xksseg (64-bit kernel mode, normal supervisor space)

When the KX bit of the Status register is 1 and bits 63 and 62 of the virtual address space are 01, the xksseg address space is selected; it is the 1 TB (2^{40} bytes) normal supervisor address space. The virtual address is extended with the contents of the 8-bit ASID field to form a unique virtual address.

References to xksseg are mapped through TLB. Whether cache can be used or not is determined by bit C of each page's TLB entry.

(8) xkphys (64-bit kernel mode, physical spaces)

When the KX bit of the Status register is 1and bits 63 and 62 of the virtual address space are 10, the virtual address space is called xkphys and one of the 8 spaces of the unmapped area is selected. Internally, bits 35 to 0 of the virtual address are used for the physical address as is. If any of bits 58 to 32 of the address is 1, an attempt to access that address results in an address error.

Bits 61 to 59 of the virtual address indicate the cache usability of each space and its attribute (algorithm). Table 5-8 shows cache algorithm corresponding to 8 address spaces.

Table 5-8. Cache Algorithm and xkphys Address Space

Bits 61 to 59	Cache Usability and Algorithm	Address
0	Reserved	0x8000 0000 0000 0000 to 0x8000 000F FFFF FFFF
1	Cacheable, write-through, write-allocated	0x8800 0000 0000 0000 to 0x8800 000F FFFF FFFF
2	Uncached	0x9000 0000 0000 0000 to 0x9000 000F FFFF FFFF
3	Cacheable, writeback	0x9800 0000 0000 0000 to 0x9800 000F FFFF FFFF
4	Cacheable, write-through, write-allocated	0xA000 0000 0000 0000 to 0xA000 000F FFFF FFFF
5	Cacheable, writeback	0xA800 0000 0000 0000 to 0xA800 000F FFFF FFFF
6	Reserved	0xB000 0000 0000 0000 to 0xB000 000F FFFF FFFF
7	Uncached, accelerated	0xB800 0000 0000 0000 to 0xB800 000F FFFF FFFF

(9) xkseg (64-bit Kernel mode, physical spaces)

When the KX bit of the Status register is 1 and bits 63 and 62 of the virtual address space are 11, the virtual address space is called xkseg and selected as either 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
  References to xkseg are mapped through TLB. Whether cache can be used or not is determined by bit C of each page's TLB entry.
- One of the four 32-bit kernel compatibility spaces, as described in the next section.

(10) 64-bit kernel mode compatible spaces (ckseg0, ckseg1, cksseg, and ckseg3)

If the conditions listed below are satisfied in kernel mode, ckseg0, ckseg1, cksseg, or ckseg3 (each having 512 MB) is selected as a compatible space according to the state of the bits 30 and 29 (lower 2 bits) of the address.

- The KX bit of the Status register is 1.
- Bits 63 and 62 of the 64-bit virtual address are 11.
- Bits 61 to 31 of the virtual address are 0xFFF FFFF.

(a) ckseg0

This space is an unmapped area, compatible with the 32-bit mode kseg0 space. The K0 field of the Config register controls cacheability and coherency. (Refer to **5.5.8 Config register (16)**).

(b) ckseg1

This space is an unmapped and uncached area, compatible with the 32-bit mode kseg1 space.

(c) cksseg

This space is the ordinaty supervisor virtual space, compatible with the 32-bit mode ksseg space.
References to cksseg are mapped through TLB. Whether cache can be used or not is determined by bit C of each page's TLB entry.

(d) ckseg3

This space is the kernel virtual space, compatible with the 32-bit mode kseg3 space.
References to ckseg3 are mapped through TLB. Whether cache can be used or not is determined by bit C of each page's TLB entry.

5.5 Memory Management Registers

The CP0 registers used for managing the memory are described below. The memory management registers are listed in Table 5-9. Each register has a unique identification number that is referred to as its register number. CP0 registers not listed below are used for exception processing (refer to **CHAPTER 6 EXCEPTION PROCESSING** for details).

Table 5-9. CP0 Memory Management Registers

Register Name	Register No.
Index register	0
Random register	1
EntryLo0 register	2
EntryLo1 register	3
PageMask register	5
Wired register	6
EntryHi register	10
PRId register	15
Config register	16
LLAddr register[Note]	17
TagLo register	28
TagHi register	29

Note This register is defined to preserve compatibility with other VR Series products and has no actual operation.

With the VR5500, the hardware automatically avoids a hazard that occurs when a TLB or CP0 register is changed, except when settings related to instruction fetch are made. For the hazards related to instruction fetch, refer to **CHAPTER 19 INSTRUCTION HAZARDS**.

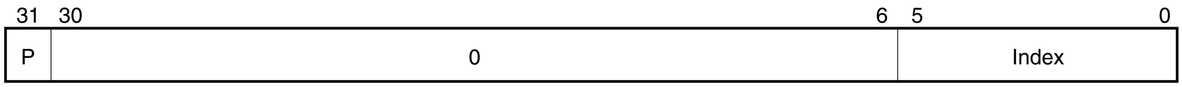
5.5.1 Index register (0)

The Index register is a 32-bit, readable/writable register containing five lower 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 field also specifies the TLB entry affected by TLB read (TLBR) or TLB write index (TLBWI) instructions. If the TLBP instruction has been successful, the index of the TLB entry that matches the contents of the EntryHi register is set to the Index field.

Since the contents of the Index register after reset are undefined, initialize this register via software.

Figure 5-11. Index Register



P: Indicates whether probing is successful or not. It is set (1) if the latest TLBP instruction fails. It is cleared (0) when the TLBP instruction is successful.

Index: Specifies an index to a TLB entry that is a target of the TLBR or TLBWI instruction.

0: Reserved. Write 0 to these bits. Zero is returned when these bits are read.

5.5.2 Random register (1)

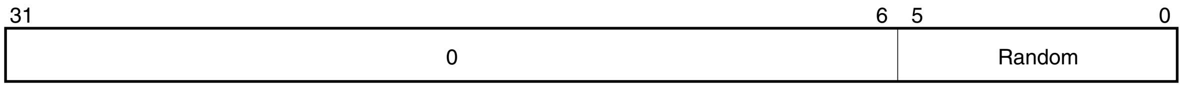
The Random register is a read-only register. The lower 6 bits are used in referencing a TLB entry. This register is decremented each time an instruction is executed. The values that can be set in the register are as follows.

- The lower bound is the content of the Wired register.
- The upper bound is 47.

The Random register specifies the entry in the TLB that is affected by the TLB write random (TLBWR) instruction. The register can be read to verify proper operation of the processor.

The Random register is set to the value of the upper boundary upon Cold Reset. This register is also set to the upper boundary when the Wired register is written.

Figure 5-12. Random Register



Random: TLB random index

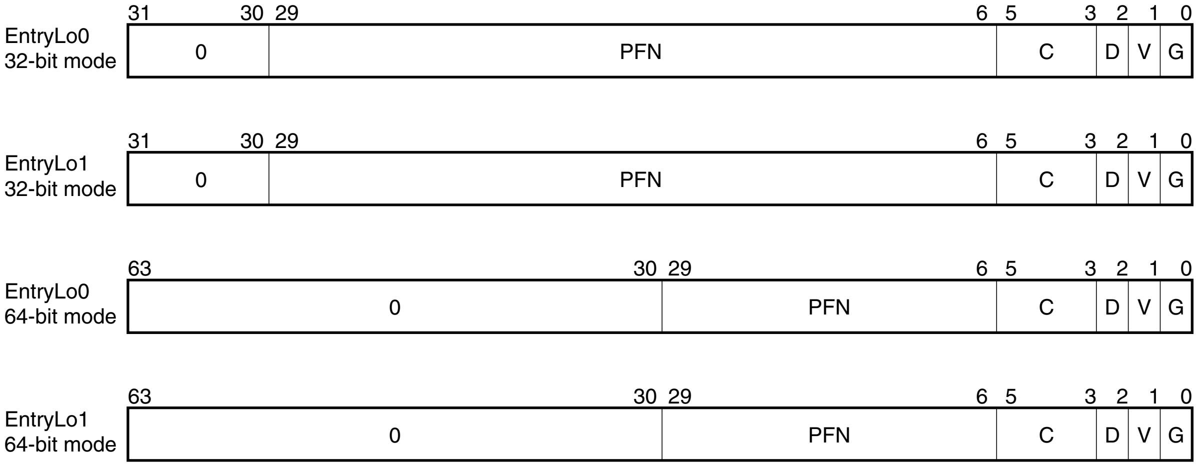
0: Reserved. Write 0 to these bits. Zero is returned when these bits are read.

5.5.3 EntryLo0 (2) and EntryLo1 (3) registers

The EntryLo register consists of two registers that have identical formats: the EntryLo0 register, used for even pages and the EntryLo1 register, used for odd pages. The EntryLo0 and EntryLo1 registers are both read-/write-accessible. They are used to access the lower bits of the on-chip TLB. When a TLB read/write operation is carried out, the EntryLo0 and EntryLo1 registers accesses the contents of the lower bits of TLB entries at even and odd addresses, respectively.

Since the contents of these registers after reset are undefined, initialize these registers via software.

Figure 5-13. EntryLo0 and EntryLo1 Registers



PFN: Page frame number; higher bits of the physical address.

C: Specifies the page attribute of the TLB entry (refer to **Table 5-10**).

D: Dirty. If this bit is set to 1, the page is 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 to 1, it indicates that the TLB entry is valid; if an entry with this bit 0 is hit, a TLB Invalid exception (TLBL or TLBS) occurs.

G: Global. If this bit is set in both the EntryLo0 and EntryLo1 registers, then the processor ignores the ASID during TLB lookup.

0: Reserved. Write 0 to these bits. Zero is returned when these bits are read.

Caution If the system interface of the VR5500 is in the 32-bit bus mode, an address error exception does not occur and physical addresses are processed with bits 35 to 32 ignored, even if the space is referenced so that bits 35 to 32 of the physical address are a value other than 0.

The C bit specifies whether the cache is used when a page is referenced. To use the cache, select an algorithm from “writeback” or “write-through, write-allocated”. Table 5-10 shows the page attributes selected by the C bit.

Table 5-10. Cache Algorithm

Value of C Bit	Cache Algorithm
0	Reserved
1	Cacheable, write-through, write-allocated
2	Uncached
3	Cacheable, writeback
4	Cacheable, write-through, write-allocated, unguarded
5	Cacheable, writeback, unguarded
6	Reserved
7	Uncached, accelerated

“Unguarded” means enabling a speculative refill operation to the external memory before a speculatively issued load/store instruction is committed if a data cache miss occurs because of the instruction. Therefore, the unguarded attribute is valid only for the data cache.

5.5.4 PageMask register (5)

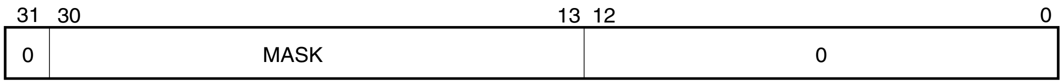
The PageMask register is a readable/writable register used for reading from or writing to the TLB; it holds a comparison mask that sets the page size for each TLB entry, as shown in Table 5-11. Page sizes can be set from 1 KB to 256 KB in five ways.

TLB read/write operation uses this register as either a source or a destination; bits 30 to 13 that are targets of comparison are masked during address translation.

Since the contents of the PageMask register after reset are undefined, initialize this register via software.

Table 5-11 lists the mask pattern for each page size. If the mask pattern is one not listed below, the TLB operates unexpectedly.

Figure 5-14. PageMask Register



MASK: Page comparison mask, which determines the virtual page size for the corresponding entry.

0: Reserved. Write 0 to these bits. Zero is returned when these bits are read.

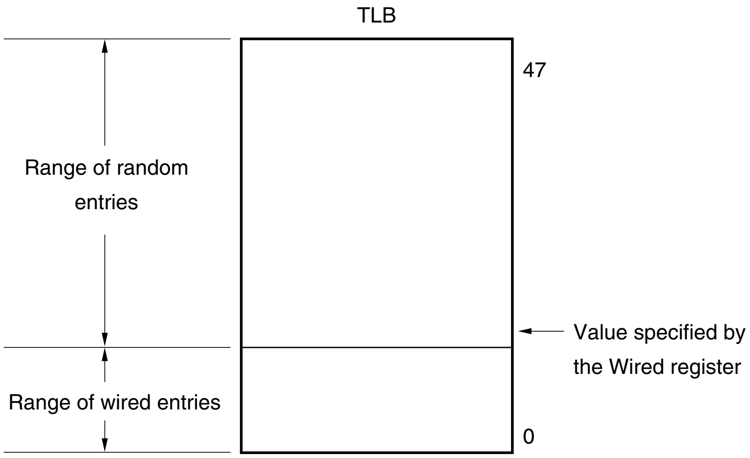
Table 5-11. Mask Values and Page Sizes

Page Size	Bit																	
	30	29	28	27	26	25	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	0	0	0	0	0	0
16 KB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
64 KB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
256 KB	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
1 MB	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
4 MB	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
16 MB	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
64 MB	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
256 MB	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1 GB	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

5.5.5 Wired register (6)

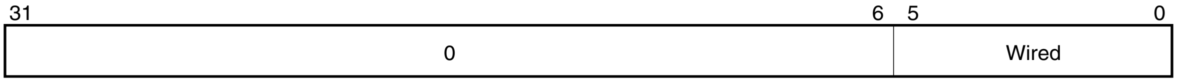
The Wired register is a readable/writable register that specifies the lower boundary of the random entry of the TLB. Wired entries cannot be overwritten by a TLBWR instruction. They can, however, be overwritten by a TLBWI instruction. Random entries can be overwritten by both instructions.

Figure 5-15. Positions Indicated by Wired Register



The Wired register is cleared to 0 after reset. Writing this register also sets the Random register to the value of its upper boundary (see **5.5.2 Random register (1)**).

Figure 5-16. Wired Register



Wired: Specifies TLB wired boundary

0: Reserved. Write 0 to these bits. Zero is returned when these bits are read.

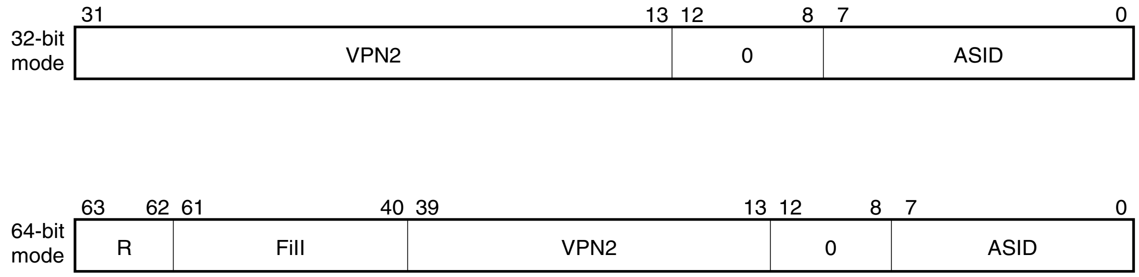
5.5.6 EntryHi register (10)

The EntryHi register is a writable register and is used to access the higher bits of the TLB. The EntryHi register holds the higher bits of a TLB entry for TLB read/write operations. If a TLB refill, TLB invalid, or TLB modified exception occurs, the EntryHi register is set with the virtual page number (VPN2) and the ASID for a virtual address where an exception occurred. See **CHAPTER 6 EXCEPTION PROCESSING** for details of TLB exceptions.

The ASID is used to read from or write to the ASID field of the TLB entry. It is also checked with the ASID of the TLB entry as the ASID of the virtual address during address translation.

The EntryHi register is accessed by the TLBP, TLBWR, TLBWI, and TLBR instructions.

Figure 5-17. EntryHi Register



VPN2: Virtual page number divided by two (mapping to two pages)

ASID: 8-bit address space ID field. This field enables the TLB to be shared by several processes. The virtual address of each process may be duplicated.

R: Space type (00 → User, 01 → Supervisor, 11 → Kernel). Matches bits 63 and 62 of the virtual address.

Fill: Reserved. Ignored on write. Zero is returned when these bits are read.

0: Reserved. Write 0 to these bits. Zero is returned when these bits are read.

5.5.7 PRId (processor revision ID) register (15)

The 32-bit, read-only processor revision ID (PRId) register contains information identifying the implementation and revision level of the CPU and CP0.

Figure 5-18. PRId Register

31 16	15 8	7 0
0	Imp	Rev

Imp: CPU processor ID number (0x55 for the VR5500)
Rev: CPU processor revision number
0: Reserved. Write 0 to these bits. Zero is returned when these bits are read.

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

The processor revision number can distinguish some revisions of the chip, however there is no guarantee that changes to the chip will necessarily be reflected in the PRId register, or that changes to the revision number necessarily reflect real chip changes. Therefore, create a program that does not depend on the processor revision number field.

5.5.8 Config register (16)

The Config register indicates/sets various statuses of processors on the VR5500.

Bits 31 to 28 and 21 to 3 are set by hardware after reset. These are read-only bits, and their status when accessed by software can be checked.

Bits 27 to 22 and 2 to 0 are readable/writable and can be manipulated by software. Since these bits are undefined after reset, initialize these bits via software.

Figure 5-19. Config Register (1/2)

31	30 28	27 24	23 22	21 20	19 18	17	16	15	14	13	12	11 9	8 6	5	4	3	2 0
0	EC	EP	EM	11	EW	1	0	BE	1	1	0	011	011	1	1	0	K0

EC: Sets the division ratio of the system clock to PClock.

000 → Divided by 2
001 → Divided by 2.5
010 → Divided by 3
011 → Divided by 3.5
100 → Divided by 4
101 → Divided by 4.5
110 → Divided by 5
111 → Divided by 5.5

EP: Sets the transfer rate of block write data. The number of data words differs depending on the bus mode of the system interface (the transfer pattern is the same).

• 32-bit bus mode

0000 → DDDDDDDD (1 word/1 cycle)
0001 → DDxDDxDDxDDx (2 words/3 cycles)
0010 → DDxxDDxxDDxxDDxx (2 words/4 cycles)
0011 → DxDxDxDxDxDxDxDx (2 words/4 cycles)
0100 → DDxxxDDxxxDDxxxDDxxx (2 words/5 cycles)
0101 → DDxxxxDDxxxxDDxxxxDDxxxx (2 words/6 cycles)
0110 → DxxDxxDxxDxxDxxDxxDxxDxx (2 words/6 cycles)
0111 → DDxxxxxxDDxxxxxxDDxxxxxxDDxxxxxx (2 words/8 cycles)
1000 → DxxxDxxxDxxxDxxxDxxxDxxxDxxxDxxx (2 words/8 cycles)
Other → Reserved

• 64-bit bus mode

0000 → DDDD (1 doubleword/1 cycle)
0001 → DDxDDx (2 doublewords/3 cycles)
0010 → DDxxDDxx (2 doublewords/4 cycles)
0011 → DxDxDxDx (2 doublewords/4 cycles)
0100 → DDxxxDDxxx (2 doublewords/5 cycles)
0101 → DDxxxxDDxxxx (2 doublewords/6 cycles)
0110 → DxxDxxDxxDxx (2 doublewords/6 cycles)
0111 → DDxxxxxxDDxxxxxx (2 doublewords/8 cycles)
1000 → DxxxDxxxDxxxDxxx (2 doublewords/8 cycles)
Other → Reserved

Figure 5-19. Config Register (2/2)

EM: Sets SysAD bus timing mode. The mode that can be selected differs depending on the bus mode of the system interface.

- In normal mode

  00 → VR4000 compatible mode
  01 → Reserved
  10 → Pipeline write mode
  11 → Write re-issuance mode

- In out-of-order return mode

  00, 10 → Pipeline mode
  01, 11 → Re-issuance mode

EW: Sets SysAD bus mode (bus width).

00 → 64-bit bus mode
01 → 32-bit bus mode
Other → Reserved

BE: Sets big-endian mode.

0 → Little endian
1 → Big endian

K0: Sets cache algorithm of kseg0.

001 → Cacheable, write-through, write-allocated
010 → Uncached
011 → Cacheable, writeback
100 → Cacheable, write-through, write-allocated, unguarded
101 → Cacheable, writeback, unguarded
111 → Uncached, accelerated
Other → Reserved

1: 1 is returned when read.

0: 0 is returned when read.

5.5.9 LLAddr (load linked address) register (17)

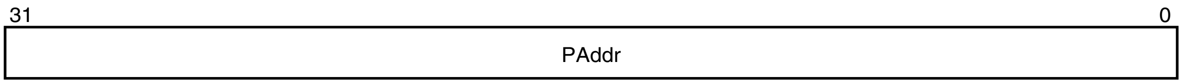
The LLAddr register is a read/write register and indicates the physical address that was read by the last LL instruction.

This register is used only for diagnostic purposes.

The PAddr field indicates the physical address PA(35:4) that is read when the LL instruction is executed.

The contents of the LLAddr register after reset are undefined.

Figure 5-20. LLAddr Register



Paddr: Bits 35 to 4 of physical address read by last LL instruction

5.5.10 TagLo (28) and TagHi (29) registers

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

The contents of these registers after reset are undefined.

Figure 5-21. TagLo and TagLo Registers

	31 8	7 6	5	4	3 1	0
TagLo	PTagLo	PState	L	R	0	P

	31 0
TagHi	0

PTagLo: Specifies physical address bits 31 to 10.

Pstate: Indicates the status of the cache.

00 → Invalid
10 → Clean
11 → Dirty
Other → Reserved

L: Sets the cache line lock.

0 → Not locked
1 → Locked

R: Specifies the way of the cache that is a candidate for replacement. The candidate for replacement is determined by the LRU algorithm.

0 → Way 0
1 → Way 1

P: Even parity bit for the cache tag

0: Reserved. Write 0 to these bits. Zero is returned when these bits are read.

The Index_Store_Tag operation of the CACHE instruction writes the value of the P bit of the TagLo register to the P bit of the cache tag as is (parity is not calculated). An operation other than the Index_Store_Tag operation that changes the contents of the cache writes the value of the parity calculated by the processor to the P bit of the cache tag.

The Index_Load_Tag operation of the CACHE instruction writes the value of the P bit of the target cache tag to the P bit of the TagLo register.

CHAPTER 6 EXCEPTION PROCESSING

This chapter describes CPU exception processing, including an explanation of the hardware that processes exceptions. For details of FPU exceptions, see **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 moves control for execution to the exception handler (fixed at a specific address as an exception processing routine implemented by software). For the exception handler, save the state of the processor, including the contents of the program counter, the current operating mode (user or supervisor), statuses, and interrupt enable. These can be restored when the exception has been processed.

When an exception occurs, the CPU loads the exception program counter (EPC) register with an address where execution can restart after the exception has been processed. The restart address in the EPC register is the address of the instruction that caused the exception or, if the instruction was being executed in a branch delay slot, the address of the branch instruction preceding the delay slot.

In addition, registers that hold address, cause, and status information during exception processing are also available. For details, refer to **6.2 Exception Processing Registers**. For details of exception processing, refer to **6.4 Details of Exceptions**.

6.2 Exception Processing Registers

This section explains 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 CP0 registers not listed in the table are used in memory management (for details, see **CHAPTER 5 MEMORY MANAGEMENT SYSTEM**).

The exception handler examines the CP0 registers during exception processing to determine the cause of the exception and the state of the CPU at the time the exception occurred.

Table 6-1. CP0 Exception Processing Registers

Register Name	Register No.
Context register	4
BadVAddr register	8
Count register	9
Compare register	11
Status register	12
Cause register	13
EPC register	14
WatchLo register	18
WatchHi register	19
XContext register	20
Performance Counter register	25
Parity Error register	26
Cache Error register	27
ErrorEPC register	30

With the VR5500, the hardware automatically avoids a hazard that occurs when a TLB or CP0 register is changed, except when settings related to instruction fetch are made. For the hazards related to instruction fetch, refer to **CHAPTER 19 INSTRUCTION HAZARDS**.

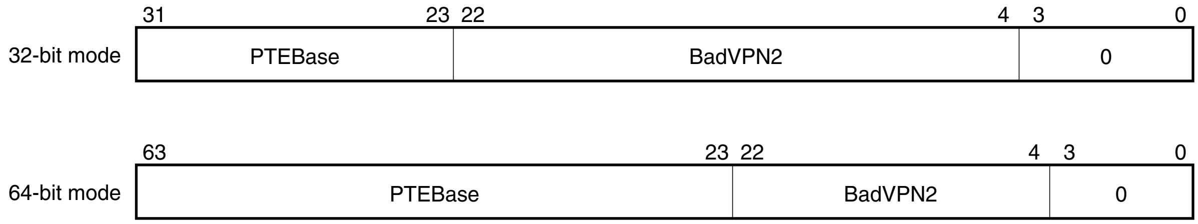
6.2.1 Context register (4)

The Context register is a read-/write-accessible register and indicates an entry in the page table entry (PTE) array in the memory. This array shows the operating system structure, and stores the virtual-to-physical address table. When a TLB miss occurs, the operating system loads the unsuccessfully translated entry from the PTE to the TLB. The Context register is used by the TLB refill exception handler for loading TLB entries.

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 TLB exception handler.

The contents of the Context register after reset are undefined.

Figure 6-1. Context Register



PTEBase: Base address of the page table entry.

BadVPN2: This field holds the value obtained by halving the virtual page number of the most recent virtual address for which translation failed.

0: Reserved. Write 0 to these bits. Zero is returned when these bits are read.

The PTEBase field is used only by the operating system as the pointer to the current PTE array on the memory.

The 19-bit BadVPN2 field contains bits 31 to 11 of the virtual address that caused the TLB miss; bit 10 is excluded because a single TLB entry maps to an even-odd page pair. For a 4 KB page size, this format can directly address the pair-table of 8-byte PTEs. When the page size is 16 KB or more, shifting or masking this value produces the correct PTE reference address.

6.2.2 BadVAddr register (8)

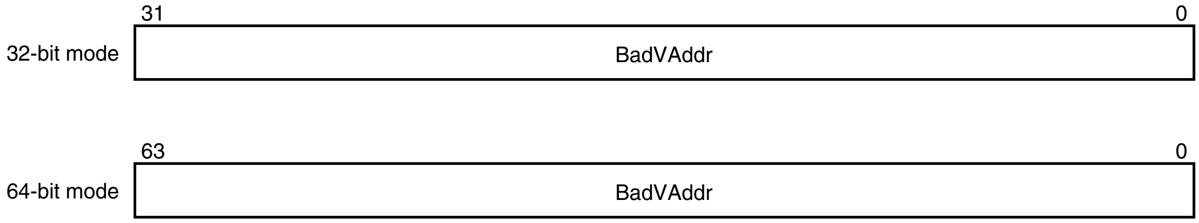
The Bad Virtual Address (BadVAddr) register is a read-only register that saves the most recent virtual address that failed to have a valid translation, or that had an addressing error. Figure 7-2 shows the format of the BadVAddr register.

If an address error occurs as a result of an instruction fetch in the 64-bit mode and a virtual address is stored in the BadVAddr register, all of bits 58 to 40 are 0 or 1.

The contents of the BadVAddr register after reset are undefined.

Caution This register saves no information after a bus error exception, because it is not an address error exception.

Figure 6-2. BadVAddr Register



BadVAddr: Most recent virtual address for which an addressing error occurred, or for which address translation failed.

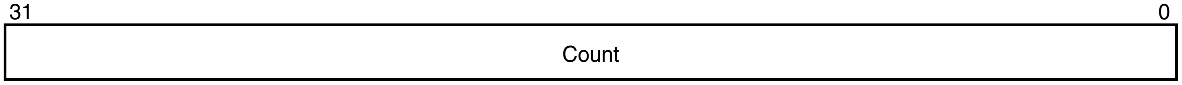
6.2.3 Count register (9)

The readable/writable Count register acts as a timer. It is incremented in synchronization with the frequency of 1/2 PClock, regardless of the instruction execution or pipeline progress status.

This register is a free-running type. When the register reaches all 1, it rolls over to 0 at the next event and continues incrementing. This register is used for self-diagnostic test, system initialization, or the establishment of inter-process synchronization.

The contents of the Count register after reset are undefined.

Figure 6-3. Count Register



Count: Most recent count value.

6.2.4 Compare register (11)

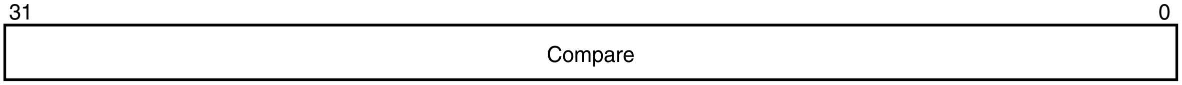
The Compare register causes a timer interrupt; it holds a value but does not change on its own. When the value of the Count register (see **6.2.3 Count register (9)**) equals the value of the Compare register, the IP7 bit in the Cause register is set. When the IP7 bit is set, this causes an interrupt as soon as the interrupt is enabled.

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

For diagnostic purposes, the Compare register is a read/write register. Normally, this register should be only used for a write.

The contents of the Compare register after reset are undefined.

Figure 6-4. Compare Register Format



Compare: Value that is compared with the count value of the Count register.

6.2.5 Status register (12)

The Status register is a readable/writable register that contains the operating mode, the interrupt enabling, and diagnostic states of the processor.

Figure 6-5. Status Register

31	30 28	27	26	25	24 16	15 8	7	6	5	4 3	2	1	0
XX	CU(2:0)	0	FR	0	DS	IM(7:0)	KX	SX	UX	KSU	ERL	EXL	IE

XX: Enables use of the MIPS IV instruction set in the user mode (0 → Disables use, 1 → Enables use).

CU: Enables use of three coprocessors (0 → Disables use, 1 → Enables use).
In the kernel mode, CP0 can be always used regardless of the CU0 bit.
CP2 is reserved for future expansion.

FR: Number of floating-point registers usable (0 → 16, 1 → 32)

DS: Self-diagnosis status field (See **Figure 6-6**.)

IM: Interrupt mask. Enables external, internal, coprocessor, and software interrupts (0 → Disables, 1 → Enables). This field consists of 8 bits and controls eight interrupts.
Each interrupt is allocated to the corresponding bit of this field as follows.
IM7: Masks timer interrupts or Int5# and external write requests.
IM(6:2): Masks ordinary external interrupts (Int(4:0)# and external write request).
IM(1:0): Masks software interrupts.

KX: Enables 64-bit addressing in kernel mode (0 → 32-bit, 1 → 64-bit). If this bit is set, an XTLB refill exception occurs if a TLB miss occurs in the kernel mode address space.
In addition, 64-bit operations are always valid in kernel mode.

SX: Enables 64-bit addressing and operation in supervisor mode (0 → 32-bit, 1 → 64-bit). If this bit is set, an XTLB refill exception occurs if a TLB miss occurs in the supervisor mode address space.

UX: Enables 64-bit addressing and operation in user mode (0 → 32-bit, 1 → 64-bit). If this bit is set, an XTLB refill exception occurs if a TLB miss occurs in the user mode address space.

KSU: Sets and indicates the operating mode (10 → User, 01 → Supervisor, 00 → Kernel).

ERL: Sets and indicates the error level (0 → Normal, 1 → Error).

EXL: Sets and indicates the exception level (0 → Normal, 1 → Exception).

IE: Sets and indicates interrupt enabling/disabling (0 → Disabled, 1 → Enabled).

0: RFU. Write 0 to this bit. Zero is returned when this bit is read.

Figure 6-6 shows the details of the Diagnostic Status (DS) field.

Figure 6-6. Status Register Diagnostic Status Field

24	23	22	21	20	19	18	17	16
DME	0	BEV	TS	SR	0	CH	CE	DE

DME: Enables setting of debug mode (0 → Disables, 1 → Enables).

BEV: Specifies base address of TLB refill exception vector and general-purpose exception vector (0 → Normal, 1 → Bootstrap).

TS: Occurrence of TLB shutdown (0 → Does not occur, 1 → Occurs)
This bit is used to avoid an adverse effect if two or more TLB entries match the same virtual address. When this bit is set (1), a TLB refill exception occurs.
TLB shutdown also occurs if the TLB entry that matches a virtual address is invalidated (by clearing the V bit of the entry).

SR: Occurrence of soft reset or NMI (0 → Does not occur, 1 → Occurs)

CH: Condition bit of CP0 (0 → False, 1 → True). This bit can be read or written only by software and is not affected by hardware.

CE: When this bit is 1, the contents of the Parity Error register are used to set or change the check bit of the cache (see **6.2.4**).

DE: Enables exception occurrence in case of cache parity error (0 → Enables, 1 → Disables).

0: Reserved. Write 0 to this bit. 0 is returned if this bit is read.

The field of the Status register that sets the mode and access status is explained next.

(1) Interrupt enable

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

- IE is set to 1.
- EXL is cleared to 0.
- ERL is cleared to 0.
- The appropriate bit of the IM is set to 1.

(2) Operating modes

The following Status register bit settings are required for user, kernel, and supervisor modes.

- The processor is in the user mode when the KSU field is 10, the EXL bit is 0, and the ERL bit is 0.
- The processor is in the supervisor mode when the KSU field is 01, the EXL bit is 0, and the ERL bit is 0.
- The processor is in the kernel mode when the KSU field is 00, the EXL bit is 1, or the ERL bit is 1.

Accessing the kernel address space is enabled only in the kernel mode.
Accessing the supervisor address space is enabled in the supervisor mode and kernel mode.
Accessing the user address space is enabled in all modes.

(3) Addressing mode

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 the kernel mode is enabled when the KX bit is 1. 64-bit operations are always valid in the kernel mode.
  If a TLB miss occurs in the kernel mode address space when this bit is set, an XTLB refill exception occurs.
- 64-bit addressing and operations are enabled for the supervisor mode when the SX bit = 1.
  If a TLB miss occurs in the supervisor mode address space when this bit is set, an XTLB refill exception occurs.
- 64-bit addressing and operations are enabled for the user mode when the UX bit = 1.
  If a TLB miss occurs in the user mode address space when this bit is set, an XTLB refill exception occurs.

(4) Status at reset

At reset, the contents of the Status register are undefined except for the following bits.

- The SR bit is 0 when a cold reset is executed and is 1 when a soft reset is executed or an NMI occurs.
- ERL bit = 1 and BEV bit = 1

6.2.6 Cause register (13)

The 32-bit readable/writable Cause register holds the cause of the most recent exception. A 5-bit in the exception code field indicates one of the exception causes (see **Table 6-2**). Other bits hold the detailed information of the specific exception. All bits in the Cause register, excepting the IP1 and IP0 bits, are read-only; IP1 and IP0 are used for software interrupts.

The contents of the Cause register after reset are undefined.

Figure 6-7. Cause Register

31	30	29 28	27 16	15 8	7	6 2	1 0
BD	0	CE	0	IP(7:0)	0	ExcCode	0

BD: Indicates whether the most recent exception occurred in the branch delay slot (1 → In delay slot, 0 → Normal).

CE: Indicates the coprocessor number in which a coprocessor unusable exception occurred.
This field will remain undefined for as long as no coprocessor unusable exception occurs.

IP: Indicates whether an interrupt is pending (1 → No interrupt pending, 0 → No interrupt).
Interrupt requests are assigned to the bits as follows.

IP7: Timer interrupt request (INT5# and external write request)

IP(6:2): Normal interrupt requests (INT(4:0)# and external write request)

IP(1:0): Software interrupt requests. These bits generate a software interrupt when they are set to 1 by software.

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

0: Reserved. Write 0 to these bits. Zero is returned when these bits are read.

Eight interrupt requests are provided in the VR5500, and requests states are reflected in IP(7:0). For details of interrupt function, refer to **CHAPTER 16 INTERRUPTS**.

- **IP7**

  This bit indicates a timer interrupt request, assertion of the interrupt request pin Int5#, and the occurrence of an interrupt due to an external write request. It is set when the contents of the count register are equal to those of the compare register, when the Performance Counter overflows, when the Int#5 signal is asserted, or when data is written to an internal register by an external write request.

  Whether the timer interrupt request, Int5# signal, or interrupt request generated by the external write request is used is specified by the TIntSel signal at reset.

- **IP(6:2)**

  Bits IP(6:2) reflect the logical sum of two internal registers. One of the registers latches the status of interrupt request pins Int(4:0)# in each cycle. Data is written to the other register by the external write request of the system interface.

- **IP1, IP0**

  A software interrupt request can be set or cleared by manipulating bits IP1 and IP0.

The following table describes the exception codes.

Table 6-2. Exception Codes

ExcCode	Mnemonic	Description
0	Int	Interrupt exception
1	Mod	TLB modified exception
2	TLBL	TLB refill exception (load or instruction fetch)
3	TLBS	TLB refill exception (store)
4	AdEL	Address error exception (load or instruction fetch)
5	AdES	Address error exception (store)
6	IBE	Bus error exception (instruction fetch)
7	DBE	Bus error exception (data load or store)
8	Sys	System call exception
9	Bp	Breakpoint exception
10	RI	Reserved instruction exception
11	CpU	Coprocessor unusable exception
12	Ov	Operation overflow exception
13	Tr	Trap exception
14	–	Reserved
15	FPE	Floating-point exception
16-22	–	Reserved
23	Watch	Watch exception
24-31	–	Reserved

To indicate the cause of the floating-point exception in detail, the exception code included in the floating-point Control/Status register is used (refer to **CHAPTER 8 FLOATING-POINT EXCEPTIONS**).

6.2.7 EPC (exception program counter) register (14)

The EPC (exception program counter) register is a readable/writable register that contains the address at which processing resumes after an exception has been processed, as shown below.

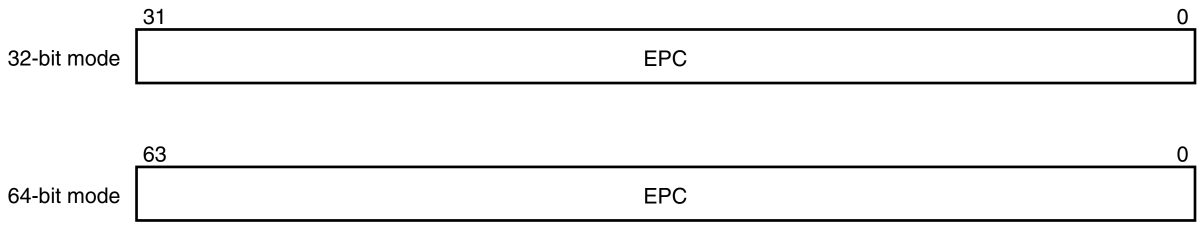
- Virtual address of the instruction that directly caused the exception.
- Virtual address of the preceding branch or jump instruction (when the instruction associated with the exception is in a branch delay slot, and the BD bit in the Cause register is set (1)).
- Virtual address of the instruction immediately after the WAIT instruction when the standby mode is released by an interrupt exception immediately after execution of the WAIT instruction

If an address error exception due to instruction fetch occurs and a virtual address is stored in the EPC register in the 64-bit mode, all of bits 58 to 40 are cleared to 0 or set to 1.

The EXL bit in the Status register is set (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.

The contents of the EPC register after reset are undefined.

Figure 6-8. EPC Register



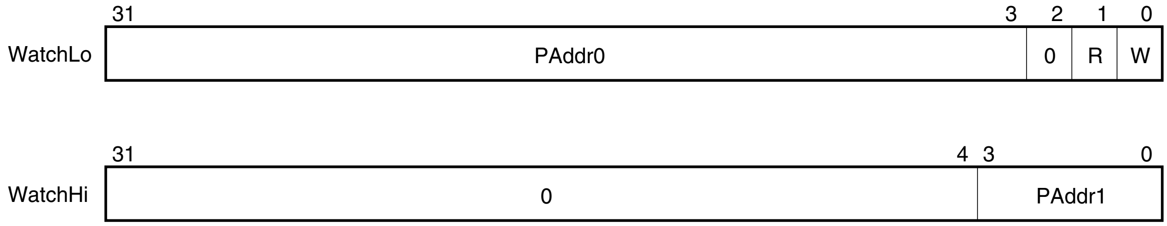
EPC: Address for a program to be restarted after exception processing.

6.2.8 WatchLo (18) and WatchHi (19) registers

The VR5500 can detect a request to reference the physical address specified by the WatchLo and WatchHi registers. This function can also be used as a debugging function to generate a watch exception at the execution of a load/store instruction.

Since the contents of these registers after reset are undefined, initialize these registers via software.

Figure 6-9. WatchLo and WatchHi Registers



Paddr1: Bits 35 to 32 of physical address.
PAddr0: Bits 31 to 3 of physical address.
R: Enables an exception occurrence when a load instruction is executed (0 → Enables, 1 → Disables).
W: Enables an exception occurrence when a store instruction is executed (0 → Enables, 1 → Disables).
0: Reserved. Write 0 to these bits. Zero is returned when these bits are read.

6.2.9 XContext register (20)

The readable/writable XContext register indicates an entry in the page table entry (PTE), an operating system data structure that stores virtual-to-physical address translations. If a TLB miss occurs, the operating system loads the untranslated data from the PTE into the TLB to handle the software error.

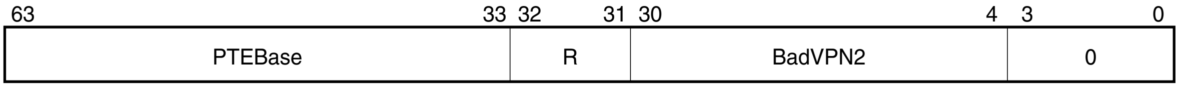
The XContext register is used by the XTLB Refill exception handler to load TLB entries in 64-bit addressing mode.

The XContext register duplicates some of the information provided in the BadVAddr register, and puts it in a form useful for the XTLB exception handler.

This register is included solely for operating system use. The operating system sets the PTEBase field in this register, as needed.

The contents of the XContext register after reset are undefined.

Figure 6-10. XContext Register



PTEBase: The PTEBase field is a base address of the page table entry.

R: Address space type (00 → user, 01 → supervisor, 11 → kernel). The setting of this field matches virtual address bits 63 and 62.

BadVPN2: Virtual address for which translation is invalid (bits 39 to 13).

0: Reserved. Write 0 to these bits. Zero is returned when these bits are read.

Only the operating system uses the PTEBase field as a pointer to the current PTE array on memory.

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

The 27-bit BadVPN2 field has bits 39 to 11 of the virtual address that caused the TLB refill; bit 12 is excluded because a single TLB entry maps to an even-odd page pair. For a 4 KB page size, this register format can be used as a pointer that references the pair-table of 8-byte PTEs. When the page size is 16 KB or more, shifting or masking this value produces the appropriate PTE reference address.

6.2.10 Performance Counter register (25)

The Performance Counter register consists of four registers: two counter registers and two control registers. Each register is a 32-bit read/write register. The VR5500 uses the Performance Counter register to count the number of events that have occurred in the processor, and can generate a timer interrupt request when the Performance Counter register overflows.

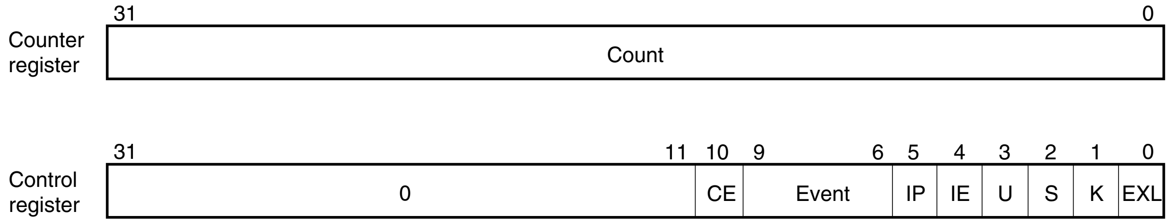
A counter register is incremented when an event specified by a control register occurs. The two counter registers correspond to the two control registers, and each counter register operates independently of each other.

The control register specifies an event to count, the mode at that time, and enables occurrence of an interrupt request.

When a counter register overflows, the IP7 bit of the Cause register is set if the control register enables occurrence of an interrupt. Even after the counter register overflows, it continues counting regardless of whether an interrupt request is reported.

When a cold reset is executed, the contents of all these registers are initialized to 0. The contents of these registers are retained after a warm reset.

Figure 6-11. Performance Counter Register



Count: Performance count value

CE: Enables performance count.

Event: Sets an event to count (refer to **Table 6-3**).

IP: Indicates occurrence of an interrupt. This bit is set (1) if the counter register overflows. Writing 0 to this bit clears the interrupt request.

IE: Enables occurrence of an interrupt. When this bit is set (1), the IP7 bit of the Cause register is set (1) if the counter register overflows.

U: When this bit is set (1), counting is performed if an event occurs in the user mode.

S: When this bit is set (1), counting is performed if an event occurs in the supervisor mode.

K: When this bit is set (1), counting is performed if an event occurs in the kernel mode and if the ERL and EXL bits are 0.

EXL: When this bit is set (1), counting is performed if an event occurs in the kernel mode and if the EXL bit is 0.

0: Reserved. Write 0 to these bits. 0 is returned if these bits are read.

Table 6-3 shows the setting of the Event field.

Table 6-3. Events to Count

Event Field	Event
0	Processor clock cycle
1	Instruction execution
2	Execution of load/prefetch/cache instruction
3	Execution of store instruction
4	Execution of branch instruction
5	Execution of floating-point instruction
6	Doubleword flush to main memory
7	TLB refill
8	Data cache miss
9	Instruction cache miss
10	Branch prediction miss
11-15	Reserved

Remark If execution of an instruction is set as an event, it is assumed that the instruction is executed when it causes an exception, and the instruction is counted as an event.

6.2.11 Parity Error register (26)

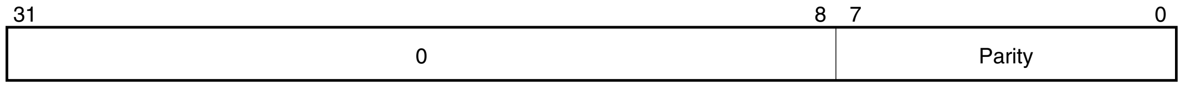
The Parity Error register reads/writes the data parity bit of the cache for initializing the cache, self-diagnosis, and error processing.

The parity is read to the Parity Error register by the CACHE instruction Index_Load_Tag.

If the CE bit of the Status register is set, the contents of the Parity Error register are written instead of the parity to the data cache by a store instruction and to the instruction cache by the Fill operation of the CACHE instruction.

The contents of the Parity Error register are undefined at reset.

Figure 6-12. Parity Error Register



Parity: Parity bit of cache data.

- For data cache
  Bit 0: Even parity for the least significant byte
  Bit 1: Even parity for the second least significant byte
  Bit 2: Even parity for the third least significant byte
  Bit 3: Even parity for the fourth least significant byte
  Bit 4: Even parity for the fourth most significant byte
  Bit 5: Even parity for the third most significant byte
  Bit 6: Even parity for the second most significant byte
  Bit 7: Even parity for the most significant byte
- For instruction cache
  Bit 0: Even parity for the lower word
  Bit 1: Even parity for the higher word

0: Reserved. Write 0 to these bits. Zero is returned when these bits are read.

6.2.12 Cache Error register (27)

The Cache Error register is a 32-bit read-only register and indicates the status of a parity error in the cache. The parity error cannot be corrected.

The Cache Error register has cache index bits that indicate the cause of an error, and status bits.

The contents of the Cache Error register after reset are undefined.

Figure 6-13. Cache Error Register

31	30	29	28	27	26	25	24	0
ER	EC	ED	ET	ES	EE	EB	0	

ER: Type of cache (0 → Instruction, 1 → Data)

EC: Cache level of error (0 → Internal, 1 → Reserved)

ED: Indicates whether a data area error has occurred (0 → No error, 1 → Error).

ET: Indicates whether a tag area error has occurred (0 → No error, 1 → Error).

ES: Set if an error occurs in the first doubleword.

EE: Set if an error occurs on the SysAD bus.

EB: Set if a data error occurs in addition to an instruction error (indicated by other bit). If this bit is set, it indicates that flushing is required for the data cache after the instruction error has been processed.

0: Reserved. Write 0 to these bits. Zero is returned when these bits are read.

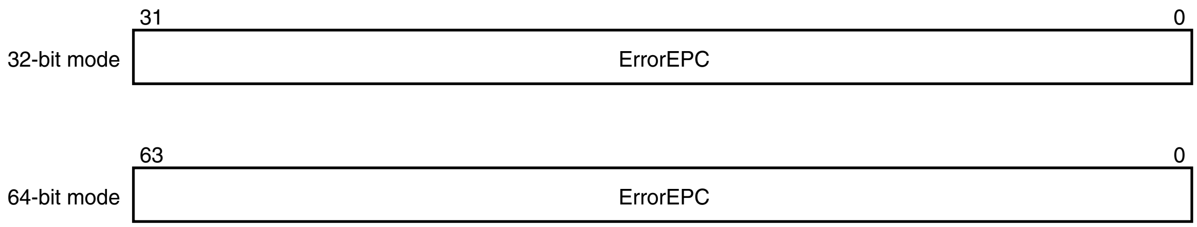
6.2.13 ErrorEPC register (30)

The ErrorEPC (error exception program counter) register is similar to the EPC register. It is used to store the program counter value at which the reset, soft reset, NMI, or cache error exception has been processed. The readable/writable ErrorEPC register holds any of the following virtual address at which instruction execution can resume after servicing an error.

- Virtual address of the instruction that directly caused the exception.
- Virtual address of the preceding branch or jump instruction (when the instruction associated with the exception is in a branch delay slot, and the BD bit in the Cause register is set (1)).
- Virtual address of the instruction immediately after the WAIT instruction when the standby mode is released by a reset, soft reset, NMI, or cache error exception immediately after execution of the WAIT instruction

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

Figure 6-14. ErrorEPC Register



ErrorEPC: Program counter that indicates the restart address after a reset, soft reset, NMI, or cache error exception.

6.3 Details of Exceptions

If an exception occurs in the processor, the EXL bit of the Status register is set to 1, and the system enters the kernel mode. Usually, the KSU field of the Status register is reset to 00 and the EXL bit is reset to 0 by an exception handler to enable occurrence of an exception in the exception handler after information has been saved. Re-set the EXL bit to 1 using the exception handler so that the saved information is not lost by any other exception while it is being restored.

When the exception processing has been completed, the setting of the KSU field before the occurrence of the exception is restored and the EXL bit is reset to 0. For details, refer to the description of the ERET instruction in **CHAPTER 17 CPU INSTRUCTION SET**.

Remark If both the EXL and ERL bits of the Status register are 0, the user mode, supervisor mode, or kernel mode is selected as the operating mode, depending on the value of the KSU field of the Status register. If either of the EXL or ERL bit is 1, the processor enters the kernel mode.

6.3.1 Exception types

Exceptions are classified as the following types, according to the internal status of the processor retained when an exception occurs.

- Reset exceptions
- Soft reset exceptions (NMI exception)
- Cache error exceptions
- Processor exceptions other than above (general exceptions)

When an exception occurs, the registers in the processor are set as follows

(1) Reset exceptions

```
T:      undefined
        Random ← TLBENTRIES – 1
        Wired ← 0
        Config ← 0 || EC || $\text{undefined}^{6}$ || 110110 || BE || 110011011110 || $\text{undefined}^{3}$
        ErrorEPC ← PC
        SR ← $\text{undefined}^{9}$ || 1 || $\text{undefined}^{19}$ || 1 || $\text{undefined}^{2}$
        PerformanceCounter ← 0
        PC ← 0xFFFF FFFF BFC0 0000
```

(2) Soft reset and NMI exceptions

```
T:      ErrorEPC ← PC
        SR ← $SR_{31:23}$ || 1 || $SR_{21}$ || 1 || $SR_{19:3}$ || 1 || $SR_{1:0}$
        PC ← 0xFFFF FFFF BFC0 0000
```

(3) Cache error exceptions

```
T:      ErrorEPC ← PC
        CacheErr ← ER || EC || ED || ET || ES || EE || EB || $0^{25}$
        SR ← $SR_{31:3}$ || 1 || $SR_{1:0}$
        if $SR_{22}$ = 1 then  /* When the BEV bit is set to 1 */
          PC ← 0xFFFF FFFF BFC0 0200 + 0x100  /* Access to the ROM area   */
        else
          PC ← 0xFFFF FFFF A000 0000 + 0x100  /* Access to the main memory area */
        endif
```

(4) General exceptions

```
T:      Cause ← BD || 0 || CE || $0^{12}$ || $Cause_{15:8}$ || ExcCode || $0^{2}$
        if $SR_{1}$ = 0 then  /* User or supervisor mode when exception processing is not in progress  */
          EPC ← PC
        endif
        SR ← $SR_{31:2}$ || 1 || $SR_{0}$
        if $SR_{22}$ = 1 then  /* When the BEV bit is set to 1 */
          PC ← 0xFFFF FFFF BFC0 0200 + vector  /* Access to the uncached area */
        else
          PC←0xFFFF FFFF 8000 0000 + vector  /* Access to the cache area */
        endif
```

6.3.2 Exception vector address

If an exception occurs, an exception vector address is set to the program counter, and processor's processing branches from the main program. Locate a program that processes the exception (exception handler) at the position of the exception vector address.

The vector address is the sum of a base address and a vector offset. The vector address differs depending on the type of exception.

64-/32-bit mode exception vectors and their offset values are shown below.

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

Exception	Vector Base Address (Virtual Address)	Vector Offset
Reset, soft reset, NMI	0xBFC0 0000 (BEV bit is automatically set to 1)	0x0000
Cache error	0xA000 0000 (BEV = 0) 0xBFC0 0200 (BEV = 1)	0x0100
TLB mismatch, EXL = 0	0x8000 0000 (BEV = 0) 0xBFC0 0200 (BEV = 1)	0x0000
XTLB mismatch, EXL = 0		0x0080
Other		0x0180

Table 6-5. 64-Bit Mode Exception Vector Addresses

Exception	Vector Base Address (Virtual Address)	Vector Offset
Reset, soft reset, NMI	0xFFFF FFFF BFC0 0000 (BEV bit is automatically set to 1)	0x0000
Cache error	0xFFFF FFFF A000 0000 (BEV = 0) 0xFFFF FFFF BFC0 0200 (BEV = 1)	0x0100
TLB mismatch, EXL = 0	0xFFFF FFFF 8000 0000 (BEV = 0) 0xFFFF FFFF BFC0 0200 (BEV = 1)	0x0000
XTLB mismatch, EXL = 0		0x0080
Other		0x0180

- **Vector of reset, soft reset, and NMI exception**

  The vector address (virtual) of each of the reset, soft reset, and NMI exceptions is in the kseg1 (uncached, non-TLB mapping) area.

- **Vector of cache error exception**

  The vector address (virtual) of the cache error exception is in the kseg1 (uncached, non-TLB mapping) area.

- **Vector of TLB refill exception (EXL = 0)**

  When the BEV bit is 0, the vector address (virtual) of this exception is in the kseg0 (cacheable, non-TLB mapping) area.

  When the BEV bit is 1, the vector address (virtual) of this exception is in kseg1 (uncached, non-TLB mapping) area.

- **Vector of general exception**

  When the BEV bit is 0, the vector address (virtual) of this exception is in the kseg0 (cacheable, non-TLB mapping) area.

  When the BEV bit is 1, the vector address (virtual) of this exception is in kseg1 (uncached, non-TLB mapping) area.

(1) Selecting TLB refill exception vector

The ISA of MIPS III or later has the following two TLB refill exception vectors.

- For referencing 32-bit address space (TLB mismatch)
- For referencing 64-bit address space (XTLB mismatch)

The TLB mismatch vector is selected in accordance with the addressing space (user, supervisor, or kernel) of the address that has generated a TLB miss, and the value of the corresponding extension addressing bits (UX, SX, or KX) of the Status register. Except when it has something to do with specifying the address space in which the address exists, the current operating mode of the processor is not important. The Context register and XContext register are completely different page table pointer registers. Each indicates a different page table and is used for refilling. No matter which TLB exception (refill exception, invalid exception, TLBL exception, or TLBS exception) occurs, the address is loaded to the BadVPN2 field of both the registers in the same way as the VR4000.

Remark Unlike the VR5500, the VR4000 selects a vector in accordance with the current operating mode of the processor (user, supervisor, or kernel) and the value of the corresponding extension addressing bit (UX, SX, or KX) of the Status register. The Context register and XContext register are provided not as completely separate registers, but share the PTEBase field. If a mismatch occurs at a specific address, a TLB refill exception or XTLB refill exception occurs, depending on the source of reference. Unless a mismatch handler decodes the address and selects a page table, only one page table can be used.

Table 6-6 shows the addresses that generate TLB mismatches and the position of the TLB refill exception vector according to the corresponding mode bit.

Table 6-6. TLB Refill Exception Vector

Space	Virtual Address Range	Area	Exception Vector
Kernel	0xFFFF FFFF E000 0000 to 0xFFFF FFFF FFFF FFFF	kseg3	TLB mismatch (KX = 0) or XTLB mismatch (KX = 1)
Supervisor	0xFFFF FFFF C000 0000 to 0xFFFF FFFF DFFF FFFF	sseg, ksseg	TLB mismatch (SX = 0) or XTLB mismatch (SX = 1)
Kernel	0xC000 0000 0000 0000 to 0xC000 0FFE FFFF FFFF	xkseg	XTLB mismatch (KX = 1)
Supervisor	0x4000 0000 0000 0000 to 0x4000 0FFF FFFF FFFF	xsseg, xksseg	XTLB mismatch (SX = 1)
User	0x0000 0000 8000 0000 to 0x0000 0FFF FFFF FFFF	xsuseg, xuseg, xkuseg	XTLB mismatch (UX = 1)
User	0x0000 0000 0000 0000 to 0x0000 0000 7FFF FFFF	useg, xuseg, suseg, xsuseg, kuseg, xkuseg	TLB mismatch (UX = 0) or XTLB mismatch (UX = 1)

6.3.3 Priority of exceptions

When more than one exception occurs for a single instruction, only the exception with the highest priority is selected for processing. Table 6-7 lists the priorities.

Table 6-7. Exception Priority Order

Priority	Exception
High	Cold reset
	Soft reset
	NMI
	Debug break (instruction fetch)
	Address error (instruction fetch)
	TLB/XTLB refill (instruction fetch)
	TLB invalid (instruction fetch)
	Cache error (instruction fetch)
	Bus error (instruction fetch)
	System call
	Breakpoint
	Coprocessor unusable
	Reserved instruction
	Trap
	Integer overflow
	Floating-point
	Debug break (data access)
	Address error (data access)
	TLB/XTLB refill (data access)
	TLB invalid (data access)
	TLB modified (data write)
	Cache error (data access)
	Bus error (data access)
	Watch
Low	Interrupt (other than NMI)

Hereafter, handling exceptions by hardware is referred to as “process”, and handling exception by software is referred to as “service”.

6.4 Details of Exceptions

6.4.1 Reset exception

(1) Cause

The reset exception occurs when the ColdReset# signal goes from active to inactive. This exception is not maskable.

(2) Processing

The special interrupt vector for reset exception is used.

- In 32-bit mode: 0xBFC0 0000 (virtual address)
- In 64-bit mode: 0xFFFF FFFF BFC0 0000 (virtual address)

The reset exception vector resides in unmapped and uncached areas, 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.
When this exception occurs, the contents of all registers are undefined except for the following registers.

- SR bit of the Status register is cleared (0).
- ERL and BEV bits of the Status register are set (1).
- The Random register is set to the value of its upper bound (47).
- The Wired register is initialized to 0.
- The Performance Counter register is initialized to 0.
- Some bits of the Config register are set in accordance with the input status of the initialization interface signal.

(3) Servicing

The reset exception is serviced by:

- Initializing all processor registers, coprocessor registers, TLB, caches, and the memory system
- Performing diagnostic tests
- Bootstrapping the operating system

6.4.2 Soft reset exception

(1) Cause

A soft reset occurs inactive while the Reset# signal goes from active to inactive when the ColdReset# signal remains.

This exception is not maskable.

(2) Processing

The special interrupt vector for reset exception (same location as reset) is used.

- In 32-bit mode: 0xBFC0 0000 (virtual address)
- In 64-bit mode: 0xFFFF FFFF BFC0 0000 (virtual address)

This vector is located within unmapped and uncached areas, so that the hardware need not initialize the TLB or the cache to process this exception. The SR bit of the Status register is set to 1 to distinguish this exception from a reset exception.

When this exception occurs, the contents of all registers are saved except for the following registers.

- The program counter value at which an exception occurs is set to the ErrorEPC register.
- ERL, SR, and BEV bits of the Status register are set (1).

During a soft reset, access to the cache or system interface may be aborted. This means that the contents of the cache and memory will be undefined if a soft reset occurs.

(3) Servicing

The soft reset exception is serviced by:

- Saving the current processor states for diagnostic tests
- Reinitializing the system in the same way as for a reset exception

6.4.3 NMI exception

(1) Cause

The NMI (non-maskable interrupt) exception occurs when the signal input to the NMI# pin becomes active. It can also be generated by writing 1 to bit 6 of the internal interrupt register from an external source via SysAD6. This exception is not maskable; it occurs regardless of the settings of the EXL, ERL, and IE bits of the Status register

(2) Processing

The special interrupt vector for NMI exception is used.

- In 32-bit mode: 0xBFC0 0000 (virtual address)
- In 64-bit mode: 0xFFFF FFFF BFC0 0000 (virtual address)

This vector is located within unmapped and uncached areas so that the hardware need not initialize an NMI exception. The SR bit of the Status register is set (1) to distinguish this exception from a reset exception.

Because the NMI exception can occur even while another exception is being processed, program execution cannot be continued after the NMI exception has been processed.

NMI occurs only at instruction boundaries. The states of the caches and memory system are saved by this exception.

When this exception occurs, the contents of all registers are saved except for the following registers.

- The program counter value at which an exception occurs is set to the ErrorEPC register.
- The ERL, SR, and BEV bits of the Status register are set (1).

(3) Servicing

The NMI exception is serviced by:

- Saving the current processor states for diagnostic tests
- Reinitializing the system in the same way as for a reset exception

6.4.4 Address error exception

(1) Cause

The address error exception occurs when an attempt is made to execute one of the following. This exception is not maskable.

- Execution of the LW or SW instruction for word data that is not located on a word boundary
- Execution of the LH or SH instruction for halfword data that is not located on a halfword boundary
- Execution of the LD or SD instruction for doubleword data that is not located on a doubleword boundary
- Referencing the kernel address space in user or supervisor mode
- Referencing the supervisor space in user mode
- Fetching an instruction that does not located on a word boundary
- Referencing the address error space
- Referencing the supervisor or kernel address space in supervisor or kernel mode using an address whose bit 31 is not sign-extended to bits 32 to 63 in 32-bit mode

(2) Processing

The general exception vector is used for this exception. The AdEL or AdES code in the Cause register is set. If this exception has been caused by an instruction reference or load operation, AdEL is set. If it has been caused by a store operation, AdES is set.

When this exception occurs, the BadVAddr register stores 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. However, if this 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 (1).

(3) Servicing

The kernel reports the UNIX™ SIGSEGV (segmentation violation) signal to the current process, and this exception is usually fatal.

(4) Restrictions

(a) With VR5500 Ver. 1.x, when the return address (contents of the EPC register) to which execution is to return from an exception handler by executing the ERET instruction is in the address error area, a value different from the contents of the program counter is stored in the EPC register if an interrupt occurs immediately after execution of the ERET instruction.
This restriction does not apply to Ver. 2.0 or later.

(b) With VR5500 Ver. 2.0 or later, if a jump/branch instruction is located two instructions before the boundary with the address error space and if a branch prediction miss (including RAS miss), ERET instruction commitment, exception (except the address error exception mentioned) does not occur (is not committed) between execution of the above jump/branch instruction and occurrence (commitment) of an address error exception due to a specific cause (refer below), the address stored in the BadVAddr register by the processing of the above address error exception is the address at the position (boundary with the address space) two instructions after the jump/branch instruction. However, the correct address is stored in the EPC register.
Therefore, do not locate a jump/branch instruction at the position two instructions before the boundary with the address space.
This restriction applies to the following causes of the address error exception.

- If an attempt is made to fetch an instruction in the kernel address space in the user or supervisor mode
- If an attempt is made to fetch an instruction in the supervisor address space in the user mode
- If an attempt is made to fetch an instruction not located at the word boundary
- If an attempt is made to reference the address error space in the kernel mode

This restriction is included in the specifications of the VR5500.

Caution With the VR5500, bits 58 to 40 of an address that is different from the actual value of the program counter are stored in the BadVAddr register and EPC register if an address error exception occurs as a result of an execution jump to the address error space in the 64-bit mode. If an address error exception occurs, therefore, do not reference the BadVAddr and EPC registers.
However, if an address error exception occurs because execution is made to jump to the address error space by the JR or JALR instruction, an incorrect address is stored in the EPC register as mentioned above, but the same value as the program counter is stored in the BadVAddr register.

6.4.5 TLB exceptions

Three types of TLB exceptions can occur.

- TLB refill exception
- TLB invalid exception
- TLB modified exception

The following three sections describe these TLB exceptions.

(1) TLB refill exception (32-bit mode)/XTLB refill exception (64-bit mode)

(a) Cause

The TLB refill exception occurs when there is no TLB entry matching the address to be referenced, or when there are multiple TLB entries to matching the address to be referenced. This exception is not maskable.

(b) Processing

There are two special exception vectors for this exception; one for 32-bit addressing mode, and one for 64-bit addressing mode. The UX, SX, and KX bits of the Status register determine which vector to use, depending on either 32-bit or 64-bit space is used for the user, supervisor or kernel mode. When the EXL bit of the Status register is set to 0, either of these two special vectors is referenced. When the EXL bit is set to 1, the general exception vector is referenced.

This exception sets the TLBL or TLBS code in the ExcCode field of the Cause register. If this exception has been caused by an instruction reference or load operation, TLBL is set. If it has been caused by a store operation, TLBS 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. However, if this 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 (1).

(c) 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 for a pair of TLB entries. The memory word is written into the TLB entry by using the EntryLo0, EntryLo1, or EntryHi register.

If the address to be referenced matches two or more entries (TLB shutdown), also clear the TS bit of the Status register to 0.

It is possible that the physical page frame and access control bits are placed in a page where the virtual address is not resident in the TLB. This condition is processed by allowing a TLB Refill exception in the TLB refill exception handler. In this case, the general exception vector is used because the EXL bit of the Status register is set (1).

(2) TLB Invalid exception

(a) Cause

The TLB invalid exception occurs when the TLB entry that matches with the virtual address to be referenced is invalid (V bit is 0). This exception is not maskable.

(b) Processing

The general exception vector is used for this exception. The TLBL or TLBS code in the ExcCode field of the Cause register is set. If this exception has been caused by an instruction reference or load operation, TLBL is set. If it has been caused by a store operation, TLBS 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 Random register normally stores 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. However, if this 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 (1).

(c) Servicing

Usually, the V bit of a TLB entry is cleared in the following cases.

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

After servicing the cause of a TLB invalid exception, the TLB entry location is identified with a TLBP (TLB Probe) instruction, and replaced by another entry with setting (1) its V bit.

(3) TLB modified exception

(a) Cause

The TLB modified exception occurs when the TLB entry that matches with the virtual address referenced by the store instruction is valid (V bit is 1) but is not writable (D bit is 0). This exception is not maskable.

(b) Processing

The general exception vector is used for this exception, and the Mod code in the ExcCode field of the Cause register 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 contents of the EntryLo register are undefined.

The EPC register contains the address of the instruction that caused the exception. However, if this 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 (1).

(c) 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 word data containing the physical page frame and access control bits (with setting (1) the D bit) is loaded to the EntryLo register, and the contents of the EntryHi and EntryLo registers are written into the TLB.

6.4.6 Cache error exception

(1) Cause

If a parity error of the cache is detected, a cache error exception occurs. This exception can be masked by the DE bit of the Status register.

When an instruction or data is read from an external source, the timing of the cache error exception differs depending on the data transfer format. When a block is transferred, only an error in the first word is checked. If an error is found in the first word, therefore, the exception immediately occurs. If an error is in the other words, however, the exception occurs when the processor uses that data. During single transfer, the exception occurs as soon as an error is found in the data.

(2) Processing

The processor sets the ERL bit of the Status register to 1, saves the exception restart address of the ErrorEPC register, and transfers information to the following special vector in a space where the cache cannot be used.

- When BEV bit = 0, the vector is 0xFFFF FFFF A000 0100
- When BEV bit = 1, the vector is 0xFFFF FFFF BFC0 0300

(3) Servicing

All errors must be logged. To correct a parity error, the system makes the cache block invalid by using the CACHE instruction, overwrites old data via a cache miss, and resumes execution by using the ERET instruction. Any other data is uncorrectable and may be fatal to the current process.

Caution **Because the data cache of the VR5500 has a non-blocking structure, a cache error exception occurs asynchronously. Even if a cache miss occurs, the subsequent instructions can be executed as long as they are not dependent upon the line where the miss occurred. Therefore, the value of the program counter when the cache error exception occurs is not always the address of the instruction that has caused the exception. Consequently, resuming execution from the instruction responsible for the exception is not guaranteed even if the system restores from the exception by using the ERET instruction.**

6.4.7 Bus error exception

(1) 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.

When an instruction or data is read from an external source, the timing of the bus error exception differs depending on the data transfer format. When a block is transferred, only an error in the first word is checked. If an error is found in the first word, therefore, the exception immediately occurs. If an error is in the other words, however, the exception occurs when the processor uses that data. During single transfer, the exception occurs as soon as an error is found in the data.

(2) Processing

The general 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), IBE is set. If it is a data reference (load/store instruction), DBE is set.

The EPC register contains the address of the instruction that caused the exception. However, if this 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 (1).

(3) Servicing

The physical address at which the fault occurred can be computed from information available in the system control coprocessor (CP0) register.

- If the IBE code in the Cause register is set (indicating an instruction fetch), the virtual address is stored in the EPC register. (4 is added to the contents of the EPC register if the BD bit of the Cause register is set to 1.)
- If the DBE code is set (indicating a load or store), the virtual address (address of the preceding branch instruction if the BD bit of the Cause register is set to 1) of the instruction that caused the exception is stored in the EPC register. (4 is added to the contents of the EPC register if the BD bit of the Cause register is set to 1.)

The virtual address of the load and store instruction 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.

At the time of this exception, the kernel reports the UNIX SIGBUS (bus error) signal to the current process, but the exception is usually fatal.

Caution Because the data cache of the VR5500 has a non-blocking structure, a bus error exception occurs asynchronously. Even if a cache miss occurs, the subsequent instructions can be executed as long as they are not dependent upon the line where the miss occurred. Therefore, the value of the program counter when the bus error exception occurs is not always the address of the instruction that has caused the exception. Consequently, resuming execution from the instruction responsible for the exception is not guaranteed even if the system restores from the exception by using the ERET instruction.

6.4.8 System call exception

(1) Cause

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

(2) Processing

The general exception vector is used for this exception, and the Sys code in the ExcCode field of the Cause register is set.

The EPC register contains the address of the SYSCALL instruction. However, if this 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 (1).

(3) Servicing

When this exception occurs, control is moved 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 before returning.

If a SYSCALL instruction is in a branch delay slot, decoding of the jump or branch instruction for identifying the branch destination is required to resume execution.

6.4.9 Breakpoint exception

(1) Cause

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

(2) Processing

The general exception vector is used for this exception, and the Bp code in the ExcCode field of the Cause register is set.

The EPC register contains the address of the BREAK instruction. However, if this 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 (1).

(3) Servicing

When the Breakpoint exception occurs, control is moved to the applicable system routine. Additional distinctions can be made by analyzing the unused bits of the BREAK instruction (bits 25 to 6), and loading the contents of the instruction whose address the EPC register contains (the address at which 4 is added to the contents of the EPC register if the BREAK instruction is 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 before returning.

If a BREAK instruction is in a branch delay slot, decoding of the branch instruction for identifying the branch destination is required to resume execution.

6.4.10 Coprocessor unusable exception

(1) Cause

The coprocessor unusable exception occurs when an attempt is made to execute a coprocessor instruction for either of the following.

- A corresponding coprocessor unit that has not been marked usable (CU0 bit of Status register = 0)
- CP0 instructions are executed in user or supervisor mode when the use of CP0 is disabled (the CU0 bit of the Status register = 0).

This exception is not maskable.

(2) Processing

The general exception vector is used for this exception, and the CpU code in the ExcCode field of the Cause register is set. The CE bit of the Cause register indicates which of the four coprocessors was referenced.

The EPC register contains the address of the instruction that caused the exception. However, if this 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 (1).

(3) Servicing

The coprocessor unit to which an attempted reference was made is identified by the CE bit of the Cause register.

One of the following processing is performed by the handler.

(a) If the process is entitled access to the coprocessor, the coprocessor is marked usable and execution is resumed.

(b) 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.

(c) If the BD bit in the Cause register is set (1), the branch instruction must be decoded; then the coprocessor instruction can be emulated and execution resumed with the EPC register advanced passing the coprocessor instruction.

(d) If the process is not entitled access to the coprocessor, the kernel reports UNIX SIGILL/ILL_PRIVIN_FAULT (illegal instruction/privileged instruction fault) signal to the current process, and this exception is fatal.

6.4.11 Reserved instruction exception

(1) Cause

The reserved instruction exception occurs when an attempt is made to execute one of the following instructions.

- Instruction with an undefined opcode (bits 31 to 26)
- SPECIAL instruction with an undefined sub opcode (bits 5 to 0)
- REGIMM instruction with an undefined sub opcode (bits 20 to 16)
- 64-bit instructions in 32-bit user or supervisor mode

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.

(2) Processing

The general exception vector is used for this exception, and the RI code in the ExcCode field of the Cause register is set.
The EPC register contains the address of the instruction that caused the exception. However, if this 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 (1).

(3) Servicing

All currently defined MIPS ISA instructions 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.

6.4.12 Trap exception

(1) 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.

(2) Processing

The general exception vector is used for this exception, and the Tr code in the ExcCode field of the Cause register is set.
The EPC register contains the address of the instruction that caused the exception. However, if this 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 (1).

(3) Servicing

At the time of a Trap exception, the kernel reports the UNIX SIGFPE/FPE_INTOVF_TRAP (floating-point exception/integer overflow) signal to the current process, and this exception is usually fatal.

6.4.13 Integer overflow exception

(1) Cause

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

(2) Processing

The general exception vector is used for this exception, and the Ov code in the ExcCode field of the Cause register is set.

The EPC register contains the address of the instruction that caused the exception. However, if this 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 (1).

(3) Servicing

At the time of the exception, the kernel reports the UNIX SIGFPE/FPE_INTOVF_TRAP (floating-point exception/integer overflow) signal to the current process, and this exception is usually fatal for current process.

6.4.14 Floating-point operation exception

(1) Cause

The floating-point exception occurs as a result of an operation of the floating-point coprocessor. This exception cannot be masked.

(2) Processing

This vector uses an ordinary exception vector and the FPE code is set to the ExcCode field of the Cause register.

The contents of the floating-point Control/Status register indicate the cause of this exception.

(3) Servicing

This exception is cleared by clearing the corresponding bit of the floating-point Control/Status register.

If an unimplemented operation exception occurs, the kernel must emulate that instruction. If any other exception occurs, the kernel passes the exception to the user program that has caused the exception.

6.4.15 Watch exception

(1) Cause

A watch exception occurs when a load or store instruction references the physical address specified by the WatchLo and WatchHi registers. The WatchLo and WatchHi registers specify whether a load or store or both could initiate this exception.

- When the R bit of the WatchLo register is set to 1: Load instruction
- When the W bit of the WatchLo register is set to 1: Store instruction
- When both the R bit and W bit of the WatchLo register are set to 1: Load instruction or store instruction

The CACHE instruction never causes a Watch exception.

The watch exception is held pending while the EXL bit of the Status register is set (1). The watch exception can be masked by either setting (1) the EXL bit of the Status register, or clearing (0) the R and W bits of the WatchLo register.

(2) Processing

The general exception vector is used for this exception, and the WATCH code in the ExcCode field of the Cause register is set.

The EPC register contains the address of the instruction that caused the exception. However, if this 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 (1).

(3) Servicing

The watch exception is a debugging aid; typically the exception handler moves control to a debugger, allowing the user to examine the situation. To continue, mask the watch exception to execute the faulting instruction.

The watch exception must then be re-enabled. The faulting instruction can be executed either by the debugger for each instruction or by setting breakpoints.

Because the contents of the WatchLo and WatchHi registers become undefined after reset, initialize these registers via software (it is particularly important to clear (0) the R and W bits). If the registers are not initialized, a watch exception may occur.

6.4.16 Interrupt exception

(1) Cause

The interrupt exception occurs when one of the eight interrupt sources[Note] is made active.

The application of these interrupts differs depending on the system. 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 IM field of the Status register, and all of the eight interrupts can be masked by clearing the IE bit of the Status register.

Note They are 1 timer interrupt, 5 ordinary interrupts, and 2 software interrupts.

Remark The timer interrupt request signal is generated if the count register matches the compare register, or if the performance counter overflows.

A timer interrupt request, or an interrupt request resulting from asserting the Int5# pin or an external write request (SysAD5) can be selected as the interrupt source reflected on the IP7 bit of the Cause register, depending on the status of the TIntSel pin after reset.

(2) Processing

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

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

The EPC register contains the address of the instruction that caused the exception. However, if this 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 (1).

(3) Servicing

If a timer interrupt request occurs, check the contents of the performance counter to identify whether a match between the count register and compare register or an overflow of the performance counter has caused the interrupt.

If the interrupt is caused by one of the two software sources, the interrupt request is cleared by setting the corresponding Cause register bit to 0.

If the interrupt is caused by hardware, the interrupt source is cleared by deactivating the corresponding interrupt request signal.

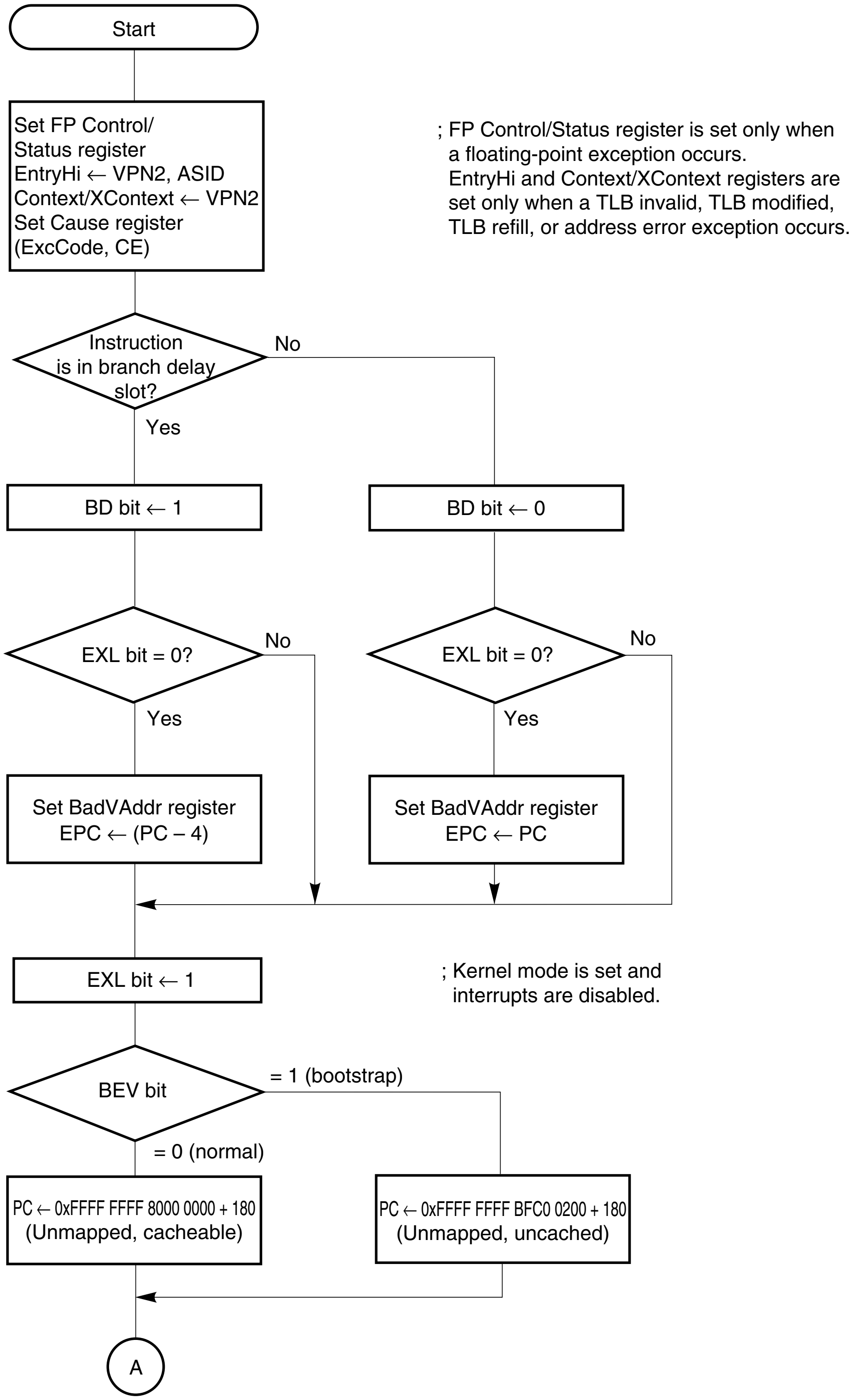
Data may not be stored in an external device until execution of the other instructions in the pipeline is completed because an internal write buffer is provided. Therefore, make sure that the data is stored correctly before the instruction that returns execution from the interrupt (ERET) is executed. If the data is not stored, the interrupt request processing may be performed again even if there is actually no pending interrupt.

6.5 Exception Processing Flowcharts

The remainder of this chapter contains flowcharts for the following exceptions and servicing for their handlers.

- General exception processing and their exception handlers
- TLB/XTLB refill exception processing and their exception handlers
- Cache error exception processing and their exception handlers
- Processing of reset, soft reset and NMI exceptions, and their exception handlers

Figure 6-15. General Exception Processing (1/2)



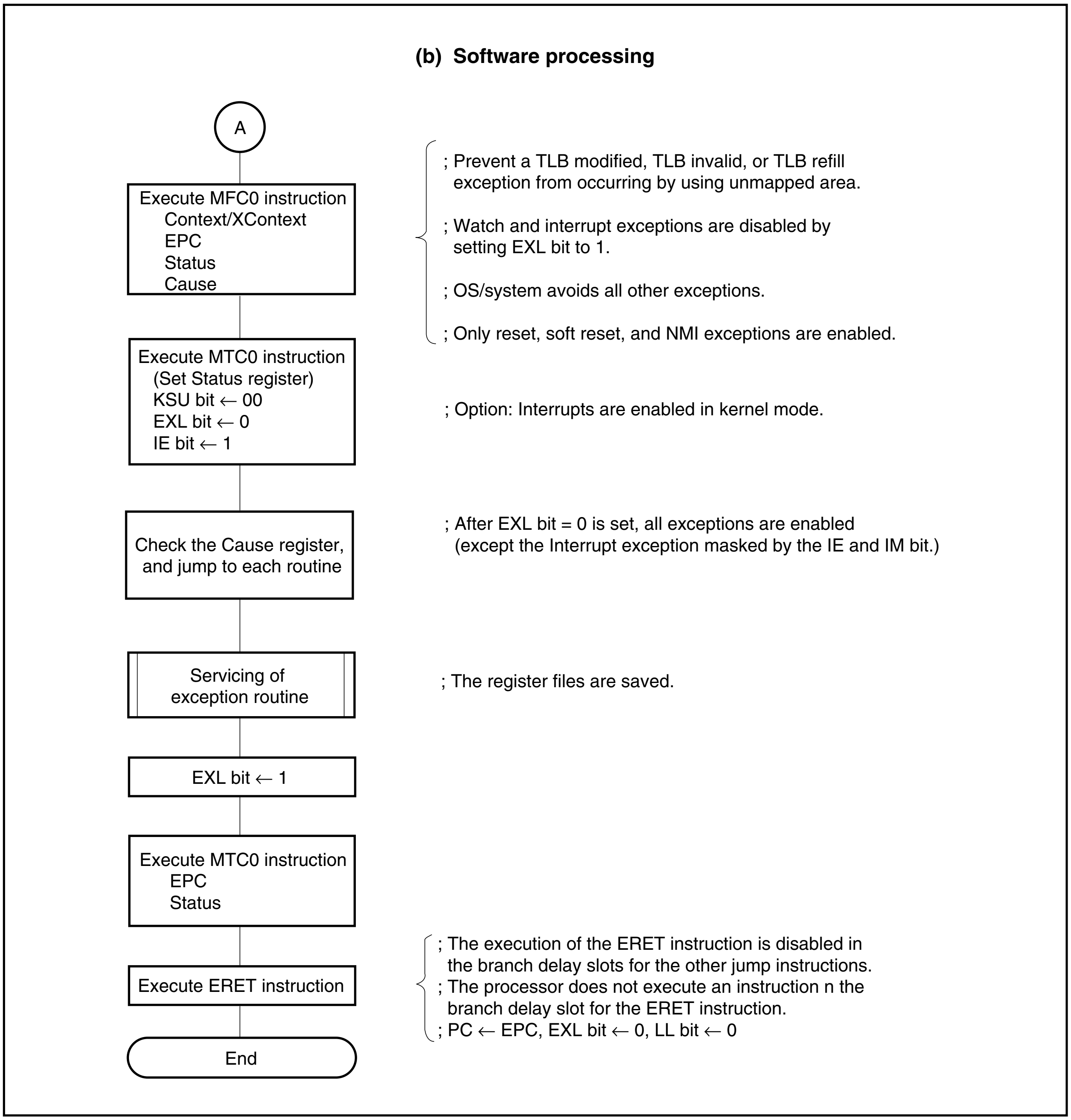
(a) Hardware processing

; FP Control/Status register is set only when a floating-point exception occurs. EntryHi and Context/XContext registers are set only when a TLB invalid, TLB modified, TLB refill, or address error exception occurs.

; Kernel mode is set and interrupts are disabled.

Remark The interrupts can be masked by setting the IE or IM bit. The watch exception can be held pending by setting the EXL bit to 1.

Figure 6-15. General Exception Processing (2/2)



(b) Software processing

A
Execute MFC0 instruction
Context/XContext
EPC
Status
Cause
; Prevent a TLB modified, TLB invalid, or TLB refill exception from occurring by using unmapped area.
; Watch and interrupt exceptions are disabled by setting EXL bit to 1.
; OS/system avoids all other exceptions.
; Only reset, soft reset, and NMI exceptions are enabled.
Execute MTC0 instruction
(Set Status register)
KSU bit ← 00
EXL bit ← 0
IE bit ← 1
; Option: Interrupts are enabled in kernel mode.
Check the Cause register, and jump to each routine
; After EXL bit = 0 is set, all exceptions are enabled (except the Interrupt exception masked by the IE and IM bit.)
Servicing of exception routine
; The register files are saved.
EXL bit ← 1
Execute MTC0 instruction
EPC
Status
Execute ERET instruction
; The execution of the ERET instruction is disabled in the branch delay slots for the other jump instructions.
; The processor does not execute an instruction n the branch delay slot for the ERET instruction.
; PC ← EPC, EXL bit ← 0, LL bit ← 0
End

Figure 6-16. TLB/XTLB Refill Exception Processing (1/2)

(a) Hardware processing

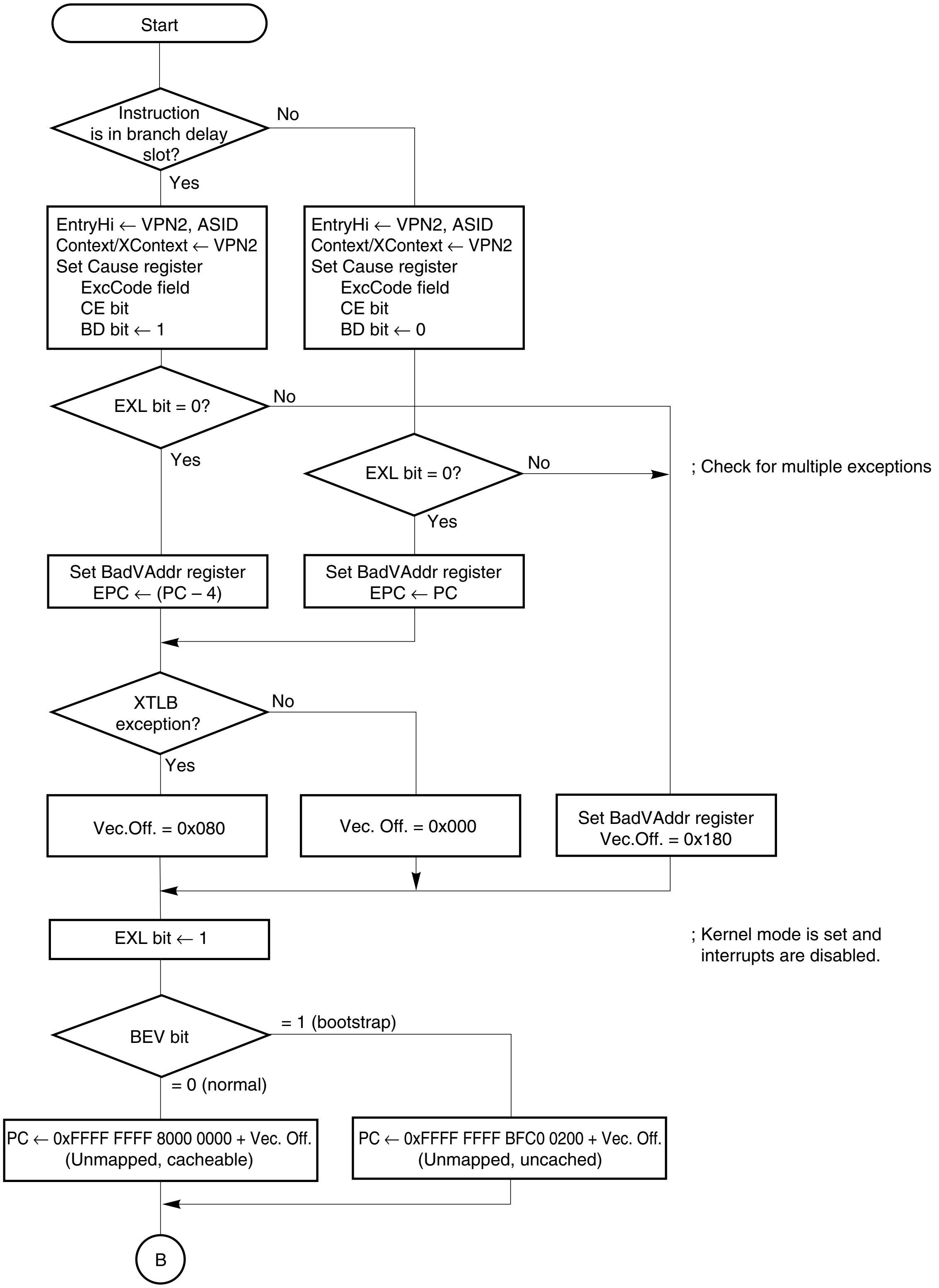
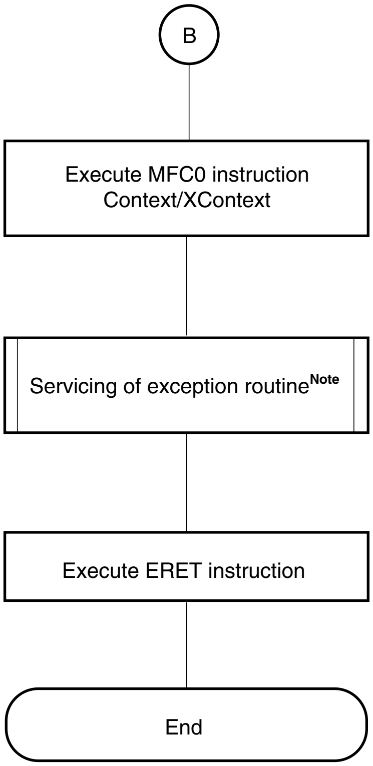


Figure 6-16. TLB/XTLB Refill Exception Processing (2/2)

(b) Software processing



; Prevent a TLB modified, TLB invalid, or TLB refill exception from occurring by using unmapped area.

; Watch and interrupt exceptions are disabled by setting EXL bit to 1.

; OS/system avoids all other exceptions.

; Only reset, soft reset, and NMI exceptions are enabled.

; The physical address for a virtual address that is loaded into the Context register is loaded into the EntryLo register and written to the TLB.

; TS bit is cleared upon TLB shutdown.

; The execution of the ERET is disabled in the branch delay slots for the other jump instructions.

; The processor does not execute an instruction n the branch delay slot for the ERET instruction.

; PC ← EPC, EXL bit ← 0, LL bit ← 0

Note A TLB refill exception may reoccur while the data/instruction addresses are in the mapping area. If an exception reoccurs, servicing will jump to the general exception vector because the EXL bit is 1. In this case, service the TLB miss in the general exception handler, return to the user program using the ERET instruction, and generate the TLB refill exception again.

Figure 6-17. Processing of Cache Error Exception

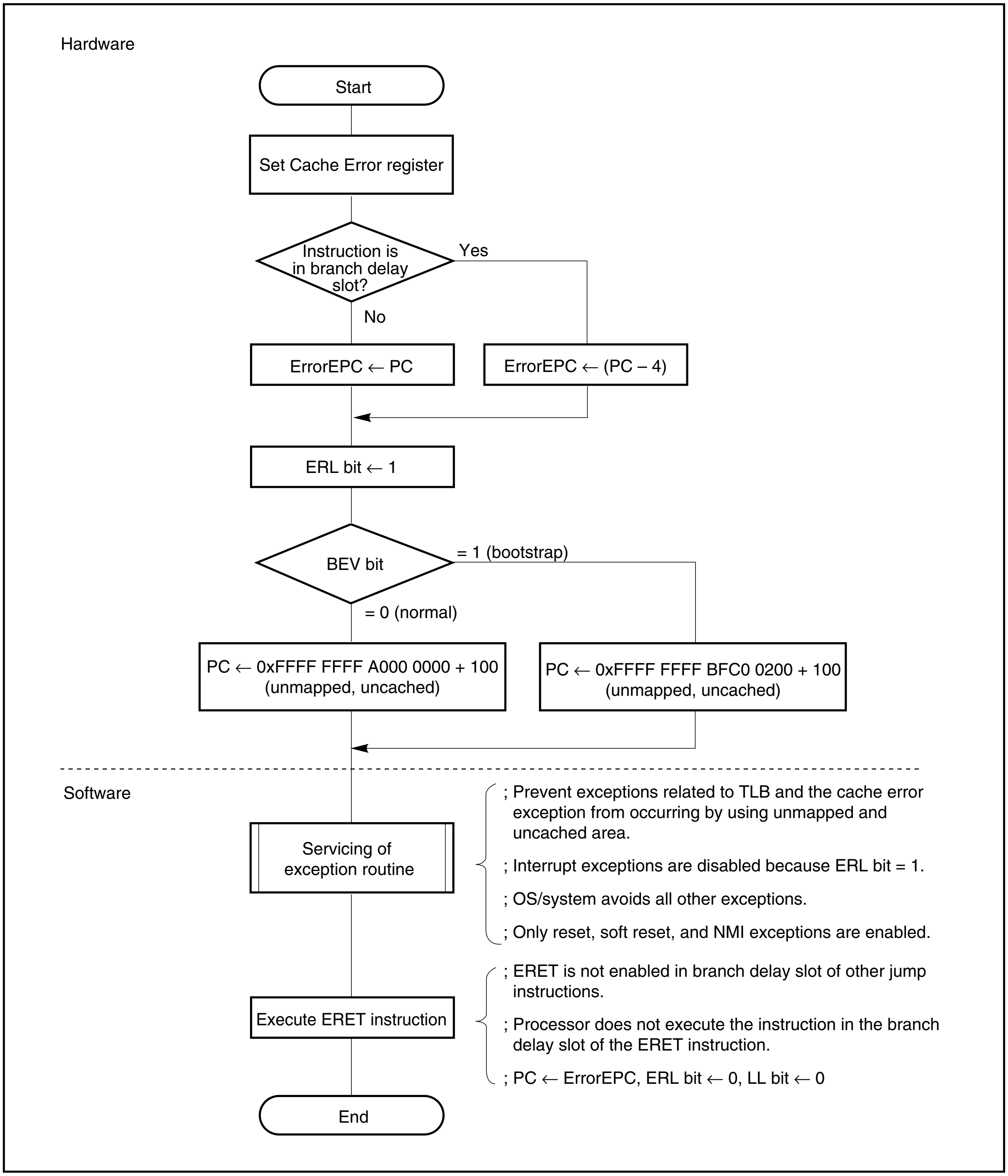
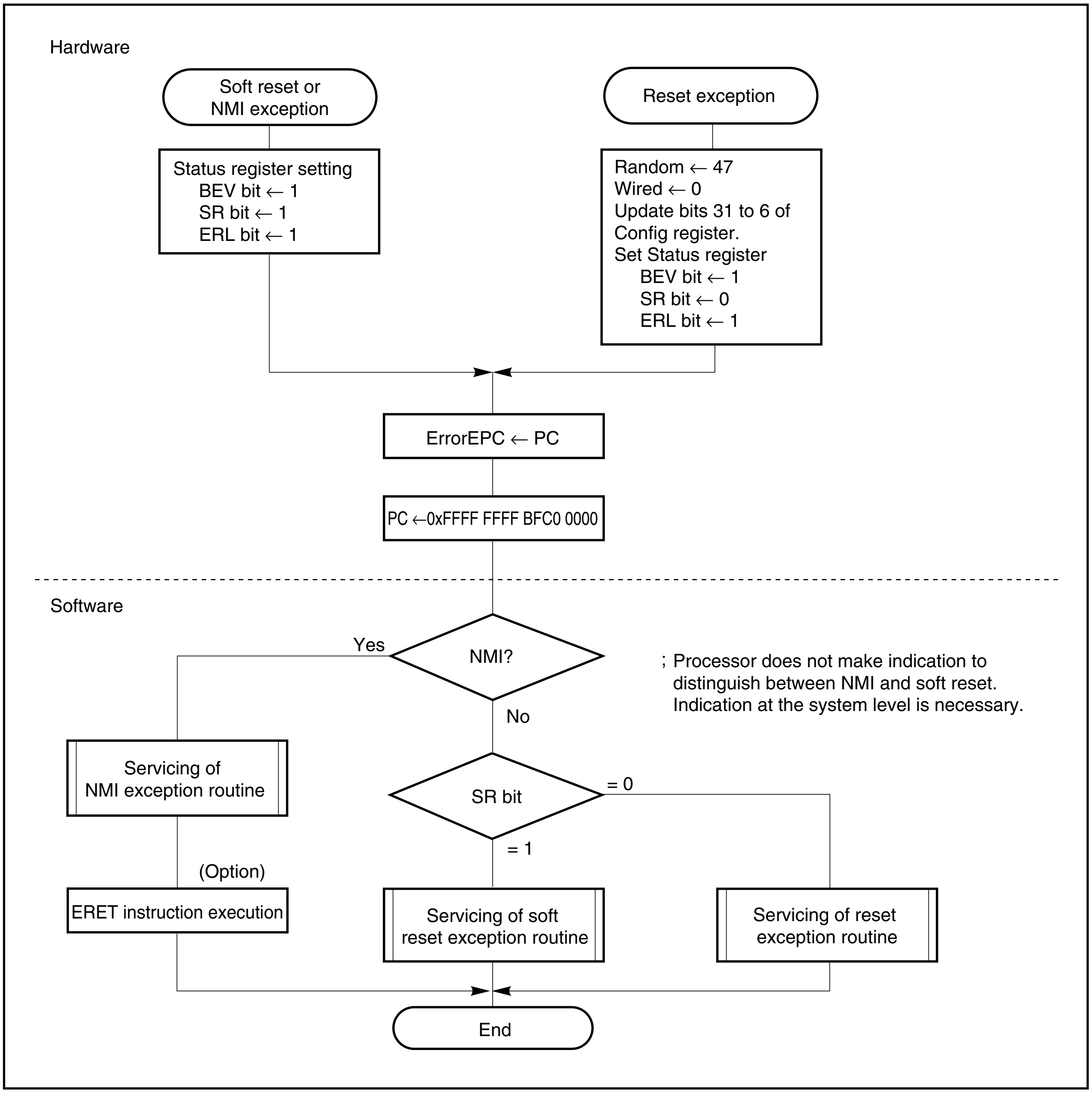

Hardware
Start
Set Cache Error register
Instruction is in branch delay slot?
Yes
No
ErrorEPC ← PC
ErrorEPC ← (PC – 4)
ERL bit ← 1
BEV bit
= 1 (bootstrap)
= 0 (normal)
PC ← 0xFFFF FFFF A000 0000 + 100 (unmapped, uncached)
PC ← 0xFFFF FFFF BFC0 0200 + 100 (unmapped, uncached)
Software
Servicing of exception routine
; Prevent exceptions related to TLB and the cache error exception from occurring by using unmapped and uncached area.
; Interrupt exceptions are disabled because ERL bit = 1.
; OS/system avoids all other exceptions.
; Only reset, soft reset, and NMI exceptions are enabled.
Execute ERET instruction
; ERET is not enabled in branch delay slot of other jump instructions.
; Processor does not execute the instruction in the branch delay slot of the ERET instruction.
; PC ← ErrorEPC, ERL bit ← 0, LL bit ← 0
End

Figure 6-18. Processing of Reset/Soft Reset/NMI Exceptions


Hardware
Soft reset or NMI exception
Reset exception
Status register setting
BEV bit ← 1
SR bit ← 1
ERL bit ← 1
Random ← 47
Wired ← 0
Update bits 31 to 6 of Config register.
Set Status register
BEV bit ← 1
SR bit ← 0
ERL bit ← 1
ErrorEPC ← PC
PC ←0xFFFF FFFF BFC0 0000
Software
Yes
NMI?
No
; Processor does not make indication to distinguish between NMI and soft reset. Indication at the system level is necessary.
Servicing of NMI exception routine
SR bit
= 0
= 1
(Option)
ERET instruction execution
Servicing of soft reset exception routine
Servicing of reset exception routine
End

CHAPTER 7 FLOATING-POINT UNIT

7.1 Overview

The floating-point unit (FPU) operates as coprocessor CP1 of the CPU and executes floating-point operation instructions. It can use both single-precision (32-bit) and double-precision (64-bit) data, and can also convert a floating-point value into a fixed-point value or vice versa.

The FPU of the VR5500 conforms to ANSI/IEEE Standard 754-1985, “IEEE2 Floating-Point Operation Standard”.

7.2 FPU Registers

The FPU has 32 general-purpose registers and 32 control registers.

Figure 7-1. Registers of FPU (1/2)

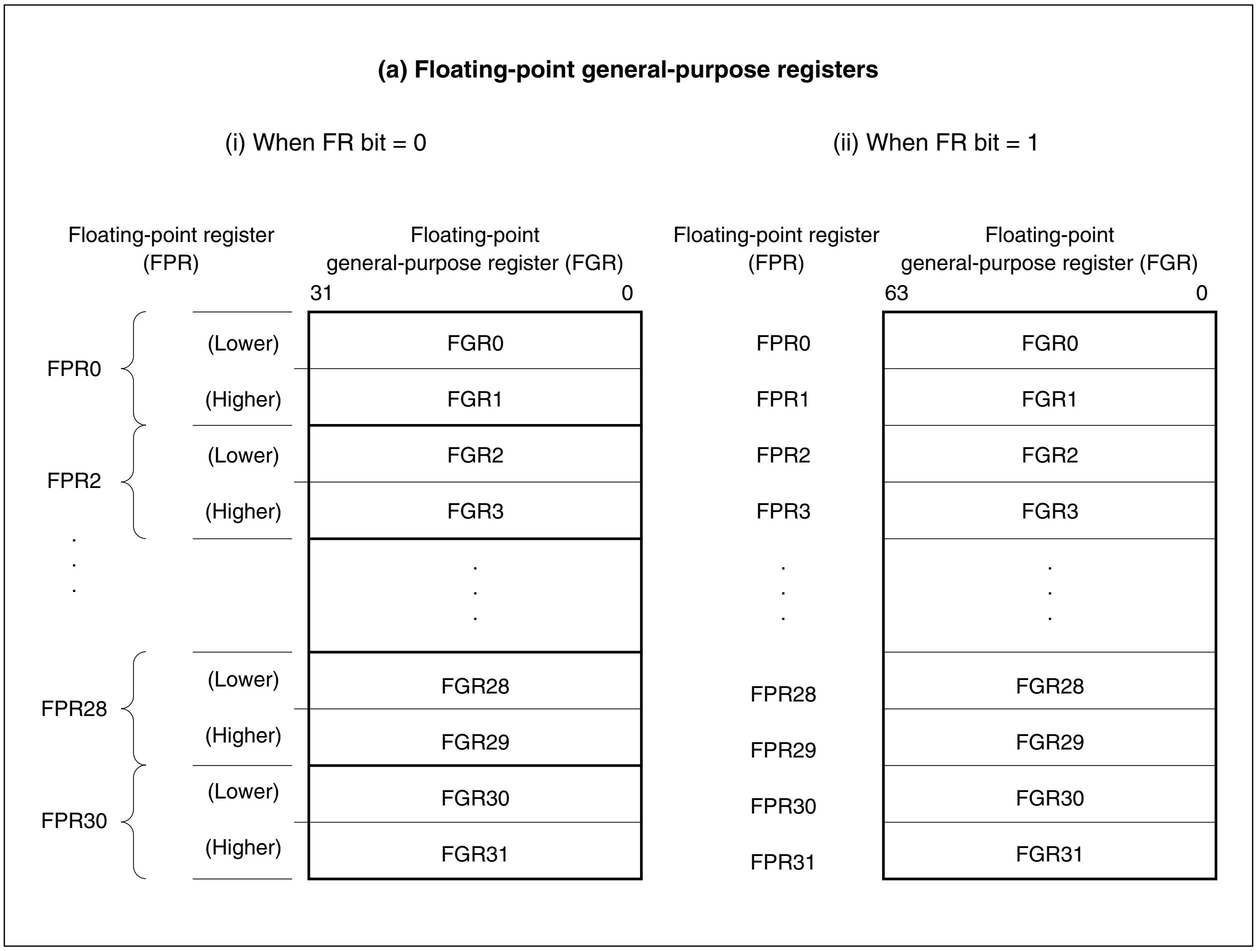
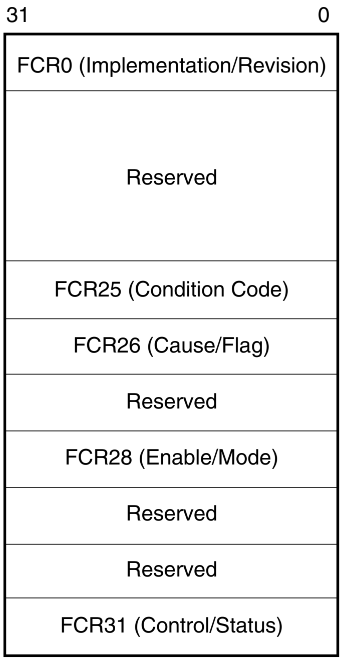

(a) Floating-point general-purpose registers
(i) When FR bit = 0
(ii) When FR bit = 1
Floating-point register (FPR)
Floating-point general-purpose register (FGR)
31
0
FPR0
FPR2
FPR28
FPR30
(Lower)
(Higher)
FGR0
FGR1
FGR2
FGR3
FGR28
FGR29
FGR30
FGR31
Floating-point register (FPR)
Floating-point general-purpose register (FGR)
63
0
FPR0
FPR1
FPR2
FPR3
FPR28
FPR29
FPR30
FPR31
FGR0
FGR1
FGR2
FGR3
FGR28
FGR29
FGR30
FGR31

Figure 7-1. Registers of FPU (2/2)

(b) Floating-point control registers



7.2.1 Floating-point general-purpose registers (FGRs)

The FPU has one set (32) of floating-point general-purpose registers (FGRs). The register length is 32 bits if the FR bit of the Status register in CP0 is 0; it is 64 bits if the FR bit is 1. The CPU accesses an FGR by using a load, store, or transfer instruction.

(1) If the FR bit of the Status register is 0, the general-purpose registers are used as sixteen 64-bit registers (FPRs) that hold single-precision or double-precision floating-point data. Each FPR corresponds to a pair of FGRs each having a serial number, as shown in Figure 7-1.

(2) If the FR bit of the Status register is 1, the general-purpose registers are used as thirty-two 64-bit registers (FPRs) that hold single-precision or double-precision floating-point data. In this case, each FPR corresponds to one FGR as shown in Figure 7-1.

7.2.2 Floating-point registers (FPRs)

If the FR bit of the Status register in CP0 is 0, sixteen floating-point registers (FPRs) can be used. If the FR bit is 1, thirty-two FPRs can be used. An FPR is a 64-bit logical register and holds a floating-point value when a floating-point operation has been executed. Physically, an FPR consists of one or two general-purpose registers (FGRs). If the FR bit of the Status register is 0, the FPR consists of two 32-bit FGRs. If the FR bit is 1, the FPR consists of one 64-bit FGR.

An FPR holds a single-precision or double-precision floating-point value. If the FR bit of the Status register is 0, only an even number is used to specify an FPR. If the FR bit is 1, all the FPR register numbers are valid. If the FR bit is 0 when double-precision floating-point operation is executed, a pair of FGRs is used as a doubleword. If FPR0 is selected for a double-precision floating-point operation, for example, two FGRs adjoining each other, FGR0 and FGR1, are used.

7.2.3 Floating-point control registers (FCRs)

The FPU has 32 control registers. The VR5500 can use the following five FCRs.

- The Control/Status register (FCR31) controls and monitors exceptions. This register also holds the result of a comparison operation and sets the rounding mode.
- The Enable/Mode register (FCR28), Cause/Flag register (FCR26), and Condition Code register (FCR25) respectively hold part of the area of FCR31, and set/hold the same contents.
- The Implementation/Revision register (FCR0) holds revision information on the FPU.

Table 7-1 shows the assignment of the FCRs.

Table 7-1. FCR

FCR No.	Usage
FCR0	Implementation/revision of coprocessor
FCR1 to FCR24	Reserved
FCR25	Condition code
FCR26	Cause, flag
FCR27	Reserved
FCR28	Exception enable, rounding mode
FCR29, FCR30	Reserved
FCR31	Condition code, rounding mode, cause, exception enable, flag

When FCR0, FCR25, FCR26, FCR28, or FCR31 is read by the CFC1 instruction, the contents of the register are transferred to the main processor after execution of all the instructions in the pipeline has been completed.

Each bit of FCR25, FCR26, FCR28, and FCR31 can be set or cleared by using the CTC1 instruction. Data is written to these registers after execution of all the instructions in the pipeline has been completed.

7.3 Floating-point Control Register

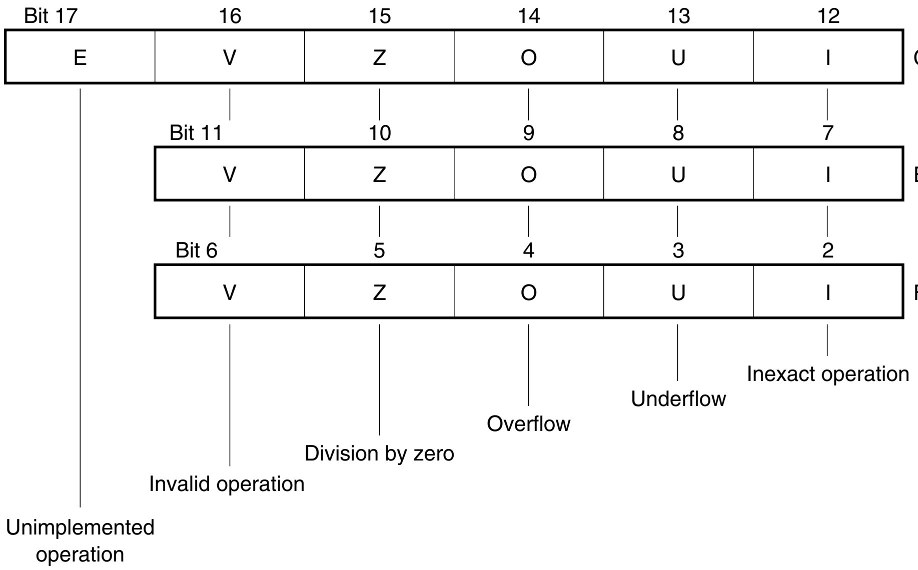
7.3.1 Control/Status register (FCR31)

The Control/Status register (FCR31) is a read/write register, and holds control data and status data. This register controls the rounding mode and enables the occurrence of a floating-point exception. It also indicates information on an exception that has occurred in the instruction executed last, and information on exceptions that have been accumulated thus far without being treated as such because they are masked. Figure 7-2 shows the configuration of FCR31. This figure shows the configuration of the cause, enable, and flag bits in FCR31.

Figure 7-2. FCR31

31 25	24	23	22 18	17 12	11 7	6 2	1 0
CC(7:1)	FS	CC0	0	Cause E V Z O U I	Enable V Z O U I	Flag V Z O U I	RM

Figure 7-3. Cause/Enable/Flag Bits of FCR31



IEEE754 defines how an exception is detected during a floating-point operation, how flags are set, and how an exception handler is called if an exception occurs. The MIPS architecture implements this specification by using the cause, enable, and flag bits of the Control/Status register. The flag bit conforms to the exception status flag of IEEE754, and the cause and enable bits conform to the exception handler of IEEE754.

Each bit of FCR31 is explained next.

(1) FS bit

The FS bit enables flushing a value that cannot be normalized (denormalized number). If this bit is set and if the enable bit of the underflow exception and illegal exception is not set, the result of a denormalized number does not cause an unimplemented operation exception to occur, but rather is flushed. Whether the denormalized number that has been flushed is 0 or the minimum normalized value depends on the rounding mode (refer to **Table 7-2**). However, the MADD.fmt, NMADD.fmt, MSUB.fmt, and NMSUB.fmt instructions cause the unimplemented operation exception to occur, regardless of the value of the FS bit.

Table 7-2. Flush Value of Denormalized Number Result

Result of Denormalized Number	Rounding Mode of Result Flushed			
	RN	RZ	RP	RM
Positive	+0	+0	+2Emin	+0
Negative	–0	–0	–0	–2Emin

(2) CC bits

Bits 31 to 25 and 23 of FCR31 are CC (condition) bits. These bits store the result of a floating-point comparison instruction. If the result is true, they are set to 1; if the result is false, they are cleared to 0. The CC bits are not affected by any instruction other than the comparison instruction and CTC1 instruction.

(3) Cause bits

Bits 17 to 12 of FCR31 are cause bits and reflect the result of the instruction executed last. The cause bits are logical extensions of the CP0 Cause register and indicate occurrence of an exception resulting from the last floating-point operation exception and its cause. If the corresponding enable bit is set, an exception occurs. If one instruction causes two or more exceptions, the corresponding bits are set.

The cause bits are rewritten by a floating-point operation (except the load, store, and transfer instructions). The E bit is set to 1 if emulation of software is necessary; otherwise it will remain 0. The other bits are cleared to 0 if an IEEE754 exception occurs, and remain set to 1 if the exception does not occur.

If a floating-point operation exception occurs, the operation result is not stored, and only the cause bits are affected.

(4) Enable bits

A floating-point operation exception occurs when both the cause bit and corresponding enable bit are set. The exception occurs as soon as a cause bit enabled for a floating-point operation has been set. The exception also occurs when the cause bit and enable bit are set by the CTC1 instruction.

No enable bit corresponding to the unimplemented operation exception is available. When the unimplemented operation exception occurs, a floating-point operation exception always occurs.

To restore from the floating-point operation exception, the cause bit that is enabled to cause the exception to occur must be cleared by software to prevent recurrence of the exception. Therefore, a cause bit that has been set cannot be seen from the program in the user mode. When using information on the cause bit via a handler in the user mode, copy the value of the Status register to another location.

Even if a cause bit is set, an exception does not occur if the corresponding enable bit is not set, and the default result defined by IEEE754 is stored. In this case, the exception caused by the floating-point operation immediately before can be identified by reading the cause bit.

(5) Flag bits

The flag bits accumulate and indicate exceptions that have occurred after reset. If an exception defined by IEEE754 occurs, the flag bit is set to 1; otherwise it will remain unchanged. The flag bit is not cleared by a floating-point operation. However, it can be set/cleared by software if a new value is written to FCR31 by using the CTC1 instruction.

If a floating-point operation exception occurs, the hardware does not set the flag bit. Therefore, set the flag bit by software before processing is transferred to the user handler.

(6) Rounding mode control bits

Bits 1 and 0 of FCR31 are RM (rounding mode control) bits. These bits define the rounding mode the FPU uses for all the floating-point instructions.

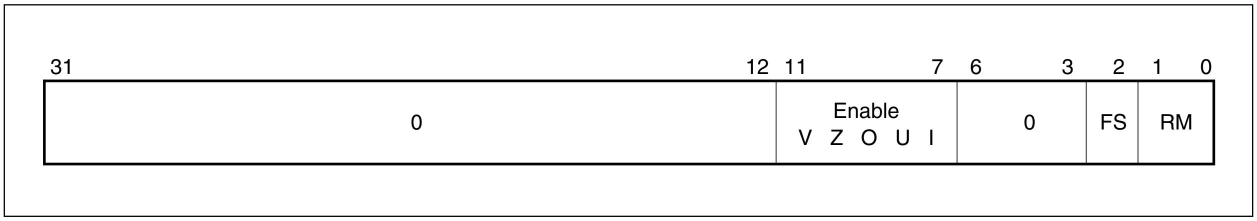
Table 7-3. Rounding Mode Control Bits

RM Bit		Mnemonic	Description
Bit 1	Bit 0		
0	0	RN	Rounds the result to the closest value that can be expressed. If the value is in between two values that can be expressed, the result is rounded toward the value whose least significant bit is 0.
0	1	RZ	Rounds the result toward 0. The result is the closest to the value that does not exceed the absolute value of the result with infinite accuracy.
1	0	RP	Rounds the result toward + ∞. The result is closest to a value greater than the accurate result with infinite accuracy.
1	1	RM	Rounds the result toward – ∞. The result is closest to a value less than the accurate result with infinite accuracy.

7.3.2 Enable/Mode register (FCR28)

The Enable/Mode register (FCR28) accesses only the enable, FS, and rounding mode control bits of FCR31. For details of each bit, refer to **7.3.1 Control/Status register (FCR31)**.

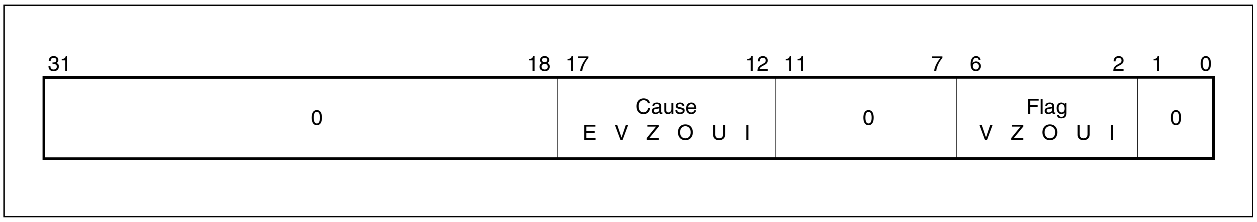
Figure 7-4. FCR28



7.3.3 Cause/Flag register (FCR26)

The Cause/Flag register (FCR26) accesses only the cause and flag bits of FCR31. For details of each bit, refer to **7.3.1 Control/Status register (FCR31)**.

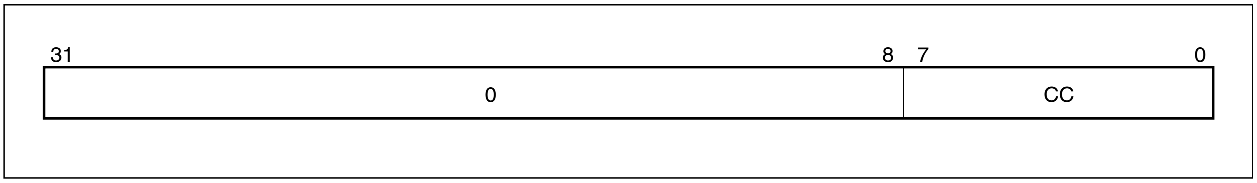
Figure 7-5. FCR26



7.3.4 Condition Code register (FCR25)

The Condition Code register (FCR25) accesses only the CC bits of FCR31. This register can treat the CC bit as eight consecutive bits. For details of the CC bits, refer to **7.3.1 Control/Status register (FCR31)**.

Figure 7-6. FCR25



7.3.5 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, status of the supported floating-point functions. This information can be used for revising the coprocessor, determining the performance level, and self-diagnosis.

Figure 7-7 shows the configuration of the Implementation/Revision register.

Figure 7-7. FCR0

31 … 20	19	18	17	16	15 … 8	7 … 0
0	3D	PS	D	S	Imp	Rev

3D: Support of three-dimensional graphics (0)
PS: Support of single-precision data pair (0)
D: Support of double-precision data pair (1)
S: Support of single-precision data (1)
Imp: Implementation identification number (0x55)
Rev: Implementation revision number
0: Reserved. Write 0 to these bits. Zero is returned when these bits are read.

Bits 19 to 16 indicate which functions are implemented in the VR5500. If a given function is not implemented, the corresponding bit is 0; if the function is implemented, the bit is 1.

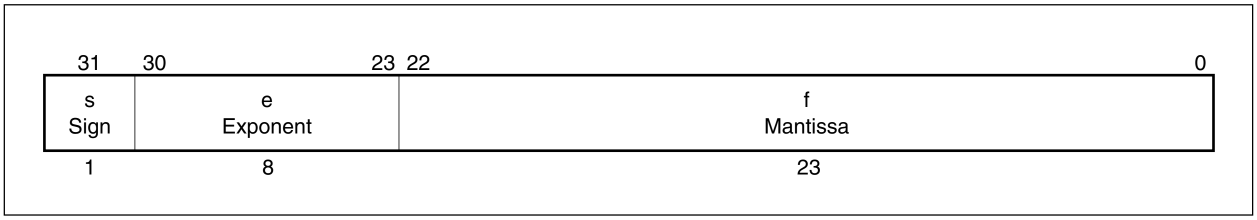
The implementation revision number is a value in the form of x.y, where y is the major revision number stored in bits 7 to 4 and x is the minor revision number stored in bits 3 to 0. The implementation revision number can be used to identify revision of the chip. However, modification of the chip is not always reflected on the revision number. Conversely, modification of the revision number does not always reflect the actual modification of the chip. Therefore, develop a program so that it does not depend upon the revision number of this register.

7.4 Data Format

7.4.1 Floating-point format

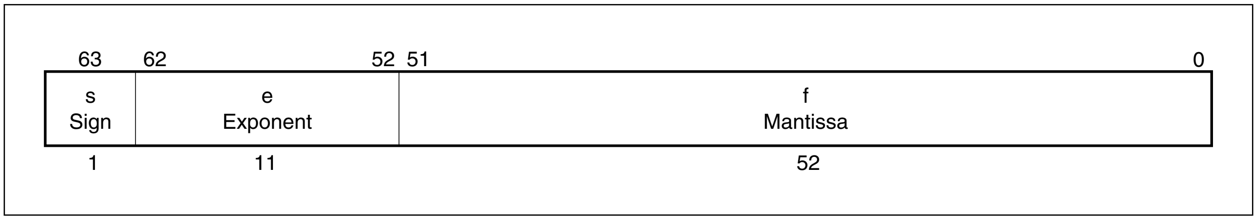
The FPU supports 32-bit (single-precision) and 64-bit (double-precision) IEEE754 floating-point operations. The single-precision floating-point format consists of a 24-bit signed mantissa (s + f) and an 8-bit exponent (e), as shown in Figure 7-8.

Figure 7-8. Single-Precision Floating-Point Format



The double-precision floating-point format consists of a 53-bit signed mantissa (s + f) and an 11-bit exponent (e), as shown in Figure 7-9.

Figure 7-9. Double-Precision Floating-Point Format



A numeric value in the floating-point format consists of the following three areas.

- Sign bit: s
- Exponent: e = E + bias value
- Mantissa: f = .b1b2…b_{P-1} (value lower than the first place below the decimal point)

The range of unbiased exponent E covers all integer values from E_{min} to E_{max}, two reserved values, $E_{min} - 1$ (±0 or denormalized number), and $E_{max} + 1$ (±∞ or NaN: Not a Number). A numeric value other than 0 is expressed in one format, depending on the single-precision and double-precision formats.

The numeric value (v) expressed in this format can be calculated by the expression shown in Table 7-4.

Table 7-4. Calculation Expression of Floating-Point Value

Type	Calculation Expression
NaN (Not a Number)	If $E = E_{max} + 1$ and $f \neq 0$, v is NaN regardless of s.
±∞ (infinite number)	If $E = E_{max} + 1$ and $f = 0$, $v = (-1)s\infty$
Normalized number	If $E_{min} \leq E \leq E_{max}$, $v = (-1)^s 2^E (1.f)$
Denormalized number	If $E = E_{min} - 1$ and $f \neq 0$, $v = (-1)^s 2^{Emin} (0.f)$
±0 (zero)	If $E = E_{min} - 1$ and $f = 0$, $v = (-1)^s 0$

- NaN (Not a Number)

  IEEE754 defines a floating-point value called NaN (Not a Number). Because it is not a numeric value, it does not have a relationship of greater than or less than.

  If v is NaN in all the floating-point formats, it may be either SignalingNaN or QuietNaN, depending on the value of the most significant bit of f. If the most significant bit of f is set, v is SignalingNaN; if the most significant bit is cleared, it is QuietNaN.

Table 7-5 shows the value of each parameter defined in the floating-point format.

Table 7-5. Floating-Point Format and Parameter Value

Parameter	Format	
	Single precision	Double precision
E_{max}	+127	+1023
E_{min}	–126	–1022
Bias value of exponent	+127	+1023
Length of exponent (number of bits)	8	11
Integer bit	Cannot be seen	Cannot be seen
Length of mantissa (number of bits)	24	53
Length of format (number of bits)	32	64

Table 7-6 shows the minimum value and maximum value that can be expressed in this floating-point format.

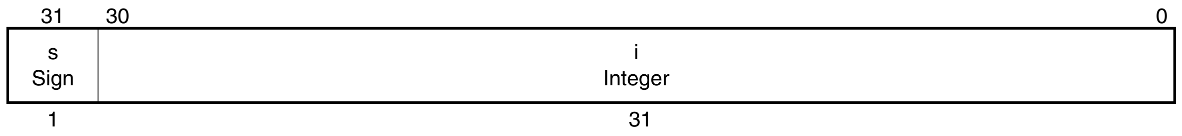
Table 7-6. Maximum and Minimum Values of Floating Point

Type	Value
Minimum value of single-precision floating point	1.40129846e – 45
Minimum value of single-precision floating point (normal)	1.17549435e – 38
Maximum value of single-precision floating point	3.40282347e + 38
Minimum value of double-precision floating point	4.9406564584124654e – 324
Minimum value of double-precision floating point (normal)	2.2250738585072014e – 308
Maximum value of double-precision floating point	1.7976931348623157e + 308

7.4.2 Fixed-point format

The value of a fixed point is held in the format of 2’s complement. Operation instructions that handle data in the unsigned fixed-point format are not provided in the floating-point instruction set. Figure 7-10 shows a 32-bit fixed-point format and Figure 7-11 shows a 64-bit fixed-point format.

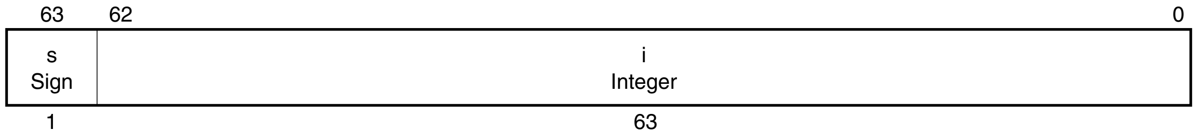
Figure 7-10. 32-Bit Fixed-Point Format



s: Sign bit

i: Integer value (2’s complement)

Figure 7-11. 64-Bit Fixed-Point Format



s: Sign bit

i: Integer value (2’s complement)

7.5 Outline of FPU Instruction Set

All the FPU instructions are 32 bits long and aligned at the word boundary. These instructions are classified as follows.

- Load/store/transfer instructions that transfer data between the general-purpose register or control register of the FPU and the CPU or memory
- Conversion instructions that convert the data format
- Arithmetic operation instructions that execute an operation on a floating-point value in an FPU register
- Comparison instructions that compares FPU registers and set the result to the CC bits of FCR31 and FCR25
- FPU branch instructions that branch execution to a specified target if the specified coprocessor condition is satisfied

fmt appended to the instruction opcode of an operation or comparison instruction indicates the data type. S indicates single-precision floating point, D indicates double-precision floating point, L indicates 64-bit fixed point, and W indicates 32-bit fixed point. For example, “ADD.D” indicates that the operand of the addition instruction is a double-precision floating-point value.

If the FR bit of the Status register in CP0 is 0, an odd-numbered register cannot be specified.

For details of each instruction, refer to **CHAPTER 18 FPU INSTRUCTION SET**.

7.5.1 Floating-point load/store/transfer instructions

(1) Load/store between FPU and memory

Loading/storing between the FPU and memory is performed by the following instructions.

- LWC1, LWXC1, SWC1, and SWXC1 instructions, which access FGR in word (32-bit) units
- LCD1, LDXC1, LUXC1, SDC1, SDXC1, and SUXC1 instructions, which access FGR in doubleword (64-bit) units

These load/store instructions are independent of the numeric value format, and format conversion is not executed. Nor does the floating-point operation exception occur.

(2) Data transfer between FPU and CPU

Data is transferred between a general-purpose register of the FPU and the CPU by the MTC1, MFC1, DMTC1, or DMFC1 instruction. Like the load/store instructions, these transfer instructions do not convert the numeric value format and the floating-point operation exception does not occur.

The CTC1 and CFC1 instructions of the CPU instruction transfer data between a control register of the FPU and the CPU.

(3) Load delay and hardware interlock

The register that is to be loaded can be used in the instruction immediately after a load instruction. In this case, however, interlocking occurs and a cycle is appended. To avoid interlocking, therefore, scheduling of the load delay slot is necessary.

With the VR5500, however, the load delay is eliminated, unless the pipeline is congested, because instructions are executed by an out-of-order mechanism. Therefore, it seems that instructions were executed without delay.

(4) Aligning data

All the load/store instructions except LUXC1 and SUXC1 reference the following aligned data.

- The access type for a word load/store instruction is always a word, and the lower 2 bits of the address must be 0.
- The access type for a doubleword load/store instruction is always a doubleword, and the lower 3 bits of the address must be 0.

(5) Byte arrangement

Regardless of the byte arrangement (endianness), an address is specified by the lowest byte address in an address area. In a big-endian system, the leftmost byte address is specified. In a little-endian system, the rightmost byte address is specified.

Table 7-7 lists the load/store/transfer instructions.

Table 7-7. Load/Store/Transfer Instructions (1/2)

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

Instruction	Format and Description COP1 base index 0 fd function
Load Word Indexed to FPU	LWXC1 fd, index (base) Adds the contents of CPU register *base* to CPU register *index* to generate an address. Loads the contents of the word specified by the address to FPU general-purpose register *fd*.
Load Doubleword Indexed to FPU	LDXC1 fd, index (base) Adds the contents of CPU register *base* to the contents of CPU register *index* to generate an address. Loads the contents of the doubleword specified by the address to FPU general-purpose registers *fd* and *fd* + 1 when FR = 0, and to FPU general-purpose register *fd* when FR = 1.
Load Doubleword Indexed Unaligned to FPU	LUXC1 fd, index (base) Adds the contents of CPU register *base* to the contents of CPU register *index* to generate an address. Loads the contents of the doubleword specified by the address to FPU general-purpose registers *fd* and *fd* + 1 when FR = 0, and to FPU general-purpose register *fd* when FR = 1.

Instruction	Format and Description COP1 base index fs 0 function
Store Word Indexed from FPU	SWXC1 fs, index (base) Adds the contents of CPU register *base* to the contents of CPU register *index* to generate an address. Stores the contents of FPU general-purpose register *fs* in the memory location specified by the address.
Store Doubleword Indexed from FPU	SDXC1 fs, index (base) Adds the contents of CPU register *base* to the contents of CPU register *index* to generate an address. Stores the contents of FPU general-purpose registers *fs* and *fs* + 1 in the memory location specified by the address when FR = 0, and FPU general-purpose register fs in the same memory location when FR = 1.
Store Doubleword Indexed Unaligned from FPU	SUXC1 fs, index (base) Adds the contents of CPU register *base* to the contents of CPU register *index* to generate an address. Stores the contents of FPU general-purpose registers *fs* and *fs* + 1 in the memory location specified by the address when FR = 0, and FPU general-purpose register *fs* in the same memory location when FR = 1.

Table 7-7. Load/Store/Transfer Instructions (2/2)

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, fs 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*.					

Instruction	Format and Description	COP1	fmt	cc	fs	fd	function
Floating-point Move Conditional on FPU True	MOVT.fmt fd, fs, cc Transfers the contents of FPU register *fs* in the specified format (*fmt*) to FPU register *fd* if the cc bit is true.						
Floating-point Move Conditional on FPU False	MOVF.fmt fd, fs, cc Transfers the contents of FPU register *fs* in the specified format (*fmt*) to FPU register *fd* if the cc bit is false.						

Instruction	Format and Description	COP1	fmt	rt	fs	fd	function
Floating-point Move Conditional on Zero	MOVZ.fmt fd, fs, rt Transfers the contents of FPU register *fs* in the specified format (*fmt*) to FPU register *fd* if CPU register *rt* is 0.						
Floating-point Move Conditional on Not Zero	MOVN.fmt fd, fs, rt Transfers the contents of FPU register *fs* in the specified format (*fmt*) to FPU register *fd* if CPU register *rt* is other than 0.						

7.5.2 Conversion instructions

The conversion instructions execute format conversion between single precision and double precision, or between fixed point and floating point.

Table 7-8 lists the conversion instructions.

Table 7-8. Conversion Instructions

Instruction	Format and Description COP1 fmt 0 fs fd function
Floating-point Convert to Single Floating-point Format	CVT.S.fmt fd, fs Converts the contents of FPU register *fs* from the specified format (*fmt*) into a single-precision floating-point format. Stores the result rounded in accordance with the setting of FCR31 and FCR28 in FPU register *fd*.
Floating-point Convert to Double Floating-point Format	CVT.D.fmt fd, fs Converts the contents of FPU register *fs* from the specified format (*fmt*) into a double-precision floating-point format. Stores the result rounded in accordance with the setting of FCR31 and FCR28 in FPU register *fd*.
Floating-point Convert to Long Fixed-point Format	CVT.L.fmt fd, fs Converts the contents of FPU register *fs* from the specified format (*fmt*) into a 64-bit fixed-point format. Stores the result rounded in accordance with the setting of FCR31 and FCR28 in FPU register *fd*.
Floating-point Convert to Single Fixed-point Format	CVT.W.fmt fd, fs Converts the contents of FPU register *fs* from the specified format (*fmt*) into a 32-bit fixed-point format. Stores the result rounded in accordance with the setting of FCR31 and FCR28 in FPU register *fd*.
Floating-point Round to Long Fixed-point Format	ROUND.L.fmt fd, fs Rounds and converts the contents of FPU register *fs* from the specified format (*fmt*) to a value closest to a 64-bit fixed-point format. Stores the result in FPU register *fd*.
Floating-point Round to Single Fixed-point Format	ROUND.W.fmt fd, fs Rounds and converts the contents of FPU register *fs* from the specified format (*fmt*) to a value closest to a 32-bit fixed-point format. Stores the result in FPU register *fd*.
Floating-point Truncate to Long Fixed-point Format	TRUNC.L.fmt fd, fs Rounds the contents of FPU register *fs* toward 0 and converts the contents from the specified format (*fmt*) into a 64-bit fixed-point format. Stores the result in FPU register *fd*.
Floating-point Truncate to Single Fixed-point Format	TRUNC.W.fmt fd, fs Rounds the contents of FPU register *fs* toward 0 and converts the contents from the specified format (*fmt*) into a 32-bit fixed-point format. Stores the result in FPU register *fd*.
Floating-point Ceiling to Long Fixed-point Format	CEIL.L.fmt fd, fs Rounds the contents of FPU register *fs* toward +∞ and converts the contents from the specified format (*fmt*) into a 64-bit fixed-point format. Stores the result in FPU register *fd*.
Floating-point Ceiling to Single Fixed-point Format	CEIL.W.fmt fd, fs Rounds the contents of FPU register *fs* toward +∞ and converts the contents from the specified format (*fmt*) into a 32-bit fixed-point format. Stores the result in FPU register *fd*.
Floating-point Floor to Long Fixed-point Format	FLOOR.L.fmt fd, fs Rounds the contents of FPU register *fs* toward –∞ and converts the contents from the specified format (*fmt*) into a 64-bit fixed-point format. Stores the result in FPU register *fd*.
Floating-point Floor to Single Fixed-point Format	FLOOR.W.fmt fd, fs Rounds the contents of FPU register *fs* toward –∞ and converts the contents from the specified format (*fmt*) into a 32-bit fixed-point format. Stores the result in FPU register *fd*.

When converting a floating-point format into a fixed-point format, make sure that the result is a value in a range of $2^{53} - 1$ to -2^{53}. If the result cannot be correctly expressed because it exceeds the range of $2^{53} - 1$ to –253 as a result of rounding the value of the source, an unimplemented operation exception occurs and the result of the operation is discarded. The instructions that cause the unimplemented operation exception under these conditions are listed below.

CEIL.L.S	CEIL.L.D
CVT.L.S	CVT.L.D
FLOOR.L.S	FLOOR.L.D
ROUND.L.S	ROUND.L.D
TRUNC.L.S	TRUNC.L.D

An unimplemented operation exception may also occur when converting a fixed-point format into a floating-point format. For details, refer to **8.3.6 Unimplemented operation exception (E)**.

7.5.3 Operation instructions

The operation instructions execute an operation on a floating-point value in a register. Table 7-9 lists the operation instructions.

Three-operand instructions execute addition, subtraction, multiplication, or division of floating-point values.

Two-operand instructions execute absolute value, transfer, square root, and arithmetic negation of a floating-point value.

Table 7-9. Operation Instructions (1/2)

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

Table 7-9. Operation Instructions (2/2)

Instruction	Format and Description COP1X fr ft fs fd function fmt
Floating-point Multiply-Add	MADD.fmt fd, fr, fs, ft Multiplies the contents of FPU registers *fs* and *ft* in the specified format (*fmt*), and adds the result to the contents of FPU register *fr* in a specified format (*fmt*). Then stores the rounded result in FPU register *fd*.
Floating-point Multiply-Subtract	MSUB.fmt fd, fr, fs, ft Multiplies the contents of FPU registers *fs* and ft in the specified format (*fmt*), and subtracts the contents of FPU register fr from the result in the specified format (*fmt*). Then stores the rounded result in FPU register *fd*.
Floating-point Negate Multiply-Add	NMADD.fmt fd, fr, fs, ft Multiplies the contents of FPU registers *fs* and ft in the specified format (*fmt*), and adds the result to the contents of FPU register *fr* in the specified format (*fmt*). Rounds the result and calculates arithmetic negation, and then stores that result in FPU register *fd*.
Floating-point Negate Multiply-Subtract	NMSUB.fmt fd, fr, fs, ft Multiplies the contents of FPU registers *fs* and *ft* in the specified format (*fmt*), and subtracts the contents of FPU register *fr* from the result in the specified format (*fmt*). Rounds the result and calculates arithmetic negation, and then stores that result in FPU register *fd*.

Instruction	Format and Description COP1 fmt 0 fs fd function
Floating-point Reciprocal	RECIP.fmt fd, fs Calculates the approximate value of the inverse number of the contents of FPU register *fs* in the specified format, and stores the result in FPU register *fd*.
Floating-point Reciprocal Square Root	RSQRT.fmt fd, fs Calculates the square root of the contents of FPU register *fs* and then the approximate value of the inverse number of that value in the specified format. Then stores the result in FPU register *fd*.

7.5.4 Comparison instruction

The comparison instruction (C.cond.fmt) converts the contents of two FPU registers (*fs* and *ft*) in the specified format (*fmt*) for comparison. The result is determined based on the comparison condition (*cond*) included in the code. Table 7-10 lists the comparison instruction, and Table 7-11 lists the conditions of the comparison instruction.

Table 7-10. Comparison Instruction

Instruction	Format and Description	COP1	fmt	ft	fs	0	function
Floating-point Compare	C.cond.fmt fs, ft Interprets the contents of FPU register *fs* and *ft* in the specified format (*fmt*), and arithmetically compares them. The result is identified by comparison and the specified condition (*cond*). The result of the comparison can be used for the FPU branch instructions of the CPU.						

Table 7-11. Conditions for Comparison Instruction

Nmemonic	Definition	Nmemonic	Definition
F	Always false	T	Always true
UN	Unordered	OR	Ordered
EQ	Equal	NEQ	Not equal
UEQ	Unordered or equal	OLG	Ordered and less than or greater than
OLT	Ordered and less than	UGE	Unordered or greater than or equal to
ULT	Ordered or less than	OGE	Ordered and greater than or equal to
OLE	Ordered and less than or equal to	UGT	Unordered or greater than
ULE	Unordered or less than or equal to	OGT	Ordered and greater than
SF	Signaling and false	ST	Signaling and true
NGLE	Not greater than, not less than, and not equal to	GLE	Greater than, less than, or equal to
SEQ	Signaling and equal to	SNE	Signaling and not equal to
NGL	Not greater than and not less than	GL	Greater than or less than
LT	Less than	NLT	Not less than
NGE	Not greater than and not equal to	GE	Greater than or equal to
LE	Less than or equal to	NLE	Not less than and not equal to
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 comparison instruction (C.cond.fmt). “Delay slot” in this table means the instruction immediately following 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 Calculates the branch target address by adding the instruction address in the delay slot and a 16-bit offset (shifts the address 2 bits to the left and sign-extends it). If the FPU condition line is true, execution branches to the target address (delay of 1 instruction).
Branch on FPU False	BC1F offset Calculates the branch target address by adding the instruction address in the delay slot and a 16-bit offset (shifts the address 2 bits to the left and sign-extends it). If the FPU condition line is false, execution branches to the target address (delay of 1 instruction).
Branch on FPU True Likely	BC1TL offset Calculates the branch target address by adding the instruction address in the delay slot and a 16-bit offset (shifts the address 2 bits to the left and sign-extends it). If the FPU condition line is true, execution branches to the target address (delay of 1 instruction). If a conditional branch does not take place, the instruction in the delay slot is invalid.
Branch on FPU False Likely	BC1FL offset Calculates the branch target address by adding the instruction address in the delay slot and a 16-bit offset (shifts the address 2 bits to the left and sign-extends it). If the FPU condition line is false, execution branches to the target address (delay of 1 instruction). If a conditional branch does not take place, the instruction in the delay slot is invalid.

7.5.6 Other instructions

Table 7-13. Prefetch Instruction

Instruction	Format and Description COP1 base index hint 0 function
Prefetch Indexed	PREFX hint, index (base) Adds the contents of CPU register base to the contents of CPU register index to generate an address. How the data specified by the address is treated is specified by the *hint* area.

Table 7-14. Conditional Transfer Instructions

Instruction	Format and Description SPECIAL rs cc rd 0 funct
Move Conditional on FPU True	MOVT rd, rs, cc Transfers the contents of CPU register *rs* to CPU register *rd* if the cc bit is true.
Move Conditional on FPU False	MOVF rd, rs, cc Transfers the contents of CPU register *rs* to CPU register *rd* if the cc bit is false.

7.6 Execution Time of FPU Instruction

Unlike the CPU, which executes almost all instructions in 1 cycle, the FPU instructions take a long time to execute.

Table 7-15 shows the minimum execution time of each floating-point instruction in the number of PCycles. This execution time is calculated on the assumption that the result of execution of each instruction is used by the instruction immediately after.

Table 7-15. Number of Execution Cycles of Floating-Point Instructions (1/2)

Instruction	Number of PCycles (When Executed Singly/Repeatedly)			
	Single	Double	Word	Long Word
ADD.fmt	4/4	4/4	–	–
SUB.fmt	4/4	4/4	–	–
MUL.fmt	5/5	6/6	–	–
MADD.fmt	9/9	10/10	–	–
MSUB.fmt	9/9	10/10	–	–
NMADD.fmt	9/9	10/10	–	–
NMSUB.fmt	9/9	10/10	–	–
DIV.fmt	30/30	59/59	–	–
SQRT.fmt	30/30	59/59	–	–
RECIP.fmt	30/30	59/59	–	–
RSQRT.fmt	60/60	118/118	–	–
ABS.fmt	2/2	2/2	–	–
NEG.fmt	2/2	2/2	–	–
ROUND.W.fmt	6/6	6/6	–	–
ROUND.L.fmt	6/6	6/6	–	–
TRUNC.W.fmt	6/6	6/6	–	–
TRUNC.L.fmt	6/6	6/6	–	–
CEIL.W.fmt	6/6	6/6	–	–
CEIL.L.fmt	6/6	6/6	–	–
FLOOR.W.fmt	6/6	6/6	–	–
FLOOR.L.fmt	6/6	6/6	–	–
CVT.D.fmt	2/2	–	6/6	6/6
CVT.S.fmt	–	4/4	6/6	6/6
CVT.W.fmt	6/6	6/6	–	–
CVT.L.fmt	6/6	6/6	–	–
C.cond.fmt	2/2	2/2	–	–

Table 7-15. Number of Execution Cycles of Floating-Point Instructions (2/2)

Instruction	Number of PCycles (When Executed Singly/Repeatedly)			
	Single	Double	Word	Long Word
BC1T	2/2 (hit), 6/6 (miss)	2/2 (hit), 6/6 (miss)	–	–
BC1F	2/2 (hit), 6/6 (miss)	2/2 (hit), 6/6 (miss)	–	–
BC1TL	2/2 (hit), 6/6 (miss)	2/2 (hit), 6/6 (miss)	–	–
BC1FL	2/2 (hit), 6/6 (miss)	2/2 (hit), 6/6 (miss)	–	–
LWC1	4/3	4/3	–	–
SWC1	NA/1	NA/1	–	–
LDC1	4/3	4/3	–	–
SDC1	NA/1	NA/1	–	–
LWXC1	4/3	4/3	–	–
SWXC1	NA/1	NA/1	–	–
LDXC1	4/3	4/3	–	–
SDXC1	NA/1	NA/1	–	–
LUXC1	4/3	4/3	–	–
SUXC1	NA/1	NA/1	–	–
MOV.fmt	2/2	2/2	–	–
MOVZ.fmt	7/7	7/7	–	–
MOVN.fmt	7/7	7/7	–	–
MOVF.fmt	7/7	7/7	–	–
MOVT.fmt	7/7	7/7	–	–
MTC1	2/2	2/2	–	–
MFC1	1/1	1/1	–	–
DMTC1	2/2	2/2	–	–
DMFC1	1/1	1/1	–	–
CTC1[Note]	10/12	10/12	–	–
CFC1[Note]	10/12	10/12	–	–

Note This instruction is executed serially. No other instructions are executed at the same time.

Remark NA: Under evaluation

CHAPTER 8 FLOATING-POINT EXCEPTIONS

This chapter explains how the FPU processes floating-point exceptions.

8.1 Types of Exceptions

A floating-point exception occurs if a floating-point operation or an operation result cannot be processed by the ordinary method.

The FPU may perform either of the following operations if an exception occurs.

- When exceptions are enabled
  The FPU sets the cause bit of the Control/Status register (FCR31) or Cause/Flag register (FCR26) and transfers processing to an exception handler routine (software processing).

- When exceptions are disabled
  The FPU stores an appropriate value (default value) in the destination register and continues execution.

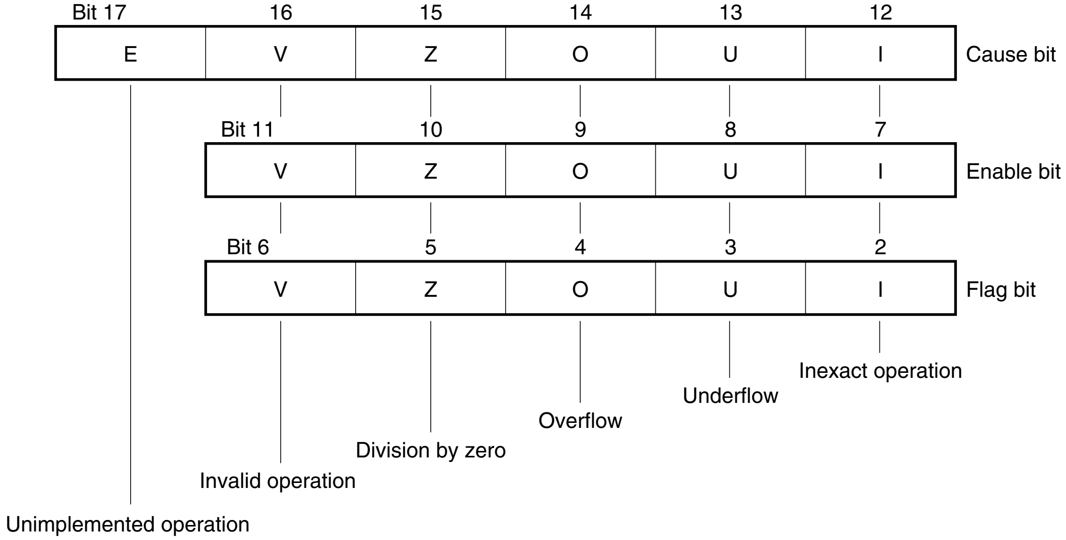
The FPU supports the following five types of IEEE754 exceptions by using the cause bit, enable bit, and flag bit (status flag).

- Inexact operation (I)
- Overflow (O)
- Underflow (U)
- Division-by-zero (Z)
- Invalid operation (V)

As the sixth exception cause, the FPU has an unimplemented operation (E) that is used if a floating-point operation cannot be executed with the standard architecture of MIPS (including when the FPU cannot correctly process an exception). This exception must be processed by software. An E bit is not provided in the enable or flag bits. If this exception occurs, unimplemented exception processing is executed (if interrupts input by the FPU to the CPU are enabled).

Figure 8-1 shows the bits of FCR31 that are used to support exceptions. The same enable bits is also provided in FCR28, and the same cause and flag bits are also provided in FCR26.

Figure 8-1. Cause/Enable/Flag Bits of FCR31



The five exceptions of IEEE754 (V, Z, O, U, and I) are enabled by setting the corresponding bit. 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 bit and flag bit corresponding to that exception are set.

8.2 Exception Processing

If a floating-point operation exception occurs, the Cause register of CP0 indicates that the cause of the exception lies in the FPU. The code of the floating-point exception (FPE) is used, and the cause bits of FCR31 and FCR26 indicate the cause of the floating-point operation exception. These bits function as an extension of the Cause register of CP0.

8.2.1 Flag

A flag bit is available for each IEEE754 exception. The flag bit is set if occurrence of the corresponding exception is disabled and if the condition of the exception is detected. The flag bit can be set/reset by writing a new value to FCR31 or FCR26 using the CTC1 instruction.

If an exception is disabled by the corresponding enable bit, the FPU performs predetermined processing. This processing gives a default value instead of the result of the floating-point operation. This default value is determined by the type of the exception. If an overflow or underflow exception occurs, the default value differs depending on the rounding mode at that time. Table 8-1 shows the default values given by each IEEE754 exception of the FPU.

Table 8-1. Default Values of IEEE754 Exceptions in FPU

Area	Description	Rounding Mode	Default Value
V	Invalid operation	–	Uses Quiet Not a Number (Q-NaN).
Z	Division-by-zero	–	Uses correctly signed ∞.
O	Overflow	RN	∞ with sign of intermediate result
		RZ	Maximum normalized number with sign of intermediate result
		RP	Negative overflow: Maximum negative normalized number Positive overflow: +∞
		RM	Positive overflow: Maximum positive normalized number Negative overflow: –∞
U	Underflow	RN	0 with sign of intermediate result
		RZ	0 with sign of intermediate result
		RP	Positive underflow: Minimum positive normalized number Negative underflow: 0
		RM	Negative underflow: Minimum negative normalized number Positive underflow: 0
I	Inexact operation	–	Uses rounded result.

The FPU internally detects nine types of statuses that may trigger an exception. When the FPU detects these abnormal statuses, an IEEE754 exception or the unimplemented operation exception (E) occurs. Table 8-2 shows the statuses that trigger exceptions, and a comparison of the contents of the corresponding cause bits of the FPU and the IEEE754 standard.

Table 8-2. FPU Internal Result and Flag Status

FPU Internal Result	IEEE754	Exception Enabled	Exception Disabled	Remark
Inexact operation	I	I	I	Result is not accurate.
Exponent overflow	O, I[Note]	O, I	O, I	Normalized exponent > E_{max}
Division-by-zero	Z	Z	Z	Zero (exponent = E_{min} – 1, mantissa = 0)
Overflow during conversion	V	E	E	Source is outside integer range
Signaling NaN (S-NaN) source	V	V	V	
Invalid operation	V	V	V	0 ÷ 0, etc.
Exponent underflow	U	E	E	Normalized exponent < E_{min}
Denormalized source	None	E	E	Exponent = E_{min} – 1 and mantissa ≠ 0
Q-NaN	None	E	E	

Note IEEE754 allows an Inexact operation exception to occur in the case of an overflow only when the overflow exception is disabled, but the VR5500 always allows an overflow exception and an inexact operation exception to occur in the case of an overflow.

8.3 Details of Exceptions

This section explains the conditions under which each exception occurs and the action taken by the FPU.

8.3.1 Inexact operation exception (I)

The FPU generates an 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 an underflow exception and an inexact operation exception are disabled and the FS bit of FCR31 and FCR28 is set

Usually, the FPU checks the operands of an instruction before executing the instruction. Based on the exponent value of the operand, the FPU judges whether an exception may occur as a result of executing this instruction. If an exception may occur, the FPU uses a stall when executing this instruction.

However, the FPU cannot predict whether executing a certain instruction results in an illegal value. If the inexact operation exception is enabled, the FPU uses a stall for executing all instructions, and thus the execution time increases by 1 cycle. This substantially affects the performance. Therefore, enable the inexact operation instruction only when it is necessary.

(1) If exception is enabled

The contents of the destination register are not changed, the contents of the source register are saved, and the inexact operation exception occurs.

(2) If exception is not enabled

If no other exception occurs, the rounded result or the result that underflows/overflows is stored in the destination register.

8.3.2 Invalid operation exception (V)

An invalid operation exception occurs if one of or both the operands are invalid. If the exception is not enabled, the result is Not a Number (Q-NaN). The invalid operations include the following operations.

- Addition/subtraction: Addition/subtraction between infinities (+∞) + (–∞) or (–∞) – (–∞)
- Multiplication: ±0 × ±∞
- Division: ±0 ÷ ±0 or ±∞ ÷ ±∞
- Comparison of “<” or “>” with an Unordered operand and without “?”
- Arithmetic operation with S-NaN included in the operand. The transfer instruction (MOV) is not treated as an arithmetic operation, but the absolute value (ABS) and arithmetic negation (NEG) are treated as arithmetic operations.
- Comparison with S-NaN as operand and conversion into floating point
- Square root: If operand is less than 0

In addition to the above, an exception can be simulated by software if an invalid operation is performed on the specified source operand. Examples of this operation include IEEE754-specified functions that can be executed by software, such as the remainder mentioned below.

- Remainder xREMy if y is 0 or if x is infinity
- Conversion of a floating-point value of infinity or NaN that triggers overflow into a decimal number
- Transcendental functions such as In(–5) and cos – 1(3)

(1) If exception is enabled

The contents of the destination register are not changed, the contents of the source register are saved, and the inexact operation exception occurs.

(2) If exception is not enabled

If no other exception occurs, Q-NaN is stored in the destination register.

8.3.3 Division-by-zero exception (Z)

A division-by-zero exception occurs if a finite number with a divisor of 0 and a dividend of other than 0 is used. This exception also occurs if an operation that produces signed infinity as the result, such as In(0), sec(π/2), csc(0), and 0 – 1, is performed.

(1) If exception is enabled

The contents of the destination register are not changed, the contents of the source register are saved, and the division-by-zero exception occurs.

(2) If exception is not enabled

If no other exception occurs, a correctly signed infinite number (±∞) is stored in the destination register.

8.3.4 Overflow exception (O)

An overflow exception occurs if the exponent range is infinite and if the size of the result of the rounded floating point is greater than the maximum finite number in the destination format (an inexact operation exception occurs and the flag bit is set).

(1) If exception is enabled

The contents of the destination register are not changed, the contents of the source register are saved, and the overflow exception occurs.

(2) If exception is not enabled

If no other exception occurs, the default value that is determined by the rounding mode and the sign of the intermediate result is stored in the destination register (refer to **Table 8-1 Default Values of IEEE754 Exceptions in FPU**).

8.3.5 Underflow exception (U)

An underflow exception occurs in the following two cases.

- If the operation result is -2^{Emin} to $+2^{Emin}$ (but other than 0)
- If the accuracy drops as a result of an operation between not normalized small numbers.

IEEE754 defines many methods for detecting an underflow. However, be sure to detect an underflow by the same method whatever processing may be performed.

The following two methods may be used to detect an underflow.

- If the result calculated after rounding and with an infinite exponent range is other than 0 and within $\pm 2^{Emin}$
- If the result calculated before rounding and with an infinite exponent range and accuracy is other than 0 and within $\pm 2^{Emin}$

The MIPS architecture detects an underflow after rounding the result.

The following two methods may be used to detect a drop in accuracy.

- Denormalized loss (if a given result and the result calculated when the exponent range is infinite differ)
- Illegal result (if a given result and the result calculated when the exponent range and accuracy are infinite differ)

The MIPS architecture detects a drop in accuracy as an illegal result.

(1) If exception is enabled

If the underflow exception/inexact operation exception is enabled or if the FS bit of FCR31 and FCR28 is not set, an unimplemented operation exception (E) occurs. At this time, the contents of the destination register are not changed.

(2) If exception is not enabled

If the underflow exception and inexact operation exception are disabled and if the FS bit of FCR31 and FCR28 is set, the default value determined by the rounding mode and the sign of the intermediate result is stored in the destination register (refer to **Table 8-1 Default Values of IEEE754 Exceptions in FPU**).

8.3.6 Unimplemented operation exception (E)

The E bit is set and an exception occurs if an attempt is made to execute an instruction with an operation code reserved for future expansion or an invalid format code. The operand and the contents of the destination register are not changed. Usually, the instruction is emulated by software. If an IEEE754 exception occurs from an emulated operation, simulate that exception.

The unimplemented operation exception also occurs in the following cases, in which an abnormal operand or abnormal result that cannot be correctly processed by hardware is detected.

- If the operand is a denormalized number (except a compare instruction)
- If the operand is a Q-NaN (except compare instruction)
- If the result is a denormalized number or underflows when the underflow/inexact operation exception is enabled or when the FS bit of FCR31 and FCR28 is not set
- If a reserved instruction is executed
- If an unimplemented format is used
- If a format whose operation is invalid is used (e.g., CVT.S.S)

Caution If the instruction is a format conversion or arithmetic operation instruction, the exception occurs only when the operand is a denormalized number or NaN. The exception occurs even if the operand is a denormalized number or NaN when a transfer instruction is executed.

The VR5500 also generates the unimplemented operation exception in the following cases.

- If the result of multiplication by the MADD, MSUB, NMADD, or NMSUB instruction is a denormalized number, underflows, or overflows
- If a MIPS IV floating-point instruction is executed when the MIPS IV instruction set is not enabled
- If the value of the result is outside the range of 2^{53} – 1 (0x001F FFFF FFFF FFFF) to -2^{53} (0xFFE0 0000 0000 0000) when the format is converted from a floating-point format to a 64-bit fixed-point format
  Instruction: CEIL.L.fmt, CVT.L.fmt, FLOOR.L.fmt, ROUND.L.fmt, TRUNC.L.fmt
- If the value of the result is outside the range of 2^{31} – 1 (0x7FFF FFFF) to -2^{31} (0x8000 0000) when the format is converted from a floating-point format to a 32-bit fixed-point format
  Instruction: CEIL.W.fmt, CVT.W.fmt, FLOOR.W.fmt, ROUND.W.fmt, TRUNC.W.fmt
- If the value of the source operand is outside the range of 2^{55} – 1 (0x007F FFFF FFFF FFFF) to -2^{55} (0xFF80 0000 0000 0000) when the format is converted from a 64-bit fixed-point format to a floating-point format
  Instruction: CVT.D.fmt, CVT.S.fmt

The unimplemented operation exception can be used in any way by the system. To maintain complete compatibility with IEEE754, the unimplemented operation exception can be handled by software if it occurs.

(1) If exception is enabled

The contents of the destination register are not changed, the contents of the source register are saved, and the unimplemented operation exception occurs.

(2) If exception is not enabled

This exception cannot be disabled because there is no corresponding enable bit.

8.4 Saving and Restoring Status

The LDC1 or SDC1 instruction is executed for 16 doublewords[Note] to save or restore the status of a floating-point register to or from memory. Information on FCR31, FCR28, FCR26, and FCR25 is saved to or restored from a CPU register by the CFC1 or CTC1 instruction. Usually, FCR31 is saved first and restored last.

If the FPU is executing a floating-point instruction when FCR31, FCR28, FCR26, or FCR25 is read, the instruction may be completely executed or reported as an exception. Because the architecture does not allow a pending instruction to cause an exception, if execution of the pending instruction cannot be completed, that instruction is transferred to an exception register (if any). Information such as the type of the exception is stored in FCR31, FCR28, FCR26, or FCR25. When the status is restored, FCR31 indicates that an exception is pending.

By writing a value of 0 to the Cause bits of FCR31 or FCR26, all pending exceptions can be cleared, and resumption of the normal processing is enabled after the status of the floating-point register has been restored.

The Cause bits of FCR31 and FCR21 hold the result of only one instruction. The FPU checks the operand before executing an instruction to judge whether an exception may occur. If an exception may occur, the FPU executes this instruction by using a stall, so that two or more instructions (that may cause an exception) are not executed at the same time.

Note Thirty-two doublewords if the FR bit of the Status register in CP0 is set to 1

8.5 Handler for IEEE754 Exceptions

IEEE754 recommends an exception handler that can store calculation results in the destination register regardless of which of the five standard exceptions occurs.

The exception handler can identify the following by using the EPC register to search for an instruction.

- Occurrence of exception during instruction execution
- Instruction under execution
- Format of destination

To obtain the correctly rounded result if an overflow, underflow (except the conversion instruction), or inexact operation exception occurs, the exception handler must have software that checks the source register and simulates instructions.

If an invalid operation exception or division-by-zero exception occurs or if an overflow exception or underflow exception occurs during floating-point conversion, the exception handler must have software that can obtain the value of the operand by checking the source register of the instruction.

IEEE754 recommends that, if possible, the overflow and underflow exceptions have a priority higher than the inexact operation exception. This priority is set by software. The hardware sets the bits of both the overflow and the underflow exceptions, and inexact operation exception.

CHAPTER 9 INITIALIZATION INTERFACE

9.1 Functional Outline

The VR5500 can be reset in three ways by using the ColdReset# and Reset# signals.

- Power-on reset
  When the power supply has been stabilized after power application, all clocks are started. A power-on reset completely initializes the internal information of the processor without saving any status information.

- Cold reset
  If the ColdReset# signal is asserted while the processor is operating, all clocks are restarted and the test interface circuit is also initialized. A cold reset completely initializes the internal statuses of the processor without saving any status information.

- Warm reset
  Although the processor is restarted, the clock and test interface circuits are not affected. By using a warm reset, most of the internal statuses of the processor can be retained. However, the contents of registers are undefined.

After reset, the processor serves as the bus master and drives the SysAD bus.

When adjusting a system reset with other system elements, the following must be noted: Generally, the operation is undefined if a bus error occurs immediately before, during, and immediately after reset. In addition, reset initializes only a part of the internal status. Therefore, completely initialize the processor by software.

The statuses of the registers, control signals, and current are undefined from when power is applied to when reset is completed.

9.2 Reset Sequence

The following two signals are used during reset.

(1) ColdReset#

Assert this signal to execute a power-on reset or cold reset. Synchronize it with SysClock to deassert it.

(2) Reset#

Assert this signal to execute all reset operations. This signal does not have to be synchronized with the ColdReset# signal when it is asserted. When only the Reset# signal is asserted, a warm reset is started. To deassert this signal, synchronize it with SysClock.

9.2.1 Power-on reset

The sequence of a power-on reset is as follows.

1. Confirm that stable V_{DD} and $V_{DD}IO$ are supplied within the specified voltage range. Also confirm that the system clock of the specified frequency is stable and continues operating.
2. After power supply has been stabilized, assert the ColdReset# signal for the duration of at least 64 K SysClock cycles. Deassert the ColdReset# signal in synchronization with SysClock.
3. The processor starts operating when the Reset# signal is asserted after the ColdReset# signal has been deasserted. Keep the Reset# signal active for the duration of at least 16 SysClock cycles after the ColdReset# signal has been deasserted. Deassert the Reset# signal in synchronization with SysClock.

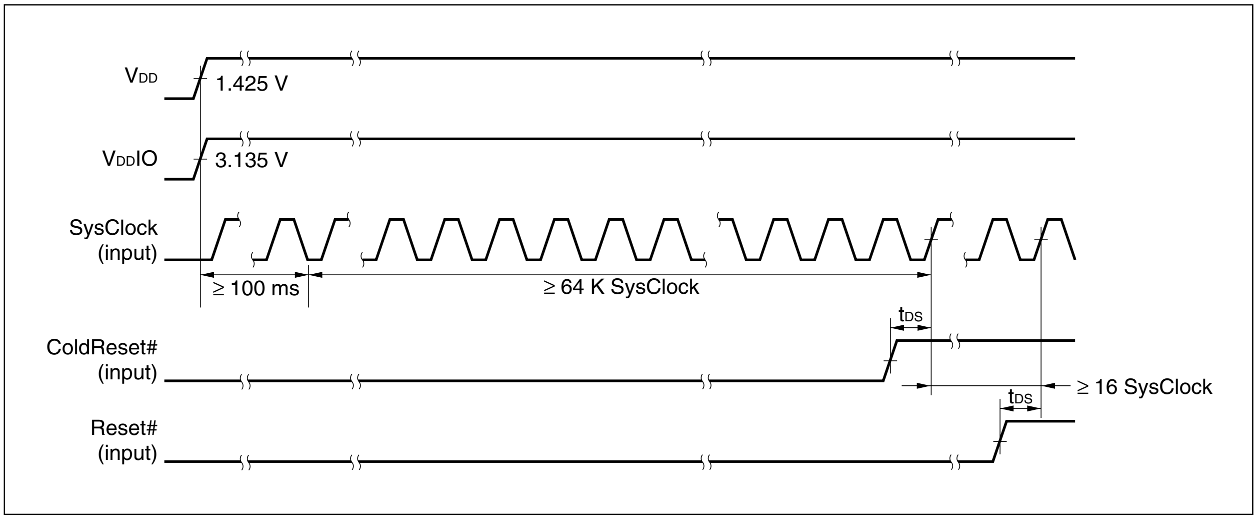
The status of the initialization signal (refer to **9.3**) is latched 1 SysClock cycle after the ColdReset# signal has been deasserted. Set the input level of the initialization signal before starting a power-on reset. Keep the level from changing during operation.

At reset, the processor serves as the bus master and drives the SysAD bus.

When the Reset# signal is deasserted, the processor branches to the reset exception vector and starts execution of the reset exception handler.

Figure 9-1 shows the timing of a power-on reset.

Figure 9-1. Power-on Reset Timing

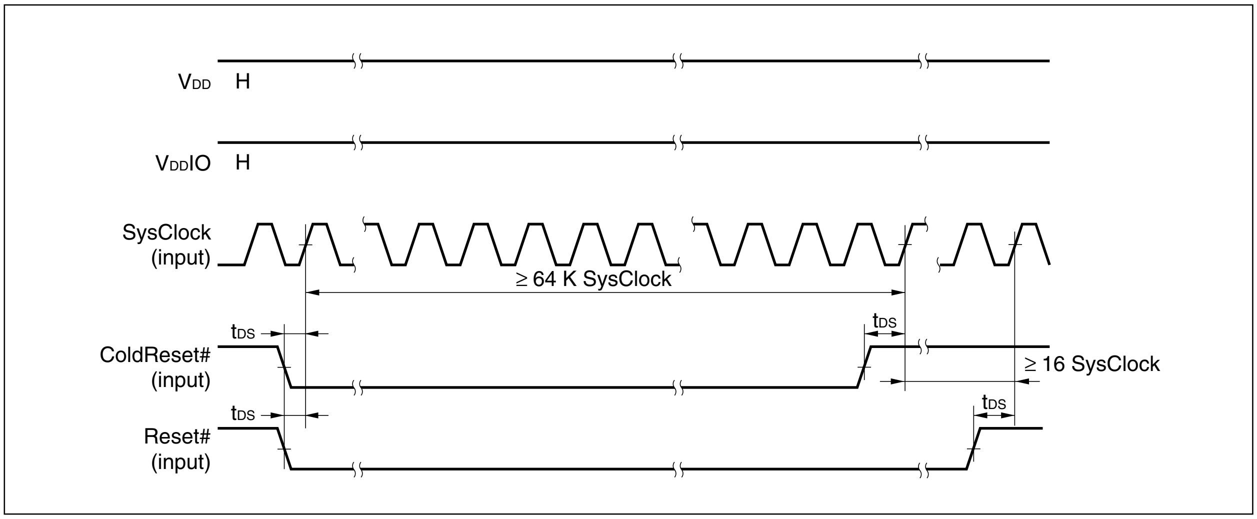


9.2.2 Cold reset

The sequence of a cold reset is the same as that of a power-on reset except that the power supply must be stabilized before the reset signal is asserted.

Figure 9-2 shows the timing of a cold reset.

Figure 9-2. Cold Reset Timing



9.2.3 Warm reset

A warm reset is started if the Reset# signal is asserted in synchronization with SysClock. Keep the Reset# signal active for the duration of at least 16 SysClock cycles before deasserting it in synchronization with SysClock. A warm reset causes the processor to generate a soft reset exception.

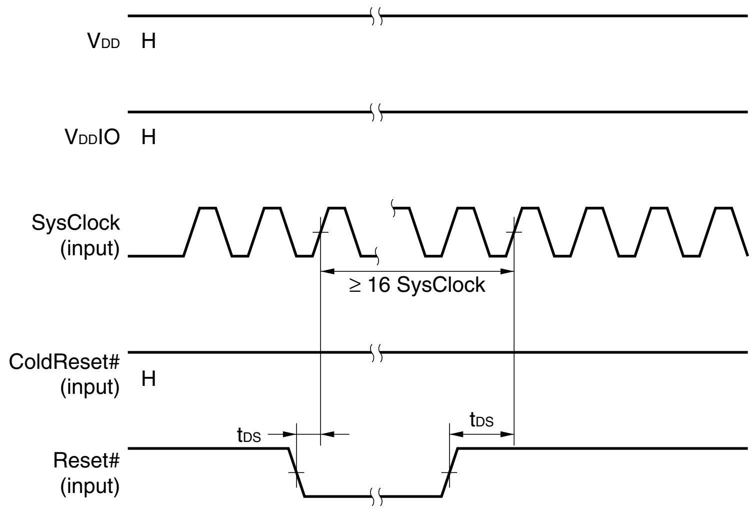
Because a warm reset is started as soon as the Reset# signal has been asserted, multiple-cycle operations such as processing of a cache miss and floating-point instructions are stopped, and the data and results may be lost.

At reset, the processor serves as the bus master and drives the SysAD bus. When executing a warm reset while a SysAD bus transaction is in progress, also reset the external agent so that a conflict does not occur on the SysAD bus.

When the Reset# signal is deasserted, the processor branches to the reset exception vector and starts executing the soft reset exception handler.

Figure 9-3 shows the timing of a warm reset.

Figure 9-3. Warm Reset Timing



9.2.4 Processor status at reset

After a power-on reset, cold reset, and warm reset, all the internal statuses of the processor are reset and the processor starts program execution from the reset vector.

The internal settings of the processor are retained after a warm reset has been executed. However, the status of the cache may be retained or not depending on whether processing of a cache miss has been aborted by resetting the processor. In addition, because the VR5500 has a non-blocking structure, updating registers is canceled if execution of a load instruction is not complete when a reset is executed.

The branch history table is initialized by a power-on reset and cold reset.

The statuses of the registers, control signals, and current are undefined from when power is applied to when reset is completed.

9.3 Initialization Signals

The VR5500 has eight types of input signals that are sampled during initialization. These signals are used to set the division ratio of the clock, the byte configuration of memory, and the protocol of the system interface.

Set the level of these signals before starting a power-on reset. Keep the level unchanged during operation.

(1) DivMode(2:0)

These signals specify the division ratio of the internal processor clock (PClock) and external system clock (SysClock). Eight types of division ratios can be set: 2, 2.5, 3, 3.5, 4, 4.5, 5, and 5.5.

(2) BigEndian

This signal specifies the byte order used by the processor during operation. When it is high, big endian is specified; when it is low, little endian is specified.

(3) BusMode

This signal specifies the bus width of the system interface. When this signal is high, the bus width is 64 bits; when it is low, the bus width is 32 bits.

(4) TIntSel

This signal specifies the interrupt source allocated to the IP7 bit of the Cause register. When it is high, the timer interrupt is selected, and an interrupt request executed by asserting the Int5# pin or an external write request (SysAD5) is ignored. When this signal is low, the interrupt request executed by the Int5# pin or an external write request (SysAD5) is selected, and the timer interrupt request is ignored.

(5) DisDValidO#

This signal specifies the operation of the ValidOut# signal. When this signal is low, the ValidOut# signal is asserted only during the address issuance cycle; when it is low, the ValidOut# signal is asserted even if address issuance is stalled due to ready control.

(6) DWBTrans#

This signal specifies expansion of the data transfer size when the system interface is 32 bits wide. If this signal is low, doubleword block transfer is enabled; it is disabled when this signal is high.

(7) O3Return#

This signal specifies the protocol of the system interface. When it is low, the out-of-order return mode is specified; when it is high, the normal mode is specified.

(8) DrvCon

This signal specifies the impedance control level of the output driver. When it is high, the level is weak; when it is low, the level is normal. It is recommended to set this signal to the low level (normal) with the VR5500.

CHAPTER 10 CLOCK INTERFACE

This chapter explains the clock interface used in the VR5500.

10.1 Term Definitions

This manual uses the following terms when describing signals.

- "Rising edge" indicates the point of transition from low level to high level.
- "Falling edge" indicates the point of transition from high level to low level.
- "Clock-Q delay" indicates the time required between when a signal inputs data to a device (clock) and when it outputs data from a device (Q).

Figures 10-1 and 10-2 illustrate the meanings of these terms.

Figure 10-1. Signal's Transition Points

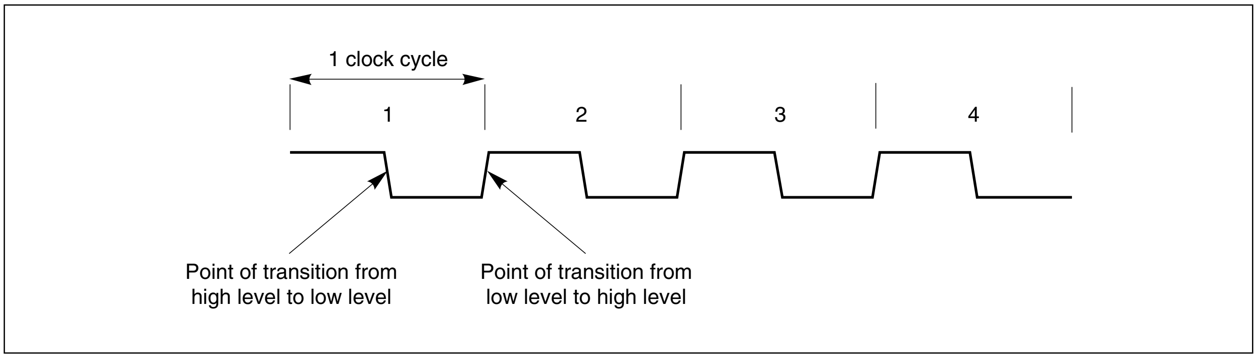
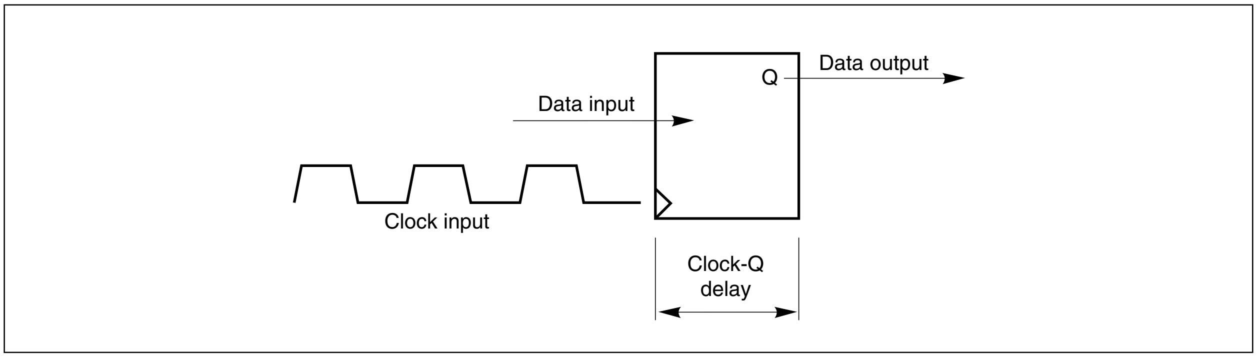

1 clock cycle
1
2
3
4
Point of transition from high level to low level
Point of transition from low level to high level

Figure 10-2. Clock-Q Delay



10.2 Basic System Clock

The VR5500 uses the following clock signals.

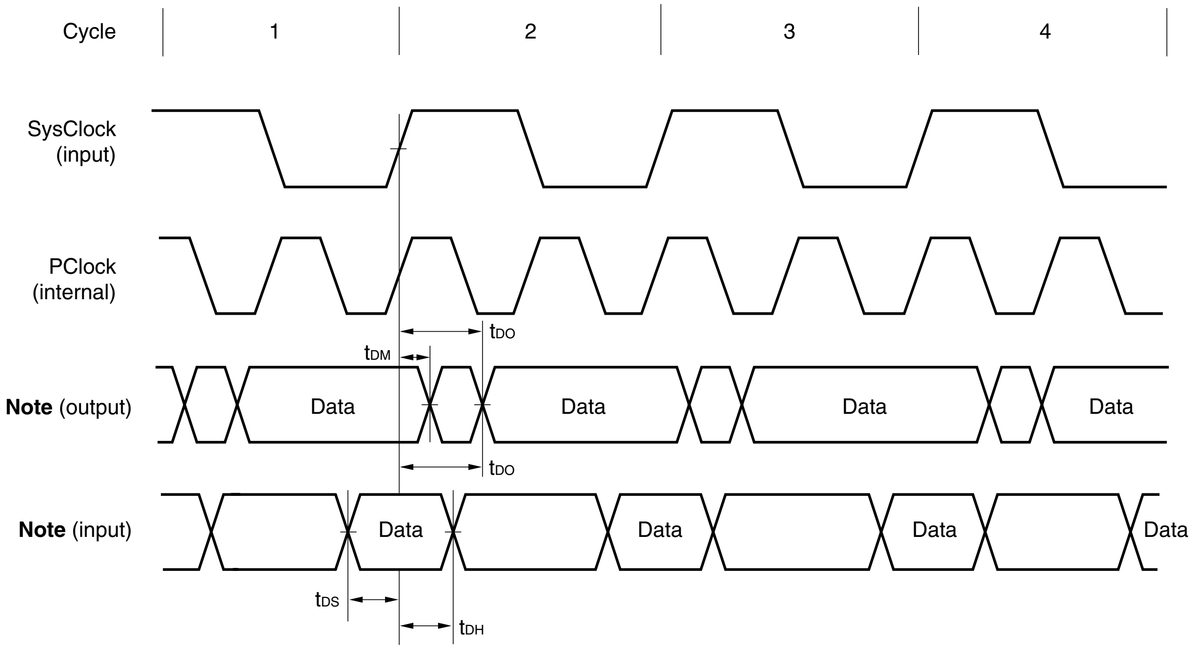
(1) SysClock

The internal clock of the VR5500 is generated based on SysClock. The interface with the external device also operates based on SysClock.

(2) PClock

The frequency ratio of PClock to SysClock can be selected from 2:1, 2.5:1, 3:1, 3.5:1, 4:1, 4.5:1, 5:1, and 5.5:1.
This ratio is set by the signals input from the DivMode(2:0) pins at reset.
All the internal registers and latches use PClock.

Figure 10-3. When Frequency Ratio of SysClock to PClock Is 1:2



Note SysAD(63:0), SysADC(7:0), SysCmd(8:0), SysID(2:0)

10.2.1 Synchronization with SysClock

The processor data changes when t_{DM} has elapsed after the rising edge of SysClock was detected, and is in the stable output status when t_{DO} has elapsed. This time is the sum of the maximum value of the Clock-Q delay of the processor output register and the maximum value of the delay when the data passes through the processor output driver.

Keep the data supplied to the processor stable for the duration of at least t_{DS} before SysClock rises, and for the duration of t_{DH} after the rising edge of SysClock, as shown in Figure 10-3.

10.3 Phase Lock Loop (PLL)

The processor has an internal PLL circuit that is used to synchronize SysClock with PClock. Because of the nature of the PLL circuit, however, a clock synchronized with the frequency of SysClock can be generated in a limited range.

The clock generated by using the PLL circuit has specific uncertainty called jitter. The clock synchronized with SysClock by the PLL circuit leads or lags behind SysClock, up to the maximum permissible value t_J of jitter.

To obtain accurate I/O timing parameters, therefore, add t_J to t_{DS}, t_{DH}, and t_{DO}, and subtract t_J from t_{DM}.

CHAPTER 11   CACHE MEMORY

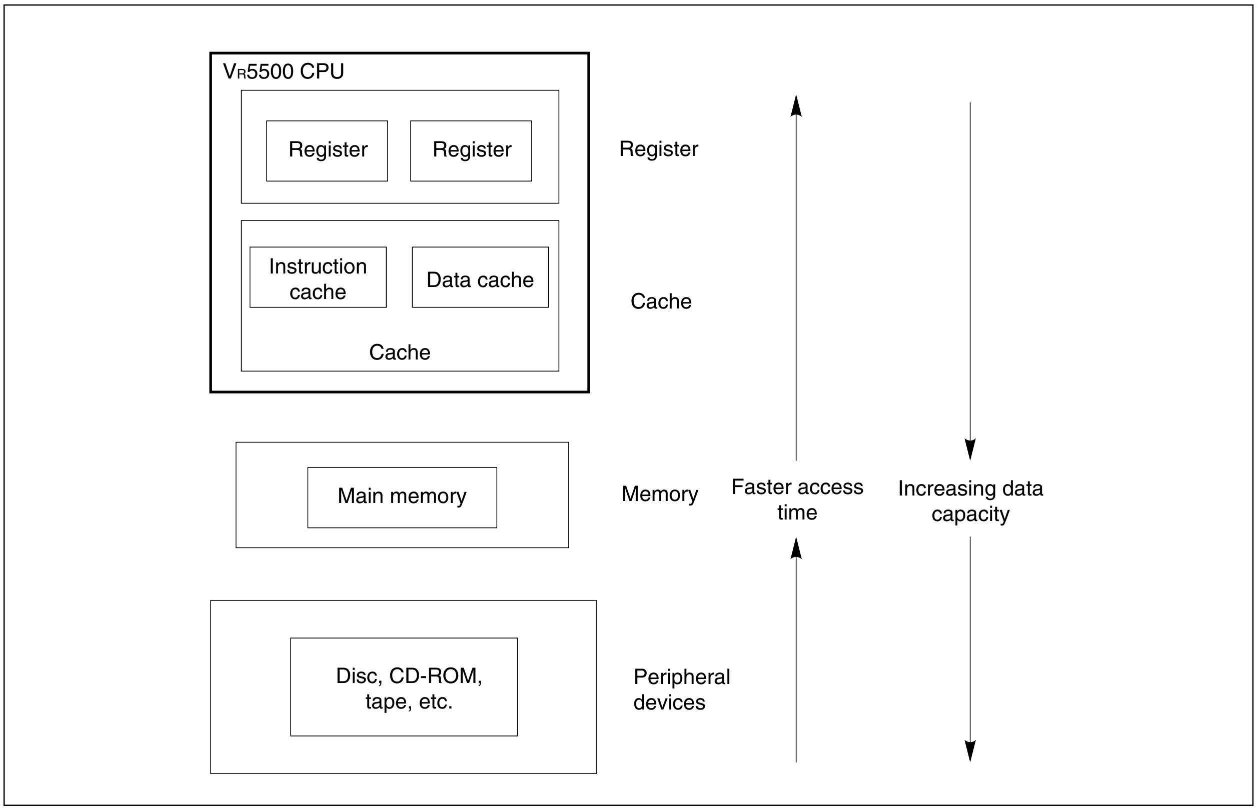
This chapter explains the cache memory: its place in the VR5500 core memory organization, and the individual organization of the caches.

11.1   Memory Organization

Figure 11-1 shows the VR5500 core system memory hierarchy. In the logical memory hierarchy, the caches are located 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 example, main memory (physical 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 example, it takes longer to access data in the main memory than in the CPU on-chip registers.

Figure 11-1. Logical Hierarchy of Memory



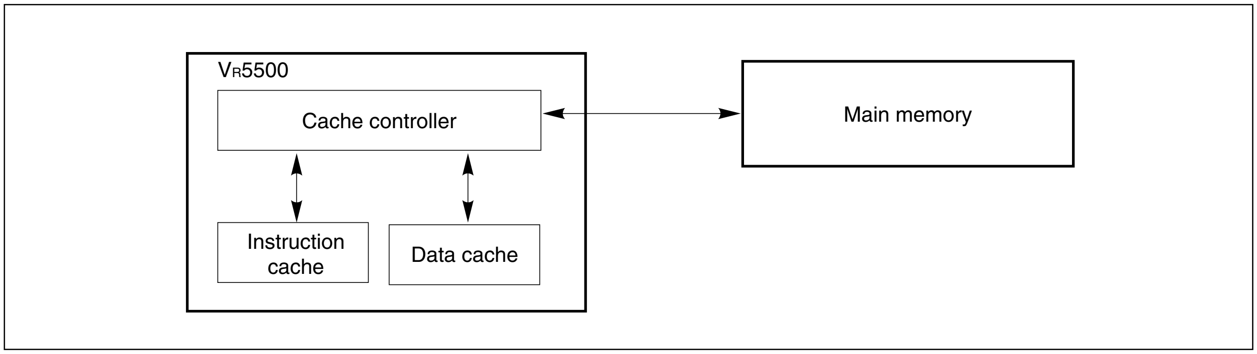
11.1.1 Internal cache

The VR5500 has two caches. One of them is an instruction cache that holds instructions (program). The other is a data cache that holds data.

When writing data to the data cache, translation of the store address and tag check are performed in the first phase, and then the data is written to RAM in the next phase.

Figure 11-2 shows the relationship between the cache and memory.

Figure 11-2. Internal Cache and Main Memory



The features of the internal cache are as follows.

- Index using virtual address
- Physical address held by tag
- Coherency with memory maintained by writeback or write through
- Data management by two-way set associative method
- Line lock can be specified
- Cache line replacement by LRU (Least Recently Used) algorithm
- Non-blocking structure (data cache only)

The size of both the instruction and data caches of the VR5500 is 32 KB.

11.2 Configuration of Cache

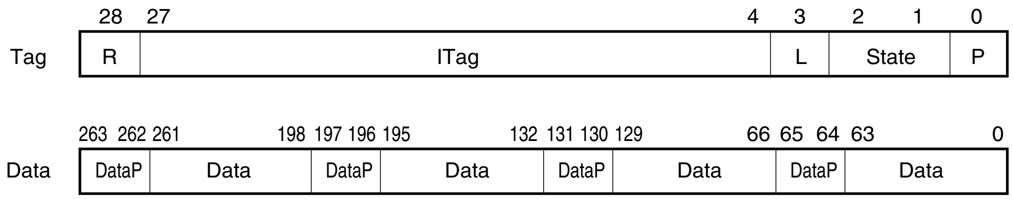
This section explains the configuration of the internal data and instruction caches of the VR5500.

A cache consists of blocks called cache lines. A cache line is the minimum unit of information that can be fetched from the main memory to the cache, and is divided into a tag and data. The size of a cache line of both the instruction cache and data cache is 8 words (32 bytes).

11.2.1 Configuration of instruction cache

Figure 11-3 shows the format of an 8-word (32-byte) instruction cache line.

Figure 11-3. Format of Instruction Cache Line

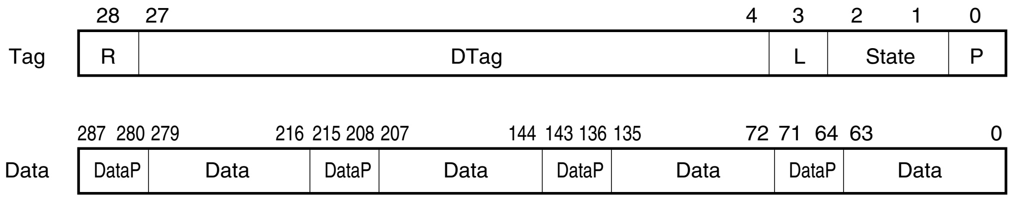


ITag: Instruction tag
L: Lock bit (line lock status)
State: Status bit (line status)
R: LRU bit (way indication of candidate for replacement)
P: Parity bit (even parity for ITag)
DataP: Even parity for Data (in word units)
Data: Data of instruction cache

11.2.2 Configuration of data cache

Figure 11-4 shows the format of an 8-word (32-byte) data cache line.

Figure 11-4. Line Format of Data Cache



DTag: Data tag
L: Lock bit (line lock status)
State: Status bit (line status)
R: LRU bit (way indication of candidate for replacement)
P: Parity bit (even parity for DTag)
DataP: Even parity for Data (in byte units)
Data: Data of data cache

11.2.3 Location of data cache

The VR5500 manages cache data by a two-way set associative method. This method divides the cache into two blocks of memory spaces (ways), and allocates two cache lines to the same index (refer to **11.3.5 Accessing cache**).

11.3 Cache Operations

As described earlier, caches provide temporary data storage, and they speed up memory accesses as seen by the user. In general, the processor accesses cache-resident instructions or data using the following procedure.

(1) The processor attempts to access the instruction used next or data in the appropriate cache via the on-chip cache controller.
(2) The cache controller checks to see if this instruction or data is present in the cache.

- If the instruction/data is present, the CPU retrieves it. This is called a cache hit.
- If the instruction/data is not present in the cache, the cache controller retrieves it from the main memory. This is called a cache miss.

(3) When the required data or instruction is found, the cache controller passes it to the processor. The processor then continues operating.

If a cache miss occurs, data is read from the main memory and one of the cache line is overwritten. This is called replacing a cache line.

The VR5500 manages the cache by a two-way set associative method, with two cache lines allocated to one index. If a cache miss occurs, which of the two lines is to be replaced is determined by the LRU (Least Recently Used) method. The way that is a candidate for replacement is indicated by the LRU bit of the cache tag.

The cache of the VR5500 has a line lock function. If a cache line is locked when it is allocated, that line is not replaced even if a cache miss occurs. If a cache miss occurs while the line of both the ways is locked, however, one of the cache lines is unlocked in accordance with the LRU bit. A cache line is locked or unlocked by the CACHE instruction. The setting status of locking is indicated by the lock bit of the cache tag.

11.3.1 Coherency of cache data

It is possible for the same data to be in two places simultaneously: the main memory and a cache. This coherency of this data is maintained by using the writeback or write-through method.

With the VR5500, the data cache management technique can be selected from writeback and write through, depending on the setting of the EntryLo register or Config register of CP0.

The writeback method stores write data only in the cache, without writing it directly to the main memory[Note]. Some time later the data written to the cache is independently transferred to the main memory. In the VR5500, a modified cache line is not written back to the memory until the cache line is to be replaced either in the course of satisfying a cache miss, or during the execution of a writeback CACHE instruction.

With the write-through method, data written to the memory is also written to the cache simultaneously.

11.3.2 Replacing instruction cache line

If a miss occurs in the instruction cache, the cache line is replaced by using sub-block ordering.

If a miss occurs in the instruction cache, the processor issues a memory read request. This means that the processor reads the cache line it requests from the main memory and writes it to the instruction cache. At this time, execution of the pipeline is resumed and the instruction cache is accessed again.

11.3.3 Replacing data cache line

If a miss occurs while data is being loaded from or stored in a cache, the cache line is replaced in compliance with the following rules.

(1) Data load miss

If the cache line on which a miss has occurred is not dirty, that cache line is replaced with a new cache line.

If the cache line is dirty, the cache line is first transferred to the write transaction buffer. Then the cache line on which a miss occurred is replaced with a new cache line, and the data transferred to the write transaction buffer is written to memory.

(2) Data store miss

(a) With writeback cache

If the cache line on which a miss has occurred is not dirty, that cache line is replaced by store data merged with a new cache line.

If the cache line is dirty, that cache line is first transferred to the write transaction buffer. Then store data merged with a new cache line is written to the cache, and the data transferred to the write transaction buffer is written to memory.

(b) With write-through cache

If the cache line on which a miss has occurred is not dirty, that cache line and memory contents are replaced by store data merged with a new cache line. If the cache line is dirty, that cache line is first transferred to the write transaction buffer. Then store data merged with a new cache line is written to the cache and memory.

11.3.4 Speculative replacement of data cache line

The VR5500 adds an unguarded attribute to the algorithm of the data cache. This attribute can be selected according to the setting of the EntryLo register or Config register of CP0, when the data cache is used (refer to **CHAPTER 5 MEMORY MANAGEMENT SYSTEM**).

The VR5500 speculatively executes instructions by using branch prediction and an out-of-order mechanism. If a data load miss or data store miss occurs as a result of speculative execution of an instruction, the refill buffer once holds data to replace cache lines. If the conventional algorithm is selected for the data cache, replacement is not started until this instruction is committed, even if the refill buffer becomes full.

By contrast, replacement can be started even before this instruction is committed if the unguarded attribute is selected. Speculative replacement like this cannot be stopped once it has been started, regardless of whether its result is necessary or not.

Caution Make sure that the following conditions are satisfied in the area where the unguarded attribute is specified.

- **The OS uses the virtual address space and all spaces are contiguous.**
- **If I/O is connected, a device whose status is not changed even if read must be used.**

If the address space is not contiguous, the result cannot be discarded when a load instruction is speculatively executed because a bus error exception occurs, and the system hangs up.
If an I/O whose status may be changed when read is connected, the result cannot be discarded because the status on the I/O side is changed when a load instruction is speculatively executed.

Remarks 1. Speculative processing using the unguarded attribute is only executed for the data cache.

2. Of the accesses to the area of the unguarded attribute, a read request is speculatively output from the system interface before the instruction is committed, but a write request is output after the instruction has been committed. By contrast, if an access is made to the uncached area, a read request is also output to the system interface after the instruction has been committed.

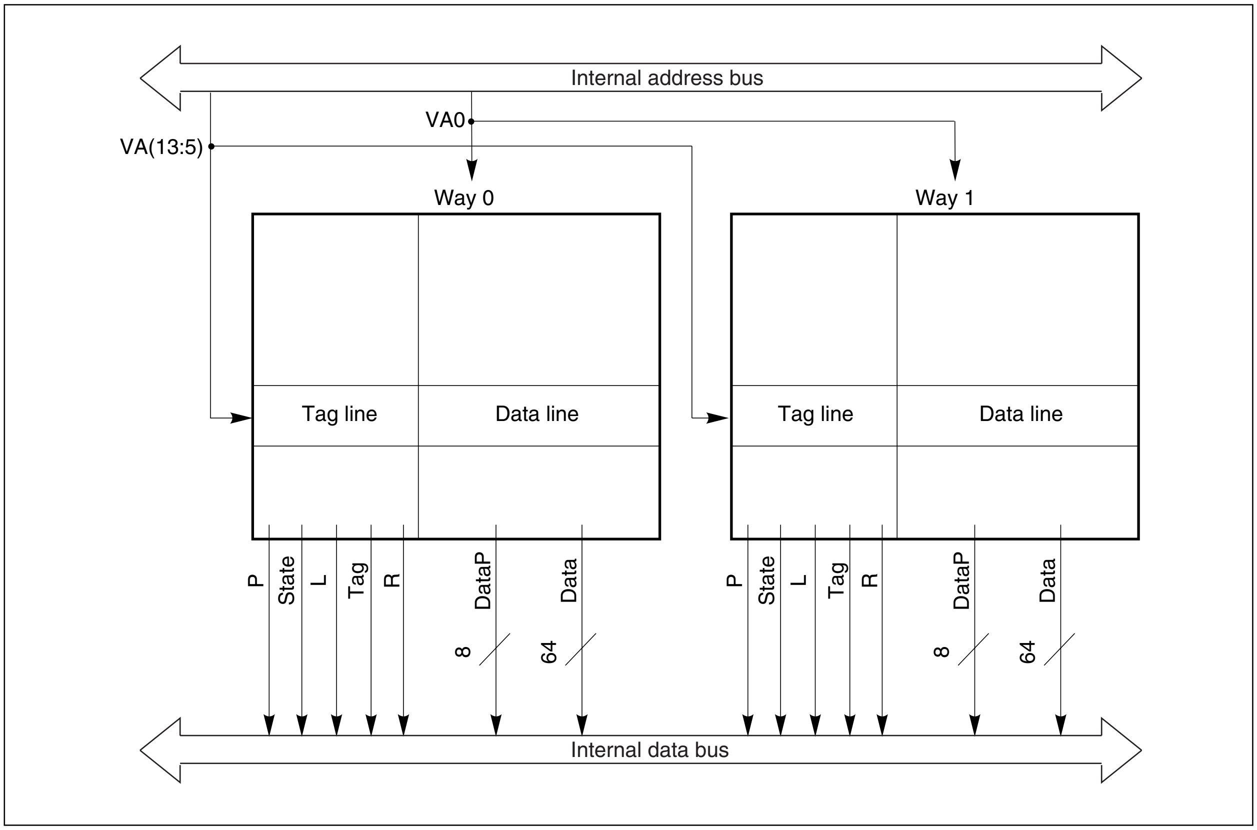
11.3.5 Accessing cache

The CACHE instruction is used to change the status of the cache line or to write back cache data (for details, refer to **CHAPTER 17 CPU INSTRUCTION SET**).

Part of the virtual address (VA) is used to index the instruction cache and data cache. Because the cache size of the VR5500 is 32 KB and has a two-way set, the most significant bit is VA13. In addition, because the line size is 8 words (32 bytes), the least significant bit is VA5. The way to be accessed is specified by the LRU method for Hit, Fill, and Fetch_and_Lock operations, and by VA0 for other operations.

Figure 11-5 shows the relationship between index and data output of the cache.

Figure 11-5. Index and Data Output of Cache



11.4 Status of Cache

The cache line may be in the following three states, which indicate the validity of data and coherency with the main memory.

The status of the cache line is undefined after reset. Initialize it by software.

(1) Instruction cache

The instruction cache may be in either of the following two states.

- Invalid: State in which the cache line does not have valid information.
  A cache line in this state cannot be used. Set all the cache lines after a warm reset to Invalid by software. A cache line not in the Invalid status is assumed to have valid information.
  Neither a cold reset nor a warm reset makes the cache status Invalid. The cache is invalidated by software.
- Clean: State in which the cache line has valid information that has been fetched from the main memory. It can be specified by software whether the cache line is locked or not.

(2) Data cache

The data cache may be in any of the following three states.

- Invalid: State in which the cache line does not have valid information.
  The cache line in this state cannot be used. Set all the cache lines after a warm reset to Invalid by software. A cache line not in the Invalid status is assumed to have valid information.
  Neither a cold reset nor a warm reset makes the cache status Invalid. The cache is invalidated by software.
- Clean: State in which the cache line has valid information that has not been changed after being fetched from the main memory. It can be specified by software whether the cache line is locked or not.
- Dirty: State in which the cache line has valid information that has been changed after being loaded from the main memory. It can be specified by software whether the cache line is locked or not.

A cache line in the Clean or Dirty status may be changed when the processor executes a certain type of CACHE instruction operation. For the operations of the CACHE instruction, refer to **CHAPTER 17 CPU INSTRUCTION SET**.

11.5 Manipulating Cache by External Agent

The VR5500 does not allow an external agent to check or manipulate the statuses and contents of either of the caches.

CHAPTER 12 OVERVIEW OF SYSTEM INTERFACE

The processor uses the system interface to access the external resources necessary for processing a cache miss and in the uncached area, and the external agent uses the system interface to access the internal resources of the processor.

The system interface of the VR5500 has several mode, including a mode in which another read request can be issued even if the first read operation is not complete and a read response can be separated and returned, and a mode that is compatible with the VR5000. These modes can be selected by a combination of the levels input to the initialization pins at reset.

This chapter explains the bus modes and basic operations of the system interface of the VR5500.

12.1 Definition of Terms

The following terms are used in CHAPTERS 13, 14, and 15.

- External agent
  A device connected to the processor via the system interface which processes requests issued by the processor

- System event
  An event that is generated in the processor and requests access to the external resources. For example, the following events are included.

  - Occurrence of a miss in the instruction cache when an instruction is fetched
  - Occurrence of a miss in the data cache when a load/store instruction is executed
  - Execution of a load/store instruction to the uncached area.

- Sequence
  Requests successively generated by the processor to process a system event

- Protocol
  Signal transition in each cycle of the system interface pins by which the processor or external agent issues requests

- Syntax
  Definition of the bit pattern of a code bus such as a command bus

12.2 Bus Modes

The VR5500 has the following five types of bus modes. For details of the operation, refer to the corresponding chapter.

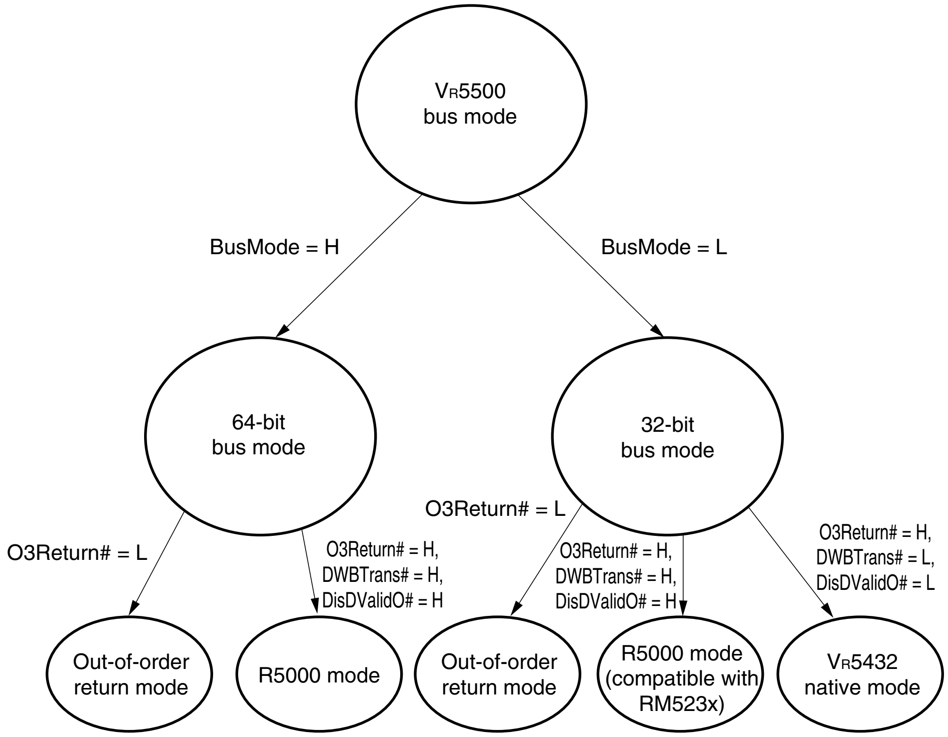
- 64-bit R5000 mode
  → Refer to **CHAPTER 13 SYSTEM INTERFACE (64-BIT BUS MODE)**.
- 64-bit out-of-order return mode
  → Refer to **CHAPTER 15 SYSTEM INTERFACE (OUT-OF-ORDER RETURN MODE)**.
- 32-bit R5000 mode (compatible with PMC-Sierra's RM523x)
  → Refer to **CHAPTER 14 SYSTEM INTERFACE (32-BIT BUS MODE)**.
- 32-bit VR5432 native mode
  → Refer to **CHAPTER 14 SYSTEM INTERFACE (32-BIT BUS MODE)**.
- 32-bit out-of-order return mode
  → Refer to **CHAPTER 15 SYSTEM INTERFACE (OUT-OF-ORDER RETURN MODE)**.

The bus modes other than the out-of-order return mode are collectively called the normal mode.

These modes are selected by using the BusMode, O3Return#, DWBTrans#, and DisDValidO# signals at reset.

The figure below shows the relationship between the setting of each signal and the mode to be selected.

Figure 12-1. Bus Modes of VR5500



Remarks 1. H: high level, L: low level

2. When the O3Return# signal is low, the DWBTrans# and DisDValidO# signals can be set to any level, but keep the level from changing during operation.

12.3 Outline of System Interface

12.3.1 Interface bus

The SysAD bus (address/data bus) and SysCmd bus (command bus) are the main communication buses of the system interface. Because the both the buses are bidirectional buses, they can be driven by a processor that issues processor requests or an external device that issues external requests (for details, refer to **12.4.4 Processor request and external request**).

A request that passes through the system interface consists of the following.

- Address
- Response data to read request or write data to write request
- Command specifying type of request/data

Figure 12-2 shows the interface bus in the 64-bit bus mode, and Figure 12-3 shows the interface bus in the 32-bit bus mode.

Figure 12-2. System Interface Bus (64-Bit Bus Mode)

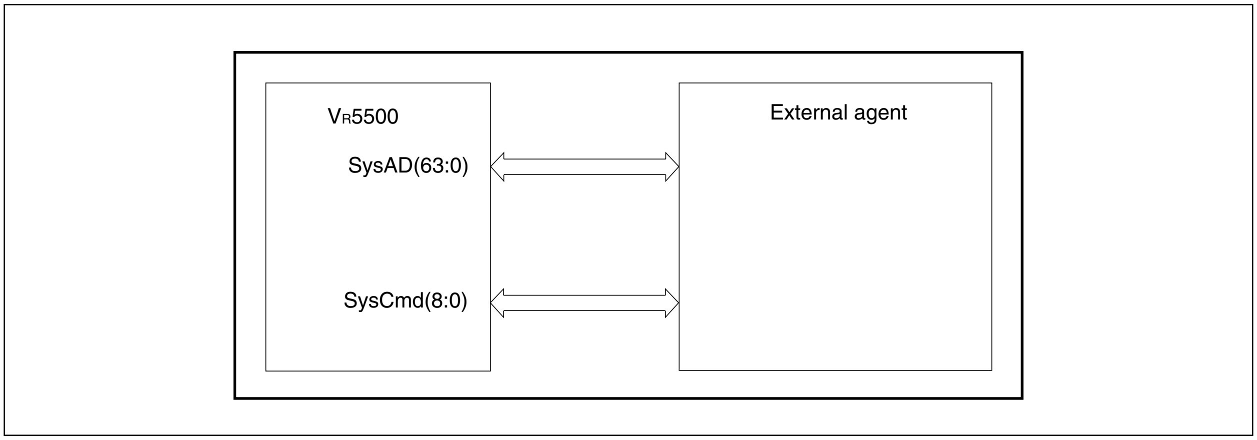
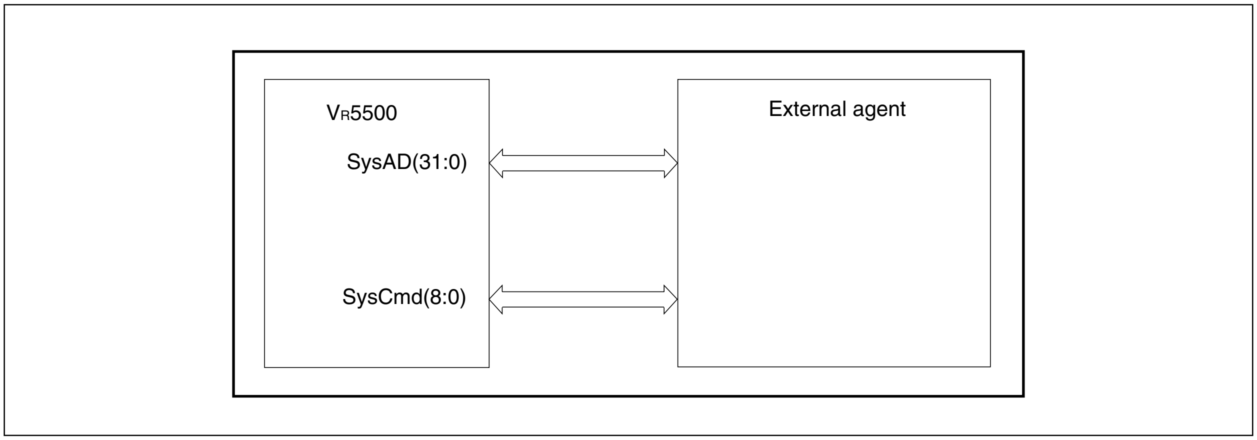


Figure 12-3. System Interface Bus (32-Bit Bus Mode)



12.3.2 Address cycle and data cycle

A cycle in which a valid address is on the SysAD bus is called an address cycle. A cycle in which valid data is on the SysAD bus is called a data cycle. The VR5500 uses the ValidOut# signal to indicate that the address/data output to the system bus is valid. The external agent uses the ValidIn# signal to indicate that the address/data output to the system bus is valid. The SysCmd bus identifies the contents of the SysAD bus cycle in a valid cycle. The most significant bit of the SysCmd bus always indicates whether the current cycle is an address cycle or a data cycle.

The SysCmd bus indicates the following contents when the ValidOut# or ValidIn# signal is active.

- In an address cycle (SysCmd8 = 0), SysCmd(7:0) on the SysCmd bus is a system interface command.
- In a data cycle (SysCmd8 = 1), SysCmd(7:0) on the SysCmd bus is a data identifier.

For details of the command and data identifier codes, refer to the descriptions on system interface commands and data identifiers in CHAPTERS 13, 14, and 15.

12.3.3 Issuance cycle

(1) Processor request

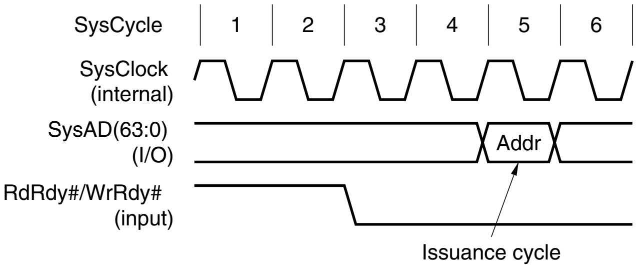
The processor issues two types of requests: a processor read request and a processor write request.

The issuance cycle of the processor read request is determined by the status of the RdRdy# signal, and that of the processor write request is determined by the status of the WrRdy# signal. The issuance cycle is a cycle that is valid in the address cycle of each processor request. Only one issuance cycle exists per processor request.

To define the issuance cycle of an address cycle, assert the Rdy#/WrRdy# signal on the external agent side up to two cycles before the address cycle of a processor read/write request, as shown in Figure 12-4.

To set an address cycle as the issuance cycle, do not deassert the RdRdy#/WrRdy# signal until that address cycle is started.

Figure 12-4. Status of RdRdy#/WrRdy# Signal of Processor Request



(2) Processor request and external request

The processor releases the system interface to the slave status and receives an external request in response to the ExtRqst# signal from the external agent even when it is about to issue a processor request.

If issuance of a processor request conflicts with issuance of an external request, the processor takes either of the following actions.

- Completes issuance of the processor request before receiving the external request.
- Releases the system interface to the slave status without completing issuance of the processor request.

In the latter case, the processor issues the processor request after the external request has been completed (if the processor request is still necessary).

12.3.4 Handshake signal

The processor manages the flow of requests by using the following seven control signals.

(1) RdRdy# and WrRdy# signals

The external agent uses these signals to indicate whether it is ready to receive a new read transaction or a new write transaction.

(2) ExtRqst#, Release#, and PReq# signals

These signals are used to control transfer between the SysAD bus and SysCmd bus. The ExtRqst# signal is used by the external agent to indicate that it needs the right to control the interface. The Release# signal is asserted by the processor when the processor grants the external agent the right to control the system interface. The PReq# signal is used by the processor to indicate that it needs the right to control the interface.

(3) ValidOut# and ValidIn# signal

The processor uses the ValidOut# signal and the external agent uses the ValidIn# signal to indicate valid command/data on the SysCmd or SysAD bus.

12.3.5 System interface bus data

The data shown in Table 12-1 is driven on the SysAD and SysCmd buses. The symbols in this table are used in the timing charts shown in the latter part of this chapter.

Table 12-1. System Interface Bus Data

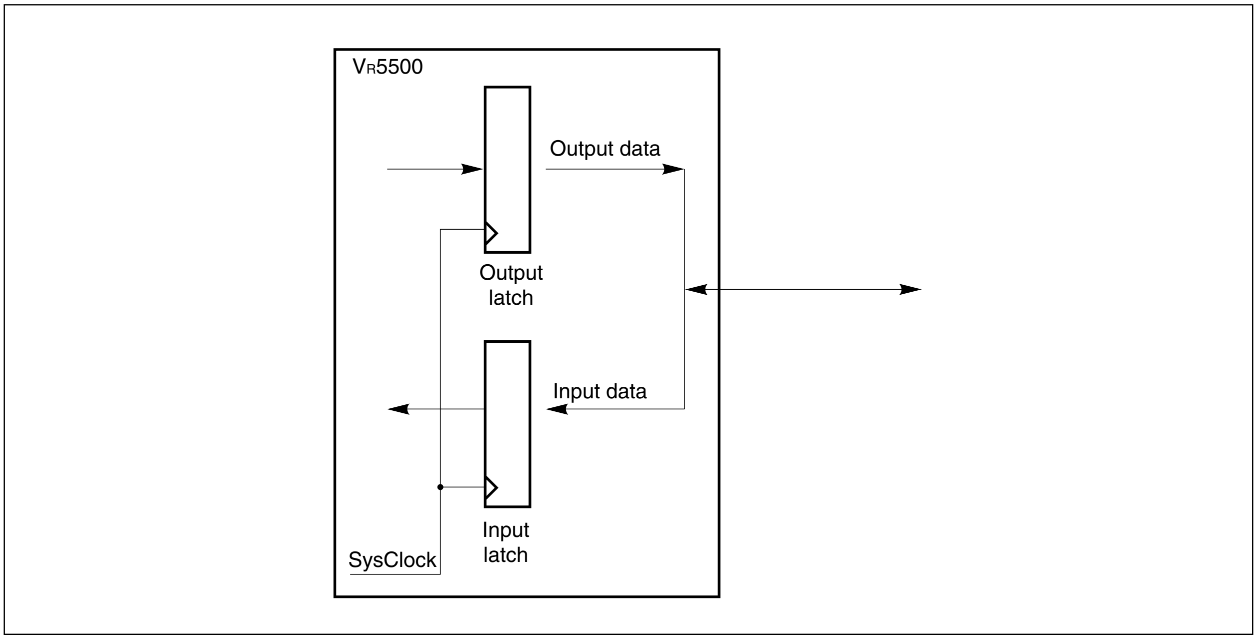
Range	Symbol	Meaning
Common	Unsd	Unused
SysAD(63:0)	Addr	Physical address
	Data<n>	(Element n + 1 of) data
SysCmd(8:0)	Cmd	Unspecific system interface command
	Read	Read request command of processor or external agent
	Write	Write request command of processor or external agent
	SINull	External null request command for releasing system interface
	NEOD	Data identifier of last data element
	NData	Data identifier of data element other than last

12.4 System Interface Protocol

Figure 12-5 shows an operation between registers that is performed via the system interface. The output signal of the processor is directly output from an output register and changes at the rising edge of SysClock.

The signal input to the processor is directly latched to an input register at the rising edge of SysClock.

Figure 12-5. Operation of System Interface Between Registers



12.4.1 Master status and slave status

The system interface is in the master status while the VR5500 is driving the SysAD bus or SysCmd bus. While the external agent is driving these buses, the system interface is in the slave status.

In the master status, the processor always asserts the ValidOut# signal if the SysAD bus and SysCmd bus are valid.

In the slave status, always assert the ValidIn# signal of the external agent if the SysAD bus and SysCmd bus are valid.

The default bus master of the system interface is the processor. The external agent serves as the master of the system interface after the result of external arbitration has been obtained or it has issued a processor read request. The external agent returns the right to control the bus to the processor when the external request has been completed.

The system interface remains in the master status unless either of the following occurs.

- The external agent requests and is granted the right to control the system interface (external arbitration).
- The processor issues a read request (compelled transition to slave status).

These two cases are explained below.

12.4.2 External arbitration

The system interface must be in the slave status when the external agent issues an external request via the system interface. So that the system interface changes its status from master to slave, the processor performs arbitration by using the handshake signals of the system interface, ExtRqst# and Release#, in the following procedure.

<1> The external agent asserts the ExtRqst# signal to transmit a request to issue an external request to the processor.
<2> When the processor is ready to receive the external request, it asserts the Release# signal to change the status of the system interface from master to slave, and releases the system interface.
<3> The system interface returns to the master status as soon as the external request has been issued.

12.4.3 Uncompelled transition to slave status

Uncompelled transition of the system interface to the slave status is performed by the processor, and the system interface changes its status from master to slave when a processor read request is held pending. The Release# signal is automatically asserted when a read request is issued. Uncompelled transition to the slave status takes place in the cycle next to that of the processor read request.

If an external request is issued after uncompelled transition to the slave status, the system interface returns to the master status. If there is a pending processor read request or if the external agent issues another external request, the processor asserts the Release# signal for one cycle, and puts the system interface in the uncompelled slaved status.

The external agent should confirm that the processor has put the system interface in the uncompelled slave status, and start driving the SysCmd and SysAD buses. While the system interface is in the slave status, the external agent can start an external request without arbitrating the system interface, i.e., without asserting the ExtRqst# signal.

If the ExtRqst# signal is active when the external request is completed, the system interface automatically returns to the master status.

12.4.4 Processor requests and external requests

There are two types of requests: processor requests and external requests.

When a system event occurs, the processor issues a request via the system interface and accesses the external resources needed to process the event. Accordingly, the system interface should be connected to the external agent that is used to control access to system resources. To request access to the processor's internal resources, the external agent issues an external request.

Processor requests include the following.

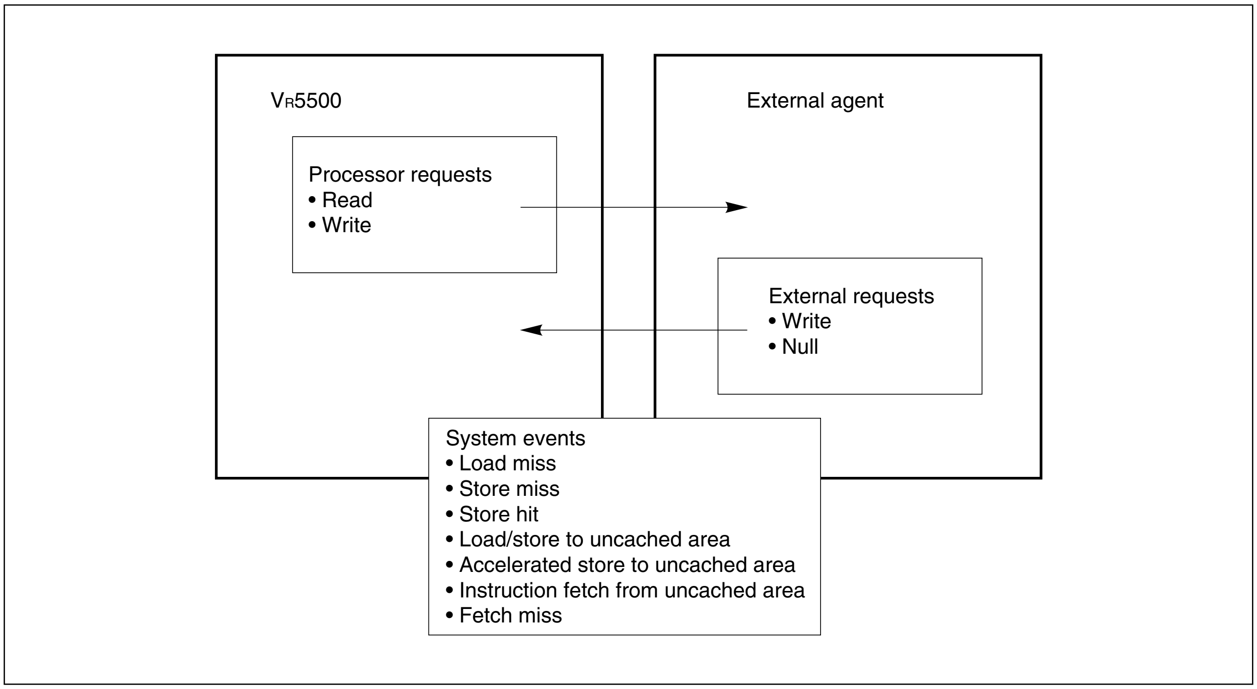
- Read request: Supplies the read address to the external agent
- Write request: Supplies the write address and either single data or block data to the external agent

External requests include the following.

- Write request: Supplies an address and word data to be written to the processor resources
- Null request: Returns the system interface to the master status without affecting the processor

These system events and requests are illustrated in Figure 12-6 below.

Figure 12-6. Requests and System Events



VR5500

Processor requests
- Read
- Write

External agent

External requests
- Write
- Null

System events
- Load miss
- Store miss
- Store hit
- Load/store to uncached area
- Accelerated store to uncached area
- Instruction fetch from uncached area
- Fetch miss

12.5 Processor Requests

A processor request is a request for access to external resources via the system interface. Processor requests include read requests and write requests.

(1) Summary of requests

A read request is a request for data of a block, a doubleword, an unaligned doubleword, a word, or an unaligned word to be retrieved from the main memory or other system resources.

A write request is a request which provides data of a block, a doubleword, an unaligned doubleword, a word, or an unaligned word to be written to the main memory or other system resources.

(2) Issuing requests

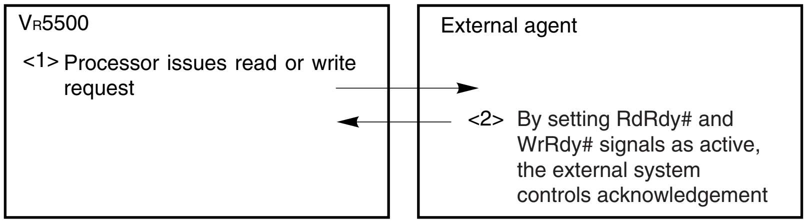
The processor issues requests using a completely sequential method. This means that the processor handles only one pending request at a time. For example, after the processor issues a read request it waits for a read response before issuing the next request (except for the out-of-order return mode). The processor issues write requests only when there are no pending read requests.

(3) Control of requests

The RdRdy# and WrRdy# signals, which are input signals for the processor, are used by the external agent to control the flow of processor requests. The RdRdy# signal controls the flow of processor read requests, and the WrRdy# signal controls the flow of processor write requests.

Figure 12-7 shows the sequence of processor request cycles.

Figure 12-7. Flow of Processor Requests


VR5500
<1> Processor issues read or write request
External agent
<2> By setting RdRdy# and WrRdy# signals as active, the external system controls acknowledgement

12.5.1 Processor read request

Once the processor has issued a read request, the external agent should access the specified resource and return the request data.

A processor read request can be separated from the response data of the external agent. In other words, the external agent can start an unrelated external request before returning response data in response to a processor read request. A processor read request ends when the last word of the response data has been received from the external agent.

The response data's data identifier may indicate whether or not any errors exist in the response data. This enables the processor to generate a bus error exception.

In the VR5500, the external agent must be able to receive a new processor read request at any time if the following condition is satisfied.

- The RdRdy# signal is active at least two cycles before issuance of the address cycle.

In the normal mode, the external agent must be able to receive a new processor read request at any time if the following condition is satisfied.

- There is currently no pending processor read request.

In the out-of-order return mode, up to five read requests can be held pending.

12.5.2 Processor write request

Once the processor has issued a write request, the specified resource is accessed and the specified data is written.

A processor write request ends when the last word of the data has been sent to the external agent.

The write requests of the VR5500 support VR4000-compatible, write re-issuance, and pipeline write timing modes.

The external agent must be able to receive a new processor write request at any time if the following two conditions are satisfied.

- There is currently no pending processor read request.
- The WrRdy# signal is active at least two cycles before issuance of the address cycle and conforms to the requirements of the timing mode set by the Config register.

In the out-of-order return mode, a write request may be issued after a read request.

12.6 External Requests

External requests include write requests and null requests.

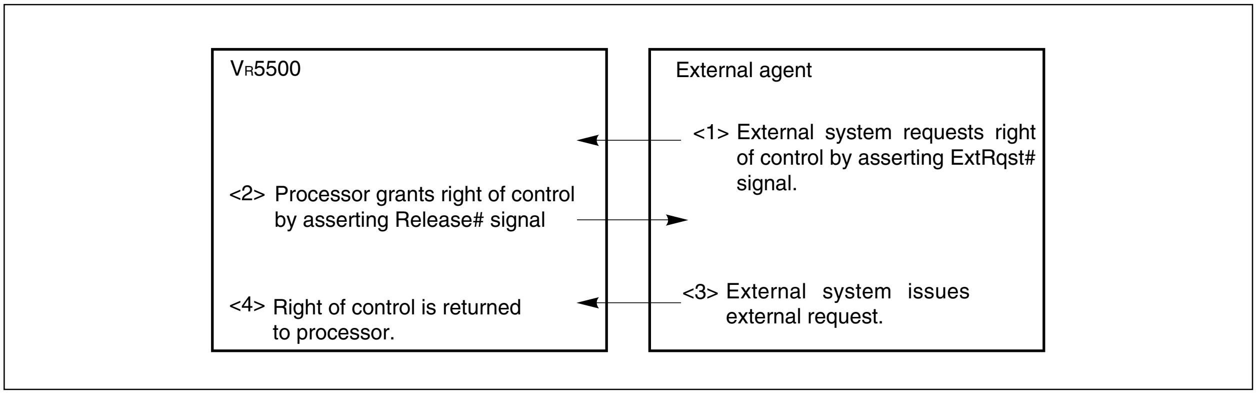
(1) Outline of request

A write request supplies data to be written to the internal resources (interrupt register) of the processor. A null request returns the system interface to the master status without affecting the processor.

(2) Controlling requests

As shown in Figure 12-8, the processor controls the flow of external requests via the arbitration signals ExtRqst# and Release#. The external agent cannot issue an external request unless it is granted the right to control the system interface. The external agent acquires the right to control the system interface by asserting the ExtRsqt# signal and waiting until the processor asserts the Release# signal for the duration of 1 cycle. When the external agent issues an external request, the right to control the system interface is returned to the processor.

Figure 12-8. Flow of External Request


VR5500
External agent
<1> External system requests right of control by asserting ExtRqst# signal.
<2> Processor grants right of control by asserting Release# signal
<3> External system issues external request.
<4> Right of control is returned to processor.

The right to control the system interface is always returned to the processor when the ValidIn# signal has been asserted after an external request was issued. The processor does not acknowledge the subsequent external requests until it completes the current request.

(3) Issuing request

If there is no pending processor request, the processor determines whether it receives an external request or issues a new processor request, depending on its internal status. The processor can issue a new processor request even while the external agent is requesting access to the system interface.

The external agent asserts the ExtRqst# signal to indicate that it wants to start an external request. In response, the processor asserts the Release# signal to release the right to control the system interface. The processor can acknowledge an external request in the following cases.

- When the processor has completed the processor request under execution
- When the ExtRqst# signal is input to the processor one or more cycles before the RdRdy#/WrRdy# signal is asserted while the processor is waiting for assertion of the RdRdy#/WrRdy signal to issue a processor read/write request
- When the processor puts the system interface in the uncompelled slave status and waits for a response to a read request (the external agent can issue an external request before supplying the read response data)

12.6.1 External write request

When the external agent issues a write request, it accesses a specified external resource and writes data to it. The external write request is completed when word data has been transferred to the processor.

The only resource of the processor that can be accessed by an external write request is the Interrupt register.

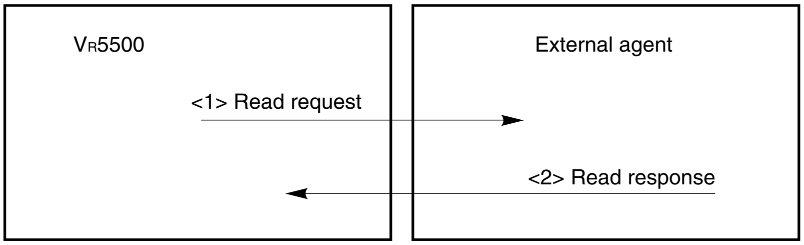
12.6.2 Read response

A read response is used by the external agent to return data in response to a processor read request.

Unlike the other external requests, a read response does not execute system interface arbitration (requesting the right to control the system interface by using the ExtRqst# signal). Therefore, a read response is treated as something different from an external request.

The data identifier of response data can also indicate that the response data contains an error, so that the processor can generate a bus error exception.

Figure 12-9. Read Response



12.7 Event Processing

This section explains the following system events.

- Load miss
- Store miss
- Store hit
- Load/store in uncached area
- Accelerated store in uncached area
- Instruction fetch from uncached area
- Fetch miss

12.7.1 Load miss

If the processor misses the data cache when loading data, it issues a read request to obtain a cache line. The external agent returns data as a read response.

If the cache data to be replaced is dirty, the processor writes back this data to memory. After writing back the data, the processor requests the external agent for clean data, and performs a write operation to the cache.

The operation when a load miss occurs is shown in Table 12-2.

Table 12-2. Operation in Case of Load Miss

Page Attribute	Status of Data Cache Line to Be Replaced	
	Clean/Invalid	Dirty
Cache	BR	BR/BW

BR: Processor block read request
BW: Processor block write request

If it is necessary to write back the current cache line, the processor issues a block write request to save the dirty cache line to memory.

12.7.2 Store miss

If a processor store miss occurs in the cache, the processor requests the external agent for the cache line that holds the target store location.

Table 12-3 shows the operation in case of a store miss.

Table 12-3. Operation in Case of Store Miss

Page Attribute	Status of Data Cache Line to Be Replaced	
	Clean/Invalid	Dirty
Writeback	BR	BR/BW
Write through	BR/W	–

BR: Processor block read request

BW: Processor block write request

W: Processor non-block write request

The processor issues a block read request to the cache line that holds the data element to be loaded, and waits until the external agent supplies read data in response to this read request. If it is necessary to write back the current cache line, the processor issues a request to write the current cache line. If the page attribute is write through, the processor issues a non-block write request.

12.7.3 Store hit

The operation in the system bus is determined by whether the cache line in question is writeback or write through. If the line uses the writeback policy, a processor request is not generated by a store hit. If the line uses the write-through policy, a non-block write request of store data is generated by a store hit.

12.7.4 Load/store in uncached area

When the processor executes loading from an uncached area, it issues a read request for a doubleword, an unaligned doubleword, a word, or an unaligned word. If the processor executes storing in an uncached area, it issues a write request for a doubleword, an unaligned doubleword, a word, or an unaligned word. All the write requests by the processor are buffered in a 4-stage write transaction buffer, and output to the system interface. Because this buffer is a FIFO, if the buffer has an entry when a read request is issued, processing of the read request is started after the buffer has become completely empty.

12.7.5 Accelerated store in uncached area

An accelerated operation to an uncached area is used to access a page with an uncached accelerated cache algorithm. When the processor executes an accelerated store operation to an uncached area, it can issue a block write request or a write request for one or more doublewords, an unaligned doubleword, a word, or an unaligned word. All the write requests by the processor are buffered in a 4-stage write transaction buffer and output to the system interface. Because this buffer is a FIFO, if the buffer has an entry when a read request is issued, processing of the read request is started after the buffer has become completely empty.

By an accelerated operation to an uncached area, several sequential uncached word/doubleword accesses can be combined into one 32-byte block write operation that can be processed by one external SysAD bus transaction. When organizing a system, utmost care must be exercised in locating data that is used to access an uncached accelerated page, so that this transaction is effectively performed.

An accelerated write operation to an uncached area is buffered in the write transaction buffer on a FIFO basis, in the same way as the other transactions. If the data used for an accelerated write operation on an uncached area is located in accordance with the following rules, however, two or more consecutive transactions are combined on a FIFO basis and processed as a 4-doubleword access.

- If the first target of the accelerated operation to the uncached area is located at a 32-byte boundary
- If all the accelerated operations to the uncached area to be processed are word or doubleword accesses
- If the target of the word or doubleword access to be processed is located at a word boundary or doubleword boundary
- In the case of word access, if the targets are located consecutively at a doubleword boundary
- If the address value is incremented sequentially

A write transaction to an uncached area that is not in compliance with these rules is not treated as an accelerated operation. If the transactions for an accelerated operation include a transaction that does not comply with the above rules, all the transactions are processed as an ordinary uncached word/doubleword access.

An accelerated operation to an uncached area is aborted when the processor enters the debug mode. In the debug mode, the contents of the write transaction buffer are cleared. If an exception occurs, the accelerated operation to the uncached area is also aborted.

12.7.6 Instruction fetch from uncached area

The processor issues a word read to fetch an instruction in an uncached area. Therefore, the system ROM address space that is accessed while booting of the processor is being resumed must support an aligned 32-bit read operation.

12.7.7 Fetch miss

If a miss occurs in the instruction cache while an instruction is being fetched, the processor issues a read request to obtain a cache line. The external agent returns data as a read response.

12.8 Error Check Function

12.8.1 Parity error check

The VR5500 performs error detection only, using an even parity.

Parity error detection is the most simple error detection method. By suffixing 1 bit called a parity bit to the end of data, an error of 1 bit can be detected. However, the error cannot be corrected.

Parity comes in the following two types.

- Odd parity is used to append a bit of 1 to data when the number of 1s in the data is even, making the total number of 1s, including that of the parity bit, odd.
- Even parity is used to append a bit of 1 to data when the number of 1s in the data is odd, making the total number of 1s, including that of the parity bit, even.

Here is an example of odd parity and even parity.

Data(3:0)	Odd Parity Bit	Even Parity Bit
0010	0	1

In this example, only one bit that is 1, Data1, is in Data(3:0).

- Even parity sets the parity bit to 1. As a result, the number of bits that are 1 is two (even).
- Odd parity sets the parity bit to 0. As a result, the number of bits that are 1 remains odd (only the one bit of Data1).

Here is an example of odd parity and even parity for various data values.

Data(3:0)	Odd Parity Bit	Even Parity Bit
0110	1	0
0000	1	0
1111	1	0
1101	0	1

Parity can detect an error of 1 bit but cannot identify the bit that has the error. For example, if a value 00011 is received as odd parity, this data has an error because the last bit is the parity bit and the number of 1s, which should be odd, is even. However, which bit has the error is unknown.

12.8.2 Error check operation

The processor uses parity to check the accuracy of data when it transfers data between the system interface and cache.

(1) System interface bus

The processor generates an accurate check bit for the data of a word or an unaligned word that is to be transferred to the system interface. It does not change the data check bit of the cache and directly passes it to the system interface because only the accuracy of the data is to be checked.

The processor does not check the data of an external write operation it receives from the system interface. The processor can also be set to not check the data of a read response it received from the system interface by setting the SysCmd4 bit of a data identifier.

The processor does not check an address it has received from the system interface, and does not generate a check bit for the address to be transferred to the system interface.

The VR5500 does not have a circuit that corrects data. If an error is detected in accordance with the data check bit, a cache error exception occurs. Perform error processing by software.

(2) System interface command bus

The VR5500 does not have a function to check the data of the system interface command bus.

(3) Outline of error check operation

Tables 12-4 and 12-5 outline the error check operation.

Table 12-4. Error Check for Internal Transaction

Transaction Bus	Uncached Load	Uncached Store	Cache Load from System Interface	System Interface Write from Cache	CACHE Instruction
Processor data	From system	Not checked	Not changed, from system interface	Checked, and trap occurs in case of error	Checked when cache is written back, and trap occurs in case of error
System address, command, check bit during transfer	Not generated	Not generated	Not generated	Not generated	Not generated
System address, command, check bit during reception	Not checked	Not checked	Not checked	Not checked	Not checked
System interface data	Checked, and trap occurs in case of error	From processor	Specified word is checked, and trap occurs in case of error	From cache	From cache
System interface data check bit	Checked, and trap occurs in case of error	Generated	Specified word is checked, and trap occurs in case of error	From cache	From cache

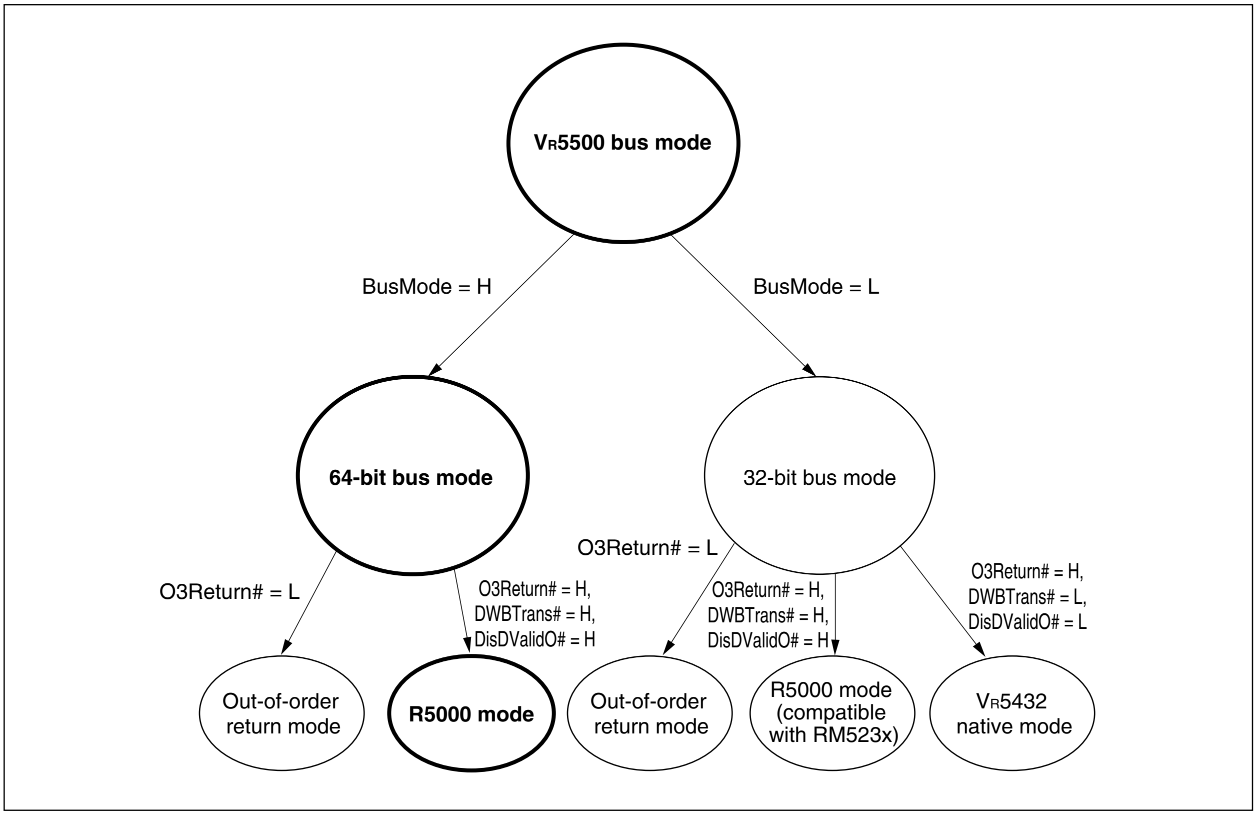
Table 12-5. Error Check for External Transaction

Transaction Bus	External Write
Processor data	Disabled
System address, command, check bit during transfer	Disabled
System address, command, check bit during reception	Not checked
System interface data	Not checked
System interface data check bit	Not checked

CHAPTER 13 SYSTEM INTERFACE (64-BIT BUS MODE)

This chapter explains the request protocol of the system interface in the 64-bit bus normal mode. The system interface of the VR5500 can be set in the 64-bit bus mode by inputting a high level to the BusMode pin before a power-on reset. It can also be set in the normal mode by inputting a high level to the O3Return# pin before a power-on reset, and in the out-of-order return mode by inputting a low level to the same pin.

The 64-bit bus normal mode is also called the R5000 mode, in which the VR5500 is compatible with the bus protocol of the VR5000 Series. To set this mode, input a high level to the DWBTrans# and DisDValidO# pins before a power-on reset.


VR5500 bus mode
BusMode = H
BusMode = L
64-bit bus mode
32-bit bus mode
O3Return# = L
O3Return# = H, DWBTrans# = H, DisDValidO# = H
O3Return# = L
O3Return# = H, DWBTrans# = H, DisDValidO# = H
O3Return# = H, DWBTrans# = L, DisDValidO# = L
Out-of-order return mode
R5000 mode
Out-of-order return mode
R5000 mode (compatible with RM523x)
VR5432 native mode

For the protocol in the 32-bit bus normal modes (operation mode compatible with native mode of the VR5432 and the RM523x), refer to **CHAPTER 14 SYSTEM INTERFACE (32-BIT BUS MODE)**. For the protocol in the out-of-order return mode, refer to **CHAPTER 15 SYSTEM INTERFACE (OUT-OF-ORDER RETURN MODE)**.

13.1 Protocol of Processor Requests

This section explains the following two processor request protocols.

- Read
- Write

13.1.1 Processor read request protocol

The following sequence explains the protocol of a processor read request for a doubleword, unaligned doubleword, word, and unaligned word (the numbers correspond to the numbers in Figure 13-1).

<1> The external agent makes the RdRdy# signal is low and is ready to acknowledge a read request.

<2> When the system interface is in the master status, the processor issues a processor read request by driving a read command onto the SysCmd bus and a read address onto the SysAD bus. A physical address is driven onto SysAD(35:0). All the other bits are driven to 0.

<3> At the same time, the processor asserts the ValidOut# signal for the duration of 1 cycle. This signal indicates that valid data is on the SysCmd and SysAD buses.

<4> The processor puts the system interface in the uncompelled slave status. The external agent must wait without asserting the ExtRqst# signal in an attempt to return a read response, until transition of the system interface to the uncompelled slave status is completed.

<5> The processor releases the SysCmd and SysAD buses 1 cycle after the Release# signal has been asserted.

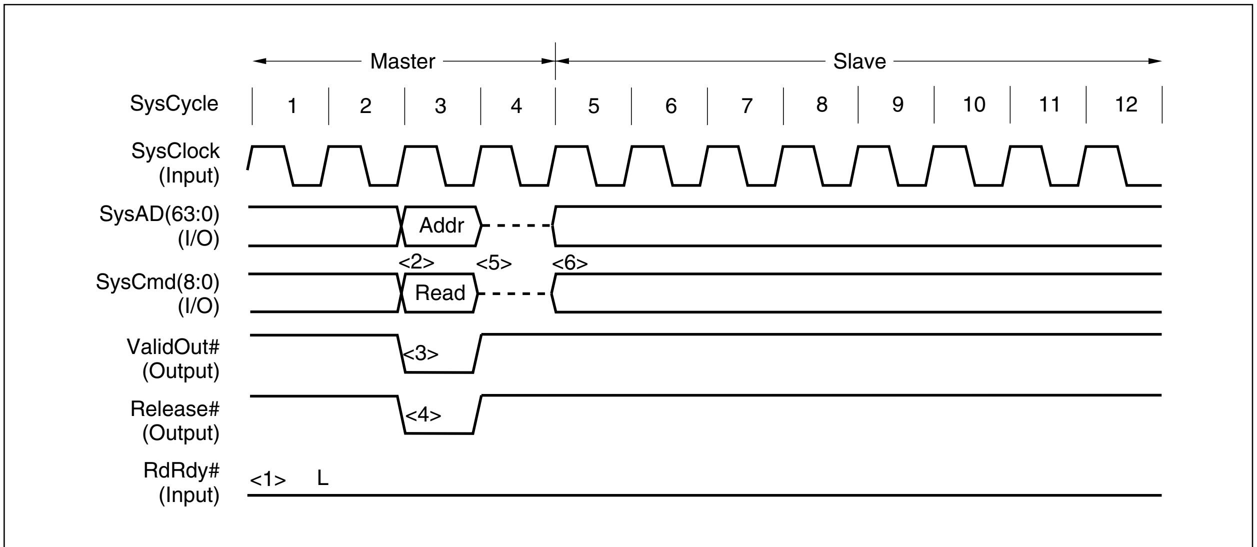
<6> The external agent drives the SysCmd and SysAD buses 2 cycles after the Release# signal has been asserted.

When the system interface has been put in the slave status, the external agent can return the requested data by using a read response. The read response can also return an indication that an error has occurred in the data if the requested data could not be searched correctly, as well as the requested data. If the returned data contains an error, the processor generates a bus error exception.

Figure 13-1 shows the processor read request, and uncompelled transition to the slave status that takes place when the read request is issued.

The timing of the SysADC bus is the same as that of the SysAD bus.

Figure 13-1. Processor Read Request



Remark The dotted line indicates high impedance.

After the Release# signal has been asserted (<6> and later in the figure), the processor can acknowledge both a read response (if the read request is pending) and an external request.

13.1.2 Processor write request protocol

The processor write request is issued by using either of the following two protocols.

- A write request for a doubleword, word, or unaligned word uses a single write request protocol.
- Cache block write and uncached accelerated write uses a block write request protocol.

A processor write request is issued when the system interface is in the master status.

Figure 13-2 shows the processor single write request cycle and Figure 13-3 shows the processor block write request cycle (the numbers in the explanation below correspond to the numbers in the figures).

<1> The external agent makes the WrRdy# signal low and is ready to acknowledge a write request.

<2> The processor issues a processor write request by driving a write command onto the SysCmd bus and a write address onto the SysAD bus. A physical address is driven onto SysAD(35:0). All the other bits are driven to 0.

<3> The processor asserts the ValidOut# signal.

<4> The processor drives a data identifier onto the SysCmd bus and data onto the SysAD bus.

<5> The data identifier corresponding to the data cycle must include an indication of the last data cycle. At the end of the cycle, the ValidOut# signal is deasserted.

Remark The timing of the SysADC bus is the same as that of the SysAD bus.

Figure 13-2. Processor Non-Block Write Request Protocol

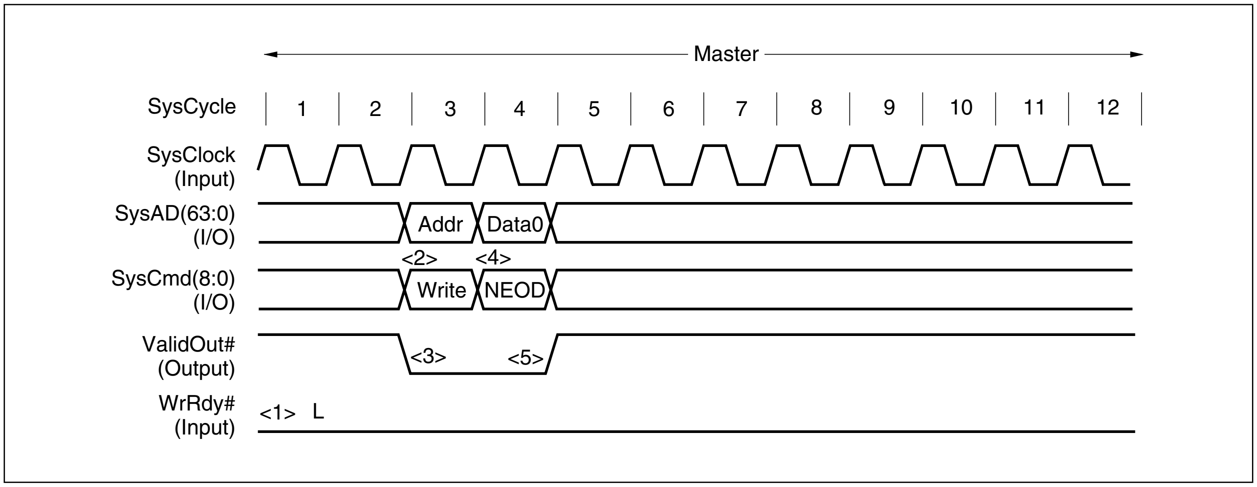
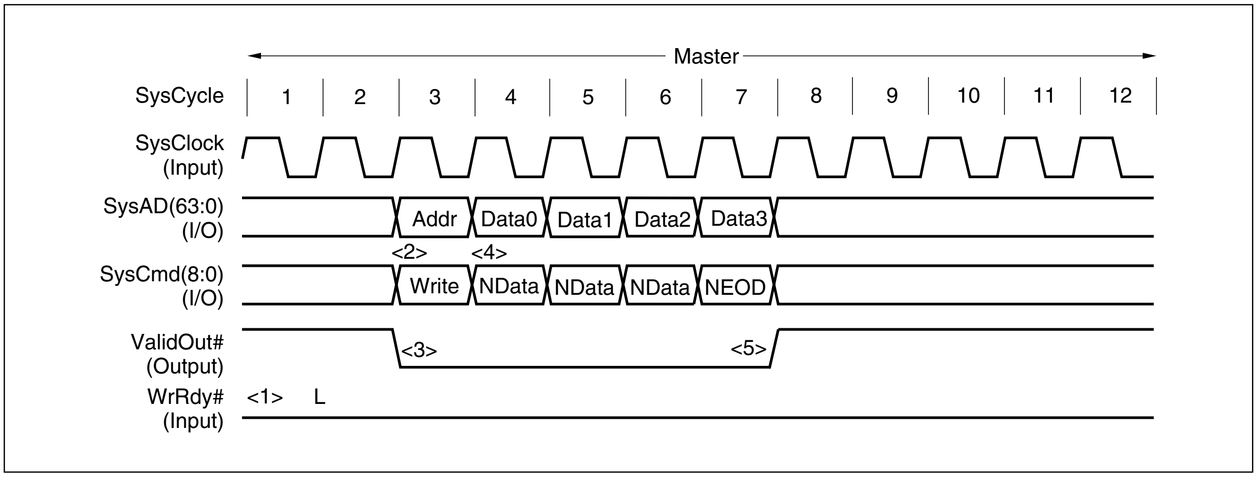


Figure 13-3. Processor Block Write Request



13.1.3 Control of processor request flow

The external agent uses the RdRdy# signal to control the flow of processor read requests.

Figure 13-4 shows the control of the read request flow (the numbers in the explanation below correspond to the numbers in the figure).

<1> The processor samples the RdRdy# signal and determines whether the external agent can acknowledge a read request.

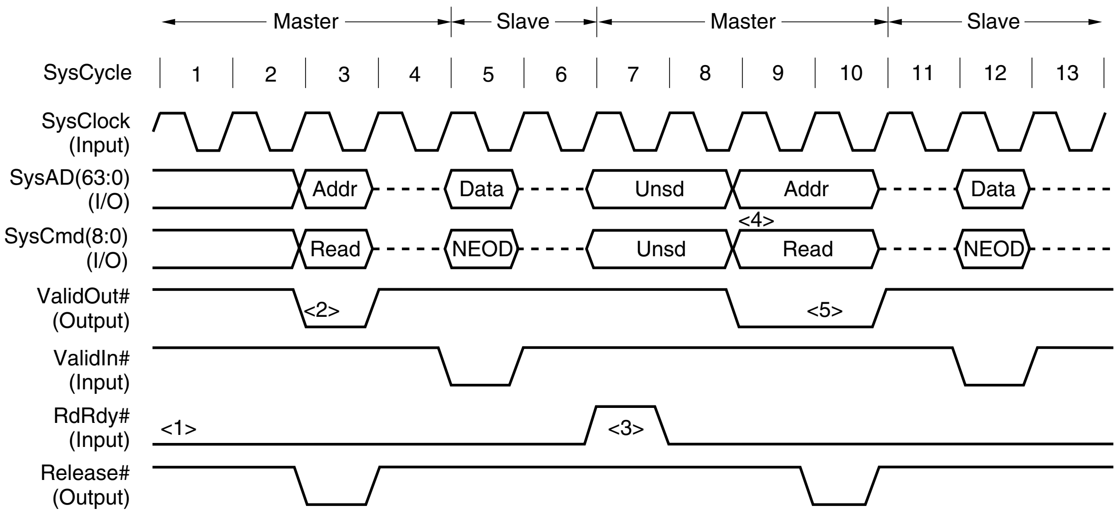
<2> The processor issues a read request to the external agent.

<3> The external agent deasserts the RdRdy# signal. This signal indicates that no more read requests can be acknowledged.

<4> Because the RdRdy# signal is deasserted two cycles before, issuance of the read request is stalled.

<5> The read request is issued again to the external agent.

Figure 13-4. Control of Processor Request Flow



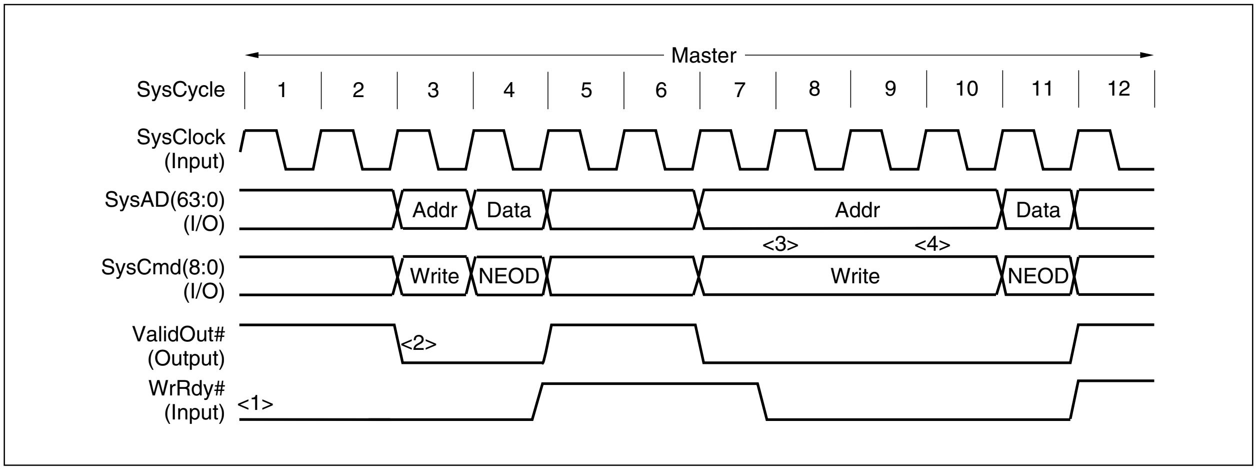
Remark The dotted line indicates high impedance.

Figure 13-5 shows an example in which two processor write requests are issued but issuance of the second request is delayed because of the condition of the WrRdy# signal (the numbers in the explanation below correspond to the numbers in the figure).

<1> The external agent asserts the WrRdy# signal to indicate that it is ready to acknowledge a write request.
<2> The processor asserts the ValidOut# signal, and drives a write command onto the SysCmd bus and a write address onto the SysAD bus.
<3> The second write request is delayed until the WrRdy# signal is asserted again.
<4> If the WrRdy# signal is active two cycles before, an address cycle is issued in response to the processor write request. This completes the issuance of the write request.

Remark The timing of the SysADC bus is the same as that of the SysAD bus.

Figure 13-5. Timing When Second Processor Write Request Is Delayed



13.1.4 Timing mode of processor request

The VR5500 has three timing modes: VR4000-compatible mode, write re-issuance mode, and pipeline write mode.

- VR4000-compatible mode
  If single write requests are successively issued, the processor inserts two unused cycles after the data cycle so that an address cycle is issued once every 4 system cycles.

- Write re-issuance mode
  If the WrRdy# signal is deasserted in the address cycle of a write request, that request is discarded, but the processor issues the same write request again.

- Pipeline write mode
  Even if the WrRdy# signal is deasserted in the address cycle of a write request, the processor assumes that it has issued that request.

(1) VR4000-compatible mode

With the VR5500 processor interface, the WrRdy# signal must be asserted two system clocks before issuance of a write cycle. If the WrRdy# signal is deasserted immediately after the external agent has received a write request that fills the buffer, the subsequent write requests are kept waiting for the duration of 4 system cycles. The processor inserts at least two unused system cycles after a write address/data pair, giving the external agent the time to keep the next write request waiting.

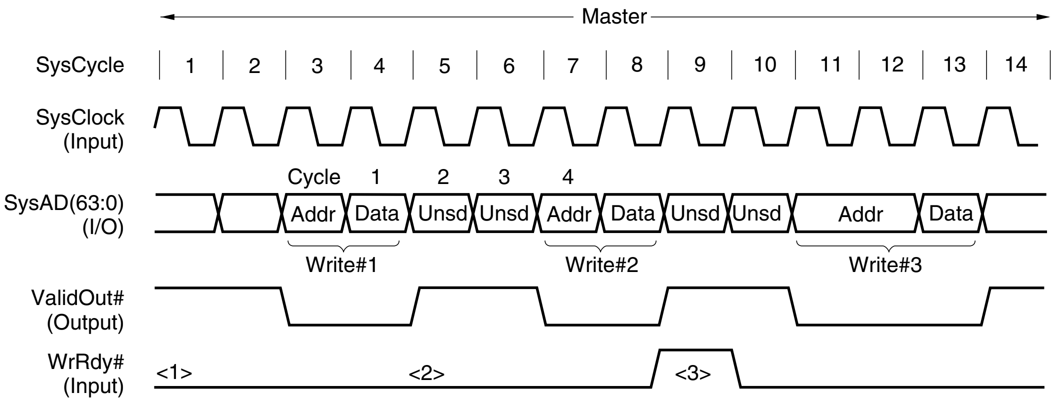
Figure 13-6 shows a back-to-back write cycle in the VR4000-compatible mode (the numbers in the explanation below correspond to the numbers in the figure).

<1> The external agent asserts the WrRdy# signal to indicate that it is ready to issue a write cycle.

<2> The WrRdy# signal remains active. This indicates that the external agent can acknowledge another write request.

<3> The WrRdy# signal is deasserted. This indicates that the external agent cannot acknowledge any more write requests, and that issuance of the next write request is stalled.

Figure 13-6. Timing of VR4000-Compatible Back-to-Back Write Cycle



(2) Write re-issuance mode

Figure 13-7 shows the write re-issuance protocol (the numbers in the explanation below correspond to the numbers in the figure).

A write request is issued when the WrRdy# signal is asserted two cycles before the address cycle and in the address cycle.

<1> The external agent asserts the WrRdy# signal to indicate that it is ready to acknowledge a write request.

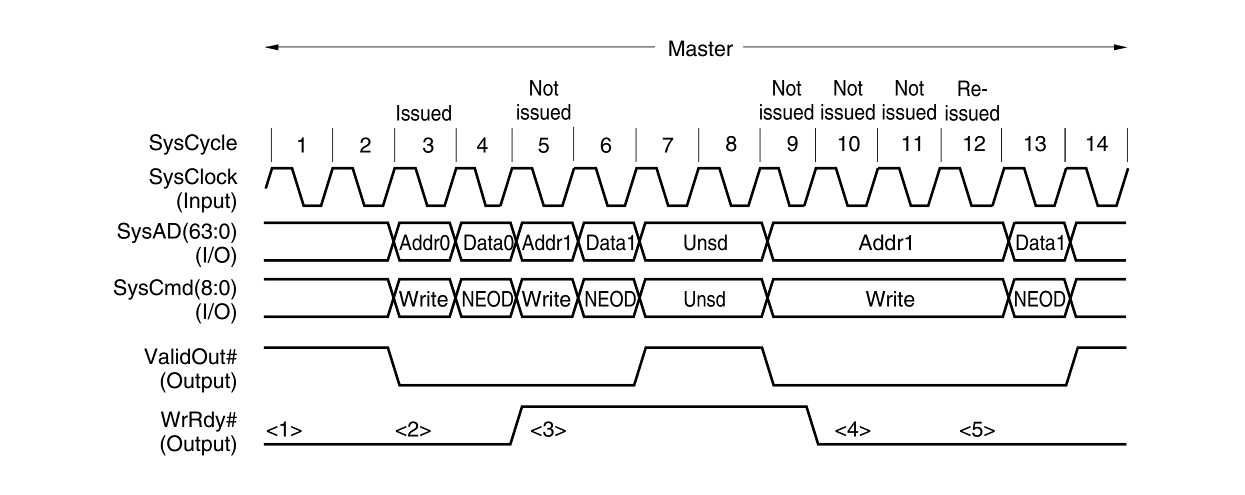
<2> The WrRdy# signal remains active even when the write request has been issued. This indicates that the external agent can acknowledge another write request.

<3> The WrRdy# signal is deasserted in the address cycle. This write cycle is aborted.

<4> The external agent asserts the WrRdy# signal, indicating that it is ready to acknowledge a write request. In response, the write request aborted in <3> is re-issued.

<5> Even if a write request is issued, the WrRdy# signal remains active. This indicates that the external agent can acknowledge another write request.

Figure 13-7. Write Re-Issuance



(3) Pipeline write mode

Figure 13-8 shows the pipeline write protocol (the numbers in the explanation below correspond to the numbers in the figure). If the WrRdy# signal is issued two cycles before the address cycle, a write request is issued. After the WrRdy# signal has been deasserted, the external agent must acknowledge one more write request.

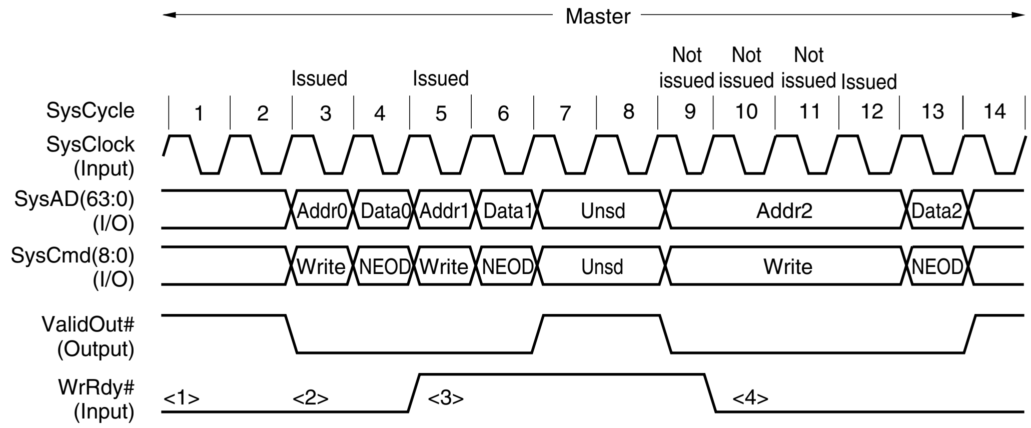
<1> The external agent asserts the WrRdy# signal to indicate that it is ready to acknowledge a write request.

<2> Even when the write request has been issued, the WrRdy# signal remains active. This indicates that the external agent can acknowledge one more write request.

<3> The WrRdy# signal is deasserted. This indicates that the external agent can acknowledge no more write requests. However, this write request is acknowledged.

<4> The external agent asserts the WrRdy# signal, indicating that it can acknowledge a write request.

Figure 13-8. Pipeline Write



13.2 Protocol of External Request

An external request can be issued only when the system interface is in the slave status. Arbitration that changes the status of the system interface from master to slave is realized by using the handshake signals of the system interface (ExtRqst# and Release#).

This section explains the following external request protocols, as well as the arbitration protocol.

- Null
- Write
- Read response

13.2.1 External arbitration protocol

To issue an external request, assert the ExtRqst# signal to arbitrate the system interface. Then wait until the processor asserts the Release# signal and releases the system interface to the slave status. When the system interface is already in the slave status, i.e., when the processor previously executed an uncompelled transition of the system interface to the slave status, the external agent can immediately start issuing an external request.

After issuing an external request, the external agent must return the right to control the system interface to the processor.

If the external agent does not have any more external requests that must be processed, it must deassert the ExtRqst# signal two cycles after the Release# signal was asserted. To issue two or more requests in a row, the ExtRqst# signal must be kept active until the last request cycle. If the last request cycle lasts for two cycles or more after the Release# signal was asserted, deassert the ExtRqst# signal.

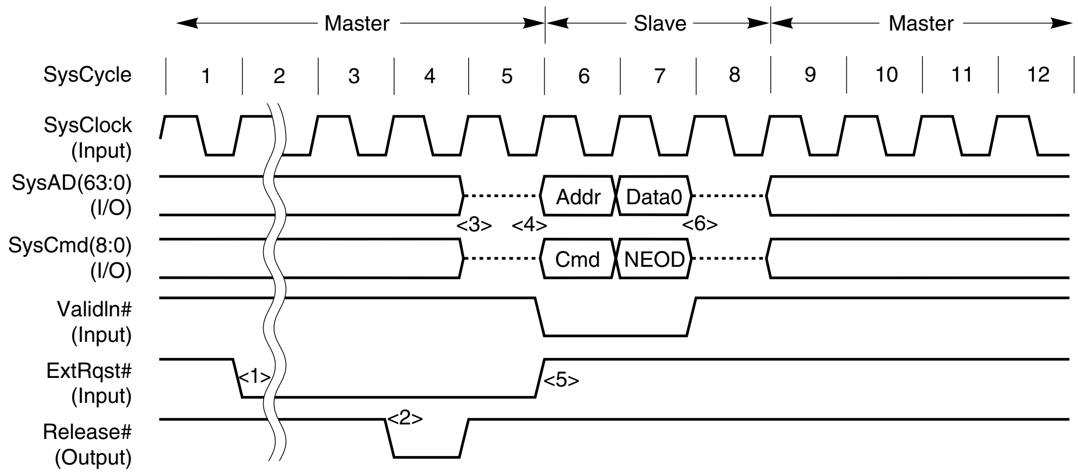
While the ExtRqst# signal is active, the processor continues processing the external request. However, the processor cannot release the system interface to process the next external request until processing of the current request is finished. While the ExtRqst# signal is active, two or more successive external requests cannot be interrupted by a processor request.

Figure 13-9 shows the arbitration protocol of an external request issued by the external agent. The following sequence explains the arbitration protocol (the numbers in the explanation below correspond to the numbers in the figure).

<1> The external agent continues asserting the ExtRqst# signal to issue an external request.
<2> The processor asserts the Release# signal for 1 cycle when it is ready to process the external request.
<3> The processor makes the SysAD and SysCmd buses go into a high-impedance state.
<4> The external agent must drive the SysAD and SysCmd buses at least two cycles after the Release# signal was asserted.
<5> The external agent must deassert the ExtRqst# signal two cycles after the Release# signal was asserted, except when it executes another external request.
<6> The external agent must make the SysAD and SysCmd buses go into a high-impedance state on completion of the external request.

Remark The timing of the SysADC bus is the same as that of the SysAD bus.

Figure 13-9. External Request Arbitration Protocol



Remark The dotted line indicates high impedance.

13.2.2 External null request protocol

The processor supports an external null request. This request only returns the system interface from the slave status to the master status, and does not have any other influence on the processor.

Figure 13-10 shows the timing of the external null request (the numbers in the explanation below correspond to the numbers in the figure).

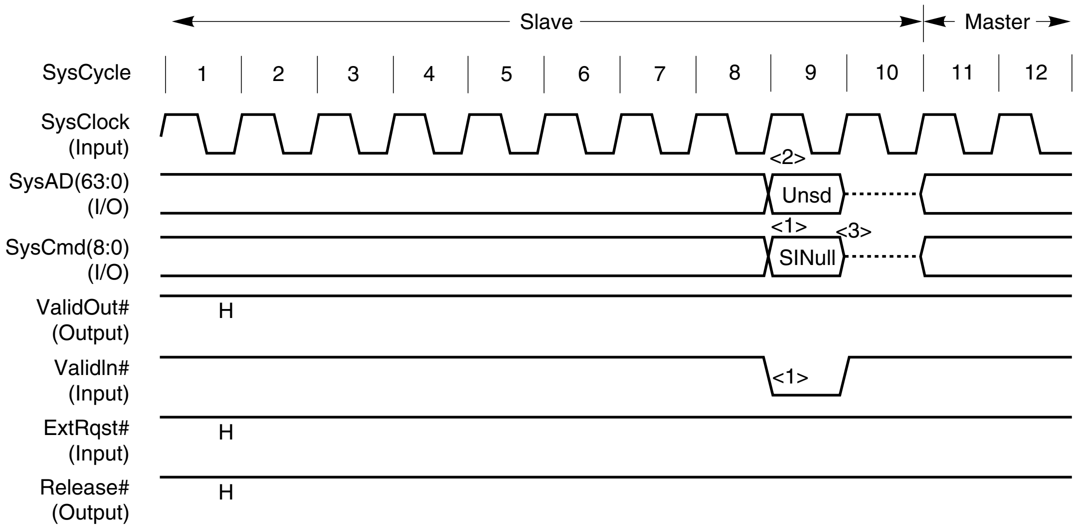
<1> The external agent drives an external null request command onto the SysCmd bus and asserts the ValidIn# signal for one cycle. This returns the right to control the system interface to the processor.

<2> The SysAD bus is not used in the address cycle corresponding to the external null request (the bus does not hold valid data).

<3> When the address cycle is issued, the null request is completed.

The external null request returns the system interface to the master status when the external agent has released the SysCmd and SysAD buses.

Figure 13-10. External Null Request Protocol



Remark The dotted line indicates high impedance.

13.2.3 External write request protocol

The external write request performs an operation close to the processor single write request, except that it asserts the ValidIn# signal, instead of the ValidOut# signal.

Figure 13-11 shows the timing of the external write request (the numbers in the explanation below correspond to the numbers in the figure).

<1> The external agent asserts the ExtRqst# signal to arbitrate the system interface.

<2> The processor asserts the Release# signal to release the system interface to the slave status.

<3> The external agent asserts the ValidIn# signal and drives a write command onto the SysCmd bus and a write address onto the SysAD bus.

<4> The external agent asserts the ValidIn# signal and drives a data identifier onto the SysCmd bus and data onto the SysAD bus.

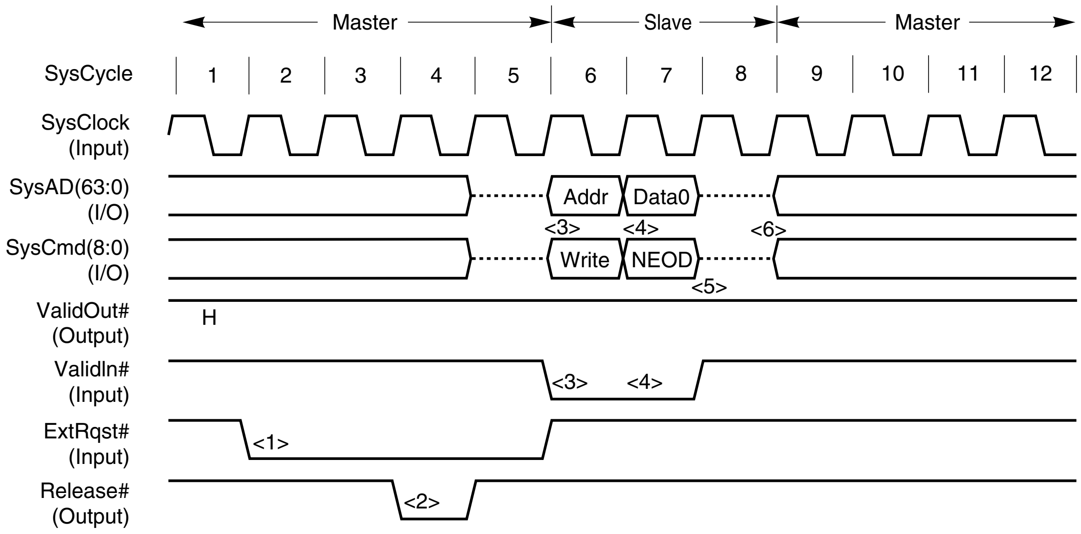
<5> The data identifier corresponding to the data cycle must contain an indication of the last data cycle.

<6> When the data cycle is issued, the write request is completed. The external agent makes the SysCmd and SysAD buses go into a high-impedance state, and returns the system interface to the master status.

Remark The timing of the SysADC bus is the same as that of the SysAD bus.

The external write request can only write word data to the processor. If a data element other than a word is specified for the external write request, the operation of the processor is undefined.

Figure 13-11. External Write Request Protocol



Remark The dotted line indicates high impedance.

13.2.4 Read response protocol

The external agent must return data to the processor by using a read response protocol, in response to a processor read request. The following sequence explains the read response protocol (the numbers in the explanation below correspond to the numbers in Figures 13-12 and 13-13).

<1> The external agent waits until the processor puts the system interface in the uncompelled slave status.

<2> The processor returns data via a single data cycle or a series of data cycles.

<3> When the last data cycle is issued, the read response is completed, and the external agent makes the SysCmd and SysAD buses go into a high-impedance state.

<4> The system interface returns to the master status.

Remark When the read request is issued, the processor always puts the system interface in the uncompelled slave status.

<5> The data identifier of the data cycle must indicate that this data is response data.

<6> The data identifier corresponding to the last data cycle must contain an indication of the last data cycle.

If the read response is for a block read request, the response data does not have to identify the initial cache status. The processor automatically allocates the cache to the clean status.

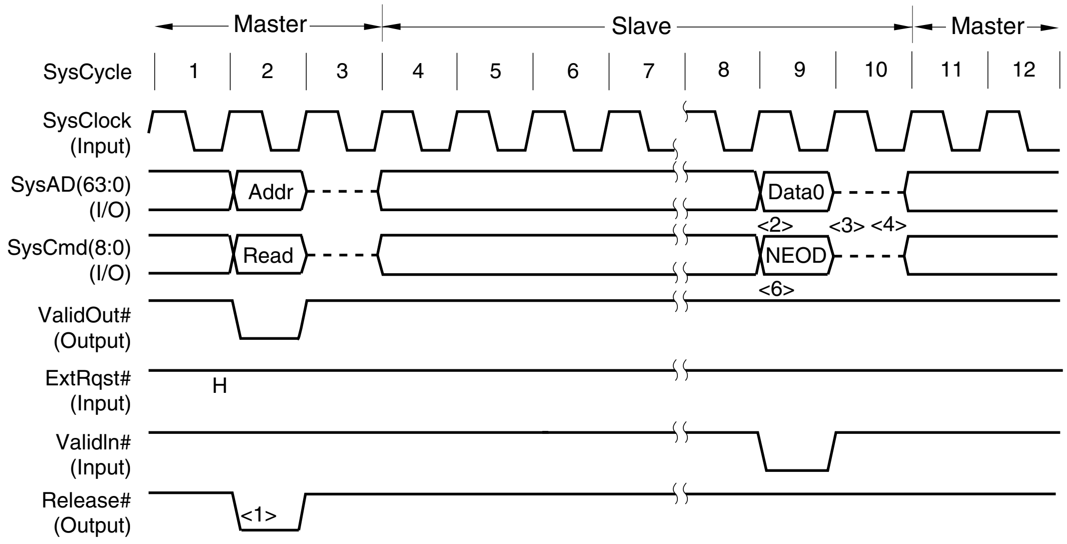
The data identifier corresponding to the data cycle can indicate that the data transferred in that cycle has an error. Even if data may have an error, however, the external agent must return a data block of the correct size. The processor checks the error bit of only the first doubleword of the block, and ignores the rest of the error bits of that block (refer to **13.2.5 SysADC(7:0) protocol for block read response**).

Only when there is a pending processor read request, read response data is passed to the processor. The operation of the processor is undefined if there is no pending processor read request when a read response is received.

Figure 13-12 shows a processor word request and the word read response that follows. Figure 13-13 shows the read response to a processor block read request when the system interface is already in the slave status.

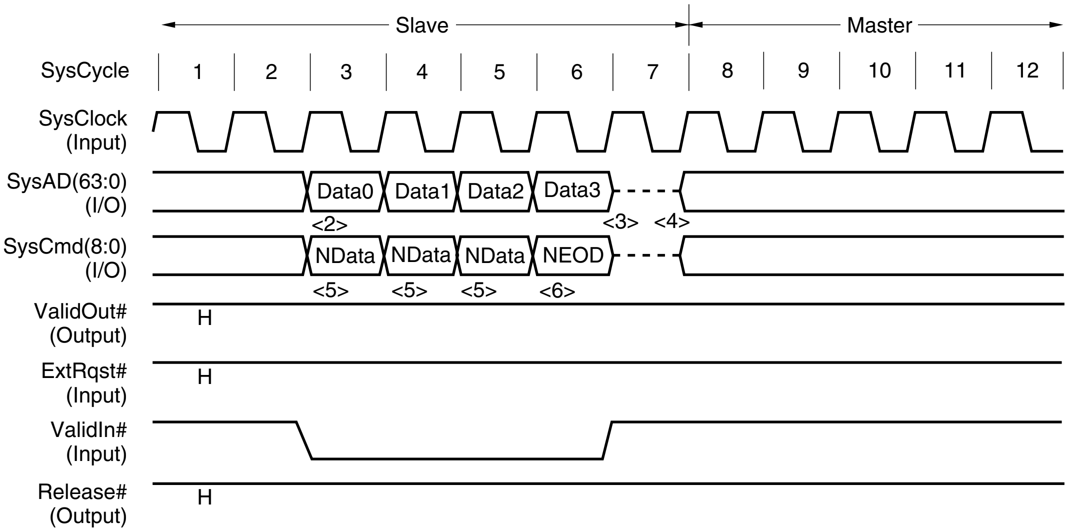
Remark The timing of the SysADC bus is the same as that of the SysAD bus.

Figure 13-12. Protocol of Read Request and Read Response



Remark The dotted line indicates high impedance.

Figure 13-13. Block Read Response in Slave Status



Remark The dotted line indicates high impedance.

13.2.5 SysADC(7:0) protocol for block read response

When a block read response is issued, SysADC(7:0) must be used in compliance with the following rules.

- Only the first doubleword of transfer data is checked. If the data has an error (SysCmd5 = 1), the cache line is invalidated, and a bus error exception occurs in the processor.
- A parity error of the first doubleword is detected when a request is issues, and a cache error exception occurs. At this time, the cache line is in the Invalid status. A parity error of a subsequent doubleword is detected again when that data is used.
- The error bits in three subsequent doublewords of data are ignored. The parity of each doubleword is written to the cache, but is not checked until the data is referenced.
- If a memory error occurs during a block read operation, the SysADC bit must be changed to an illegal parity during a read response operation for all the bytes that are affected by the memory error. However, even if SysCmd5 is set to 1 during data transfer other than the first doubleword, a bus error exception does not occur. If the SysADC bit has been changed to an illegal parity, a cache error exception occurs when any of the remaining three doublewords is referenced.

13.3 Data Flow Control

The system interface supports a data rate of 1 doubleword per cycle.

13.3.1 Data rate control

The external agent can send data to the processor at the maximum data rate of the system interface.

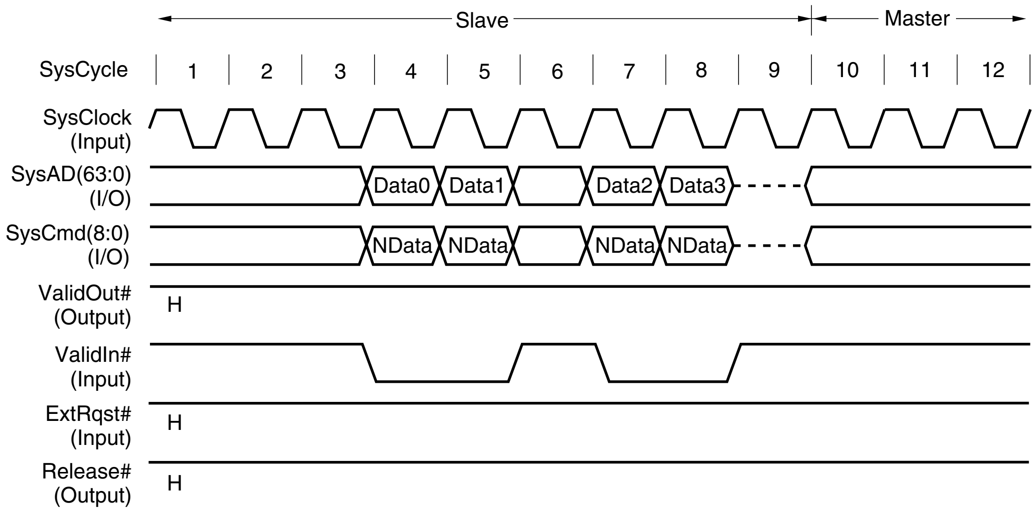
The rate at which data is to be sent to the processor can be controlled on the external agent side. The transfer rate from the external agent is not limited. The external agent asserts the ValidIn# signal in the cycle in which it transfers data.

When the ValidIn# signal has been asserted and as long as a data identifier is on the SysCmd bus, the processor acknowledges the cycle as valid. It then goes on acknowledging data until it receives a data word with NEOD.

The operation of the processor is undefined if data is sent in a pattern of other than 1 cycle for single data, and other than 4 cycles for block data.

Figure 13-14 shows the timing of the read response where the data rate pattern is DDx.

Figure 13-14. Read Response with Data Rate Pattern DDx



Remark The dotted line indicates high impedance.

13.3.2 Block write data transfer pattern

The rate at which the processor transfers block write data to the external agent can be set by the EP bit of the Config register after reset. The data pattern is indicated by characters D and x that indicate the array of data cycle and unused cycle at each data rate. D indicates a data cycle, and x indicates an unused cycle. For example, Dxx data pattern indicates a data rate of 1 doubleword in every 3 cycles.

Table 13-1 shows the maximum data rate that can be set after reset.

Table 13-1. Transfer Data Rate and Data Pattern

Maximum Data Rate	Data Pattern
1 doubleword/1 cycle	DDDD
2 doublewords/3 cycles	DDxDDx
2 doublewords/4 cycles	DDxxDDxx
1 doubleword/2 cycles	DxDxDxDx
2 doublewords/5 cycles	DDxxxDDxxx
2 doublewords/6 cycles	DDxxxxDDxxxx
1 doubleword/3 cycles	DxxDxxDxxDxx
2 doublewords/8 cycles	DDxxxxxxDDxxxxxx
1 doubleword/4 cycles	DxxxDxxxDxxxDxxx

13.3.3 System endianness

The endianness of the system is set by the BigEndian pin after reset. The set endianness is indicated by the BE bit of the Config register.

13.4 Independent Transfer with SysAD Bus

For general applications, the SysAD bus connects the processor and a bidirectional register type transceiver in the external agent between two points. For such applications, only the processor and external agent can be connected to the SysAD bus.

For specific applications, other drivers and receivers are connected to the SysAD bus so that transfer can be performed independently of the processor on the SysAD bus. This is called independent transfer. To execute independent transfer, the external agent must adjust the right to control the SysAD bus by using the arbitration handshake signals and external null request.

The procedure of independent transfer of the SysAD bus is as follows.

<1> The external agent requests the right to control the SysAD bus by asserting the ExtRqst# signal to issue an external request.
<2> The processor releases the system interface to the slave status by asserting the Release# signal.
<3> In this way, the external agent can execute independent transfer on the SysAD bus. The ValidIn# signal must not be asserted during transfer.
<4> When transfer is completed, the external agent releases and returns the system interface to the master status by issuing an external null request.

13.5 System Interface Cycle Time

Because processor requests are restricted by the system interface protocol, the number of request cycles is checked by the protocol. Because external requests have the following two types of wait times, the number of request cycles differs depending on these wait times.

- Standby time until the processor releases the system interface to the slave status in response to an external request (release wait time)
- Response time of the external request that requires a response (external response wait time)

While an external request is being issued, the release wait time differs depending on the status of the system interface. When the external request is detected, the system interface is released to the external agent after the cycle under processing.

The external response time of the VR5500 is kept to the minimum. Data that is written is immediately loaded.

13.6 System Interface Commands and Data Identifiers

A system interface command defines the type and attribute of a system interface request. This definition is indicated in the address cycle of a request.

The system interface data identifier defines the attribute of the data transferred in the system interface data cycle.

This section explains the syntax of the commands and data identifiers of the system interface, i.e., coding in bit units.

Set the reserved bits and reserved area in the commands and data identifiers of the system interface related to external requests to 1.

The reserved bits and reserved area in the commands and data identifiers of the system interface related to processor requests are undefined.

13.6.1 Syntax of commands and data identifiers

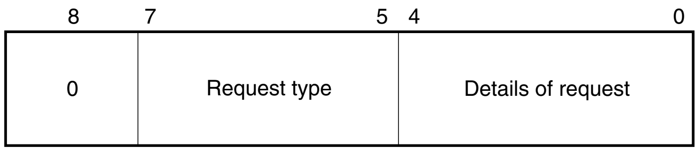
The commands and data identifiers of the system interface are coded in 9-bit units, and transferred from the processor to the external agent, or vice versa, via the SysCmd bus in the address cycle and data cycle.

SysCmd8 (most significant bit) determines whether the current contents of the SysCmd bus are a command (address cycle) or data identifier (data cycle). If they are a command, clear SysCmd8 to 0; if they are a data identifier, set it to 1.

13.6.2 Syntax of command

This section explains the coding of the SysCmd bus when a system interface command is used. Figure 13-15 shows the common code used for all the system interface commands.

Figure 13-15. Bit Definition of System Interface Command



Be sure to clear SysCmd8 to 0 when a system interface command is used.

SysCmd(7:5) define the types of system interface requests such as read, write, and null.

Table 13-2. Code of System Interface Command SysCmd(7:5)

Bit	Contents
SysCmd(7:5)	Command 0: Read request 1: Reserved 2: Write request 3: Null request 4 to 7: Reserved

SysCmd(4:0) are determined according to the type of request. A definition of each request is given below.

(1) Read request

The code of the SysCmd bus related to a read request is shown below.

Figure 13-16 shows the format of the command when a read request is issued.

Tables 13-3 to 13-5 show the code of the read attribute of the SysCmd(4:0) bits related to the read request.

Figure 13-16. Bit Definition of SysCmd Bus During Read Request

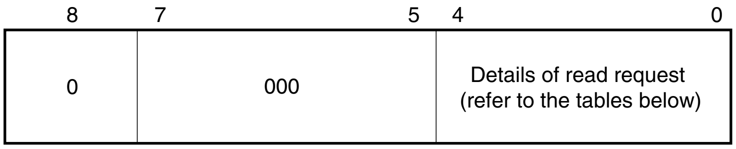


Table 13-3. Code of SysCmd(4:3) During Read Request

Bit	Contents
SysCmd(4:3)	Read attribute 0, 1: Reserved 2: Block read 3: Single read

Table 13-4. Code of SysCmd(2:0) During Block Read Request

Bit	Contents
SysCmd2	Reserved
SysCmd(1:0)	Size of read block 0: Reserved 1: 8 words 2, 3: Reserved

Table 13-5. Code of SysCmd(2:0) During Single Read Request

Bit	Contents
SysCmd(2:0)	Read data size 0: 1 byte is valid (byte). 1: 2 bytes are valid (halfword). 2: 3 bytes are valid. 3: 4 bytes are valid (word). 4: 5 bytes are valid. 5: 6 bytes are valid. 6: 7 bytes are valid. 7: 8 bytes are valid (doubleword).

(2) Write request

The code of the SysCmd bus related to a write request is shown below.

Figure 13-17 shows the format of the command when a write request is issued.

Tables 13-6 to 13-8 show the code of the write attribute of the SysCmd(4:0) bits related to the write request.

Figure 13-17. Bit Definition of SysCmd Bus During Write Request

8	7 5	4 0
0	010	Details of write request (refer to the tables below)

Table 13-6. Code of SysCmd(4:3) During Write Request

Bit	Contents
SysCmd(4:3)	Write attribute 0, 1: Reserved 2: Block write 3: Single write

Table 13-7. Code of SysCmd(2:0) During Block Write Request

Bit	Contents
SysCmd2	Update of cache line 0: Replaced 1: Retained
SysCmd(1:0)	Size of write block 0: Reserved 1: 8 words 2, 3: Reserved

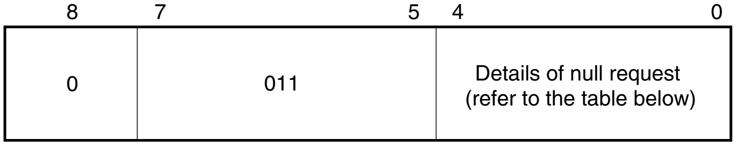
Table 13-8. Code of SysCmd(2:0) During Single Write Request

Bit	Contents
SysCmd(2:0)	Write data size 0: 1 byte is valid (byte). 1: 2 bytes are valid (halfword). 2: 3 bytes are valid. 3: 4 bytes are valid (word). 4: 5 bytes are valid. 5: 6 bytes are valid. 6: 7 bytes are valid. 7: 8 bytes are valid (doubleword).

(3) Null request

Figure 13-18 shows the format of the command when a null request is used.

Figure 13-18. Bit Definition of SysCmd Bus During Null Request



8	7 – 5	4 – 0
0	011	Details of null request (refer to the table below)

Table 13-9 shows the code of the SysCmd(4:3) bits related to the null request.

For the null request, the SysCmd(2:0) bits are reserved.

Table 13-9. Code of SysCmd(4:3) During Null Request

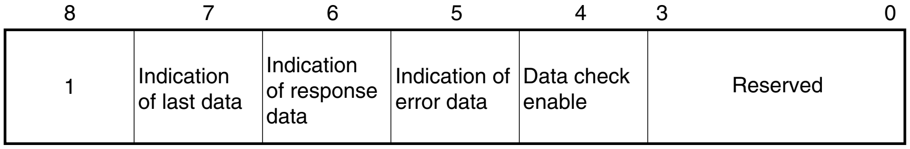
Bit	Contents
SysCmd(4:3)	Null attribute 0: Released 1 to 3: Reserved

13.6.3 Syntax of data identifier

This section explains coding of the SysCmd bus when a system interface data identifier is used.

Figure 13-19 shows the common code used for all system interface data identifiers.

Figure 13-19. Bit Definition of System Interface Data Identifier



8	7	6	5	4	3 – 0
1	Indication of last data	Indication of response data	Indication of error data	Data check enable	Reserved

Be sure to set SysCmd8 of the system interface data identifier to 1.

A definition of the SysCmd(7:0) bits is given below.

SysCmd7: Indicates whether the data element is the last one.

SysCmd6: Indicates whether the data is response data. Response data is returned in response to a read request.

SysCmd5: Indicates whether the data element contains an error. The error indicated in the data cannot be corrected. If this data is returned to the processor, a bus error exception occurs. In the case of a response block, send the entire line to the processor regardless of the degree of error. The processor checks SysCmd5 of the first doubleword of the block response data. The external agent should ignore this bit in a processor data identifier because no error is indicated.

SysCmd4: This bit in an external data identifier indicates whether the data of the data element and check bit are checked. This bit in a processor data identifier is reserved.

SysCmd(3:0): These bits are reserved.

Table 13-10 indicates the codes of SysCmd(7:5) of a processor data identifier, and Table 13-11 shows the codes of SysCmd(7:4) of an external data identifier.

Table 13-10. Codes of SysCmd(7:5) of Processor Data Identifier

Bit	Contents
SysCmd7	Indication of last data element 0: Last data element 1: Not last data element
SysCmd6	Indication of response data 0: Response data 1: Not response data
SysCmd5	Indication of error data 0: Error occurred 1: No error occurred

Table 13-11. Codes of SysCmd(7:4) of External Data Identifier

Bit	Contents
SysCmd7	Indication of last data element 0: Last data element 1: Not last data element
SysCmd6	Indication of response data 0: Response data 1: Not response data
SysCmd5	Indication of error data 0: Error occurred 1: No error occurred
SysCmd4	Data check enables 0: Data and check bit checked 1: Data and check bit not checked

13.7 System Interface Address

The system interface address is a 36-bit physical address and is output to SysAD(35:0) in the address cycle. The other bits of the SysAD bus are not used in the address cycle.

13.7.1 Address specification rules

An address related to transferring data such as a word and an unaligned word is aligned in accordance with the size of the data element. The system uses the following address rules.

- An address related to the request of a block is aligned at the requested doubleword boundary. Therefore, the lower 3 bits of the address are 0.
- The lower 3 bits of an address for a doubleword request are cleared to 0.
- The lower 2 bits of an address for a word request are cleared to 0.
- The least significant bit of an address for a halfword request cleared to 0.
- Each request of 1, 3, 5, 6, and 7 bytes uses a byte address.

13.7.2 Sub-block ordering

The order of the data returned in response to a processor block read request is sub-block ordering. With sub-block ordering, the processor outputs the address of the doubleword required in a block. The external agent must return a block that starts with the specified doubleword, by using sub-block ordering (for details, refer to **APPENDIX A SUB-BLOCK ORDER**).

For a block write request, the processor always outputs the address of the first doubleword in the block. It sequentially outputs the doublewords in the block, starting from the first doubleword of the block.

In the data cycle, whether the byte line of an aligned doubleword (or byte, halfword, 3 bytes, word, 6 bytes, or 7 bytes) is valid or not depends on the position of the data. In the little-endian mode, for example, SysAD(7:0) of a byte request where lower 3 address bits are 0 are valid in the data cycle.

For the byte lane that is used when an unaligned word in big endian and little endian is transferred, refer to **Figure 3-3 Byte Specification Related to Load/Store Instruction**.

13.7.3 Processor internal address map

For an external write, the external agent accesses the internal resources of the processor. When an external write request is made, the processor decodes the SysAD(6:4) bits of the address that is output, to determine which of the resources of the processor is to be accessed. The only internal resource of the processor that can be accessed by an external write request is the interrupt register. Access the interrupt register by an external write access, by specifying an address that clears SysAD(6:4) to 000.

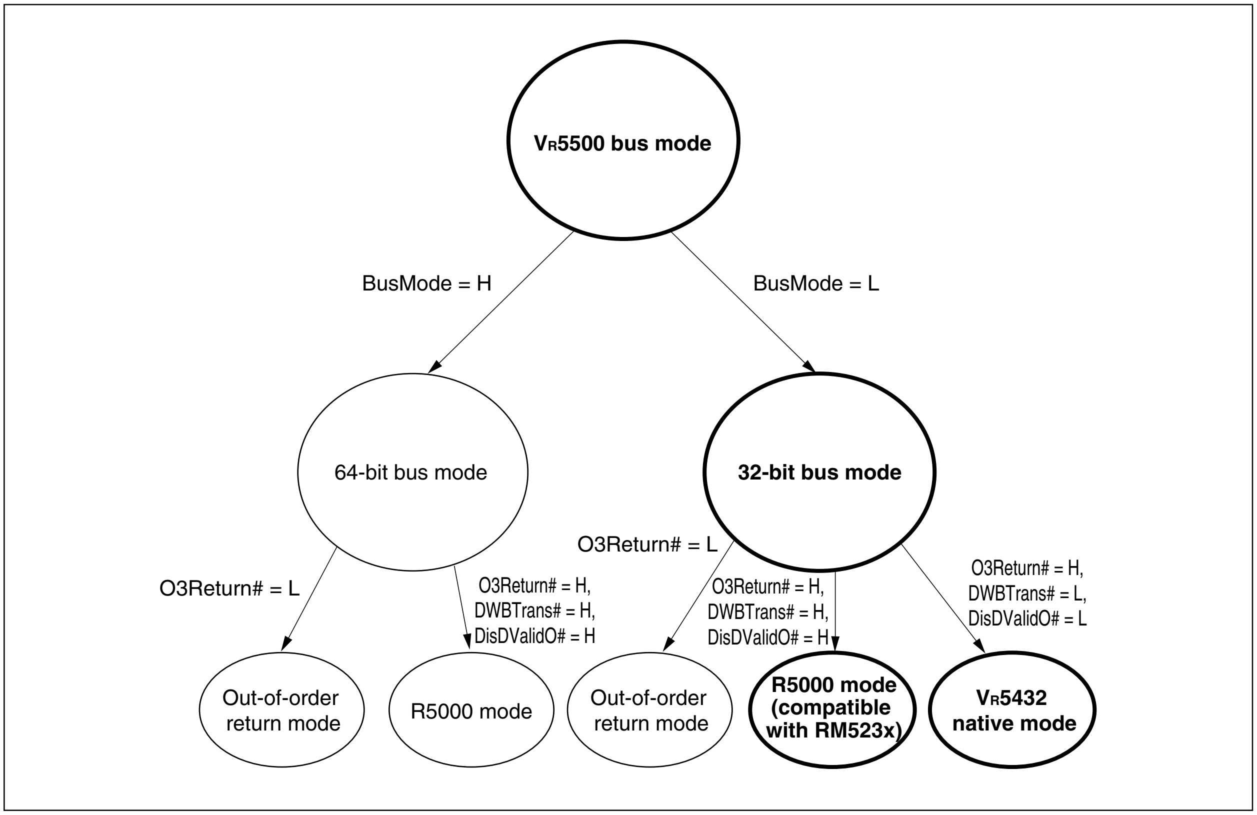
CHAPTER 14 SYSTEM INTERFACE (32-BIT BUS MODE)

This chapter explains the request protocol of the system interface in the 32-bit bus normal mode. The system interface of the VR5500 can be set in the 32-bit bus mode by inputting a low level to the BusMode pin before a power-on reset. It can also be set in the normal mode by inputting a high level to the O3Return# pin before a power-on reset, and in the out-of-order return mode by inputting a low level to the same pin.

The 32-bit bus normal mode includes two protocol modes: R5000 mode and VR5432 native mode. These modes can be selected according to the combination of levels input to the DWBTrans# and DisDValidO# pins before a power-on reset.

- R5000 mode
  The R5000 mode is selected when a high level is input to both the DWBTrans# and DisDValidO# pins. This mode is compatible with the bus protocol of the RM523x (a product of PMC-Sierra).

- VR5432 native mode
  The VR5432 native mode is selected when a low level is input to both the DWBTrans# and DisDValidO# pins. This mode is compatible with the bus protocol of the native mode of the VR5432.


VR5500 bus mode
BusMode = H
BusMode = L
64-bit bus mode
32-bit bus mode
O3Return# = L
O3Return# = H, DWBTrans# = H, DisDValidO# = H
O3Return# = L
O3Return# = H, DWBTrans# = H, DisDValidO# = H
O3Return# = H, DWBTrans# = L, DisDValidO# = L
Out-of-order return mode
R5000 mode
Out-of-order return mode
R5000 mode (compatible with RM523x)
VR5432 native mode

For the protocol in the 64-bit bus normal modes (operation mode compatible with the VR5000), refer to **CHAPTER 13 SYSTEM INTERFACE (64-BIT BUS MODE)**. For the protocol in the out-of-order return mode, refer to **CHAPTER 15 SYSTEM INTERFACE (OUT-OF-ORDER RETURN MODE)**.

14.1 Protocol of Processor Requests

This section explains the following two processor request protocols.

- Read
- Write

14.1.1 Processor read request protocol

The following sequence explains the protocol of a processor read request for a doubleword, unaligned doubleword, word, and unaligned word (the numbers correspond to the numbers in Figure 14-1).

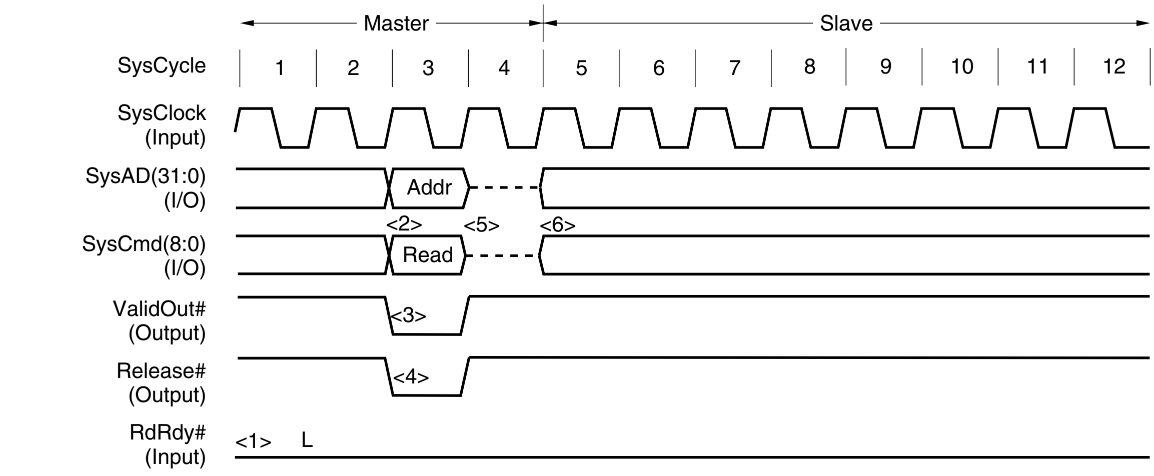
<1> The external agent makes the RdRdy# signal is low and is ready to acknowledge a read request.
<2> When the system interface is in the master status, the processor issues a processor read request by driving a read command onto the SysCmd bus and a read address (physical address) onto the SysAD bus.
<3> At the same time, the processor asserts the ValidOut# signal for the duration of 1 cycle. This signal indicates that valid data is on the SysCmd and SysAD buses.
<4> The processor puts the system interface in the uncompelled slave status. The external agent must wait without asserting the ExtRqst# signal in an attempt to return a read response, until transition of the system interface to the uncompelled slave status is completed.
<5> The processor releases the SysCmd and SysAD buses 1 cycle after the Release# signal has been asserted.
<6> The external agent drives the SysCmd and SysAD buses 2 cycles after the Release# signal has been asserted.

When the system interface has been put in the slave status, the external agent can return the requested data by using a read response. The read response can also return an indication that an error has occurred in the data if the requested data could not be searched correctly, as well as the requested data. If the returned data contains an error, the processor generates a bus error exception.

Figure 14-1 shows the processor read request, and uncompelled transition to the slave status that takes place when the read request is issued.

The timing of the SysADC bus is the same as that of the SysAD bus.

Figure 14-1. Processor Read Request



Remark The dotted line indicates high impedance.

After the Release# signal has been asserted (<6> and later in the figure), the processor can acknowledge both a read response (if the read request is pending) and an external request.

14.1.2 Processor write request protocol

The processor write request is issued by using either of the following two protocols.

- A write request for a word or unaligned word uses a single write request protocol.
- Cache block write and uncached accelerated write uses a block write request protocol.

A processor write request is issued when the system interface is in the master status.

Figure 14-2 shows the processor single write request cycle and Figure 14-3 shows the processor block write request cycle (the numbers in the explanation below correspond to the numbers in the figures).

<1> The external agent makes the WrRdy# signal low and is ready to acknowledge a write request.

<2> The processor issues a processor write request by driving a write command onto the SysCmd bus and a write address onto the SysAD bus.

<3> The processor asserts the ValidOut# signal.

<4> The processor drives a data identifier onto the SysCmd bus and data onto the SysAD bus.

<5> The data identifier corresponding to the data cycle must include an indication of the last data cycle. At the end of the cycle, the ValidOut# signal is deasserted.

Remark The timing of the SysADC bus is the same as that of the SysAD bus.

Figure 14-2. Processor Non-Block Write Request Protocol

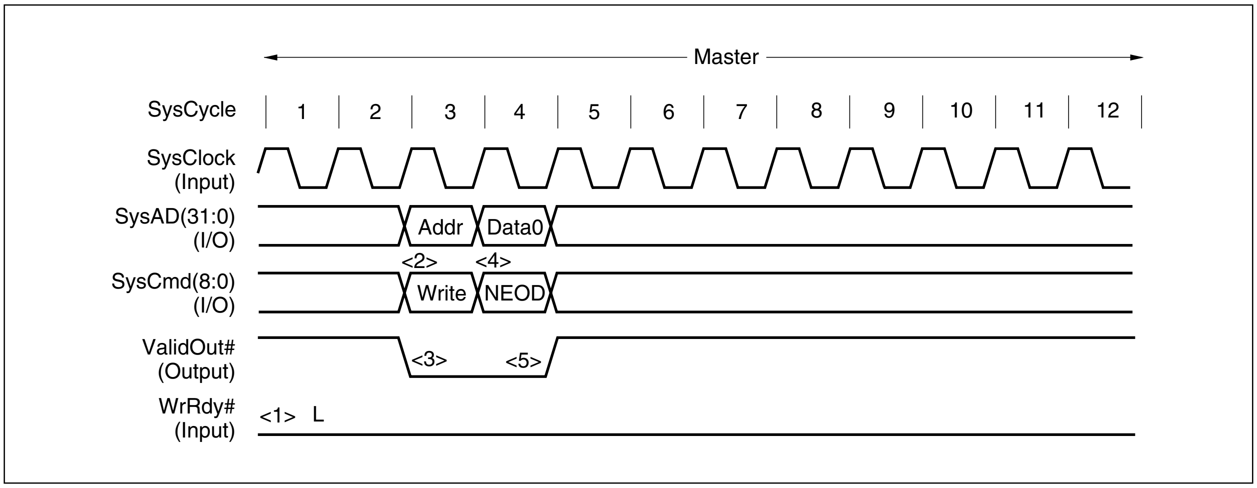
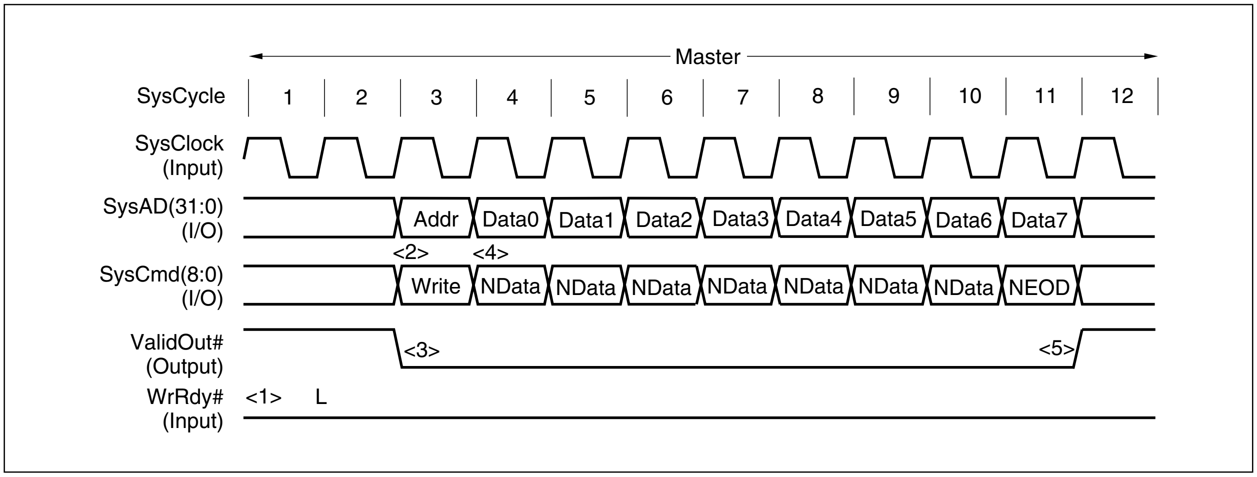


Figure 14-3. Processor Block Write Request



14.1.3 Control of processor request flow

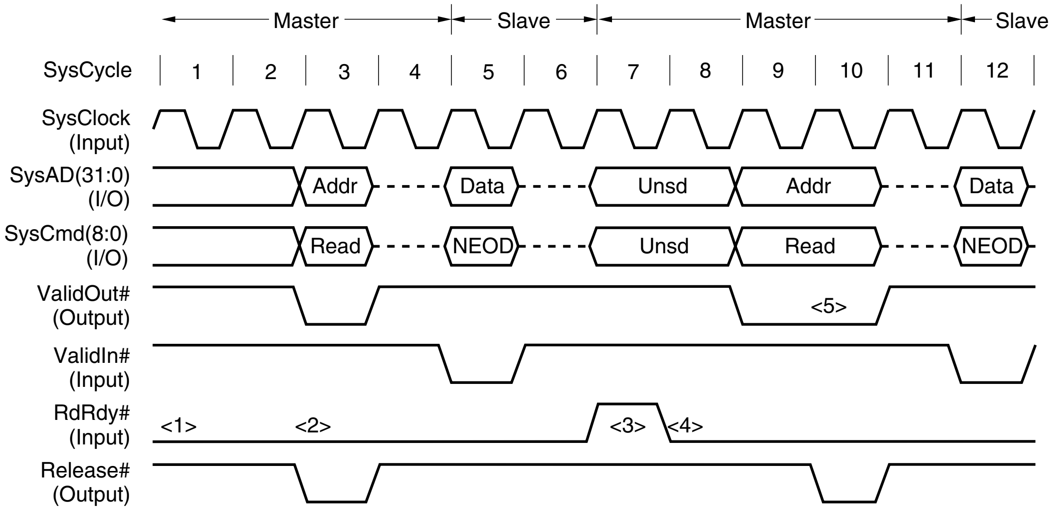
The external agent uses the RdRdy# signal to control the flow of processor read requests.

Figure 14-4 shows the control of the read request flow (the numbers in the explanation below correspond to the numbers in the figure).

<1> The processor samples the RdRdy# signal and determines whether the external agent can acknowledge a read request.
<2> The processor issues a read request to the external agent.
<3> The external agent deasserts the RdRdy# signal. This signal indicates that no more read requests can be acknowledged.
<4> Because the RdRdy# signal is deasserted two cycles before, issuance of the read request is stalled.
<5> The read request is issued again to the external agent.

Figure 14-4. Control of Processor Request Flow (1/2)

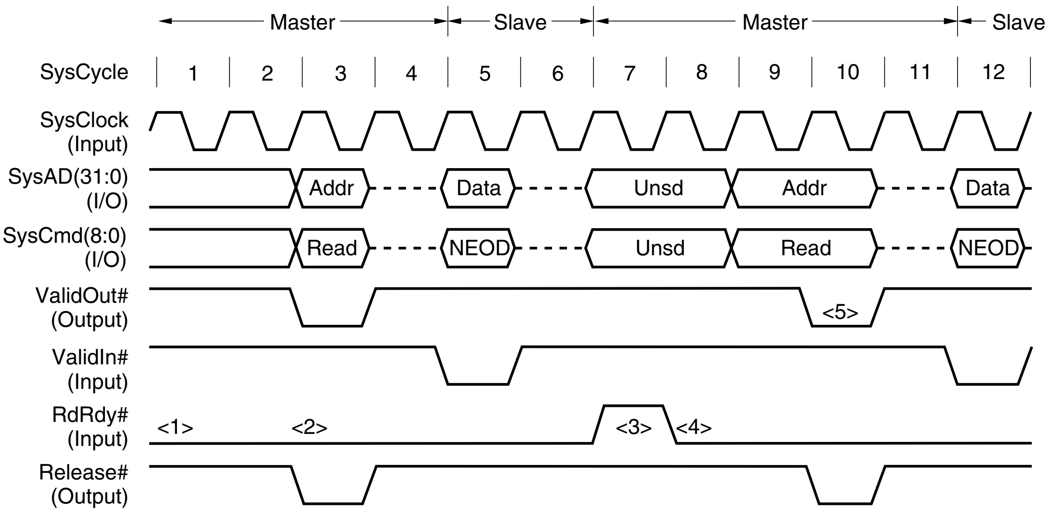
(a) R5000 mode



Remark The dotted line indicates high impedance.

Figure 14-4. Control of Processor Request Flow (2/2)

(b) VR5432 native mode



Remark The dotted line indicates high impedance.

Figure 14-5 shows an example in which two processor write requests are issued but issuance of the second request is delayed because of the condition of the WrRdy# signal (the numbers in the explanation below correspond to the numbers in the figure).

<1> The external agent asserts the WrRdy# signal to indicate that it is ready to acknowledge a write request.

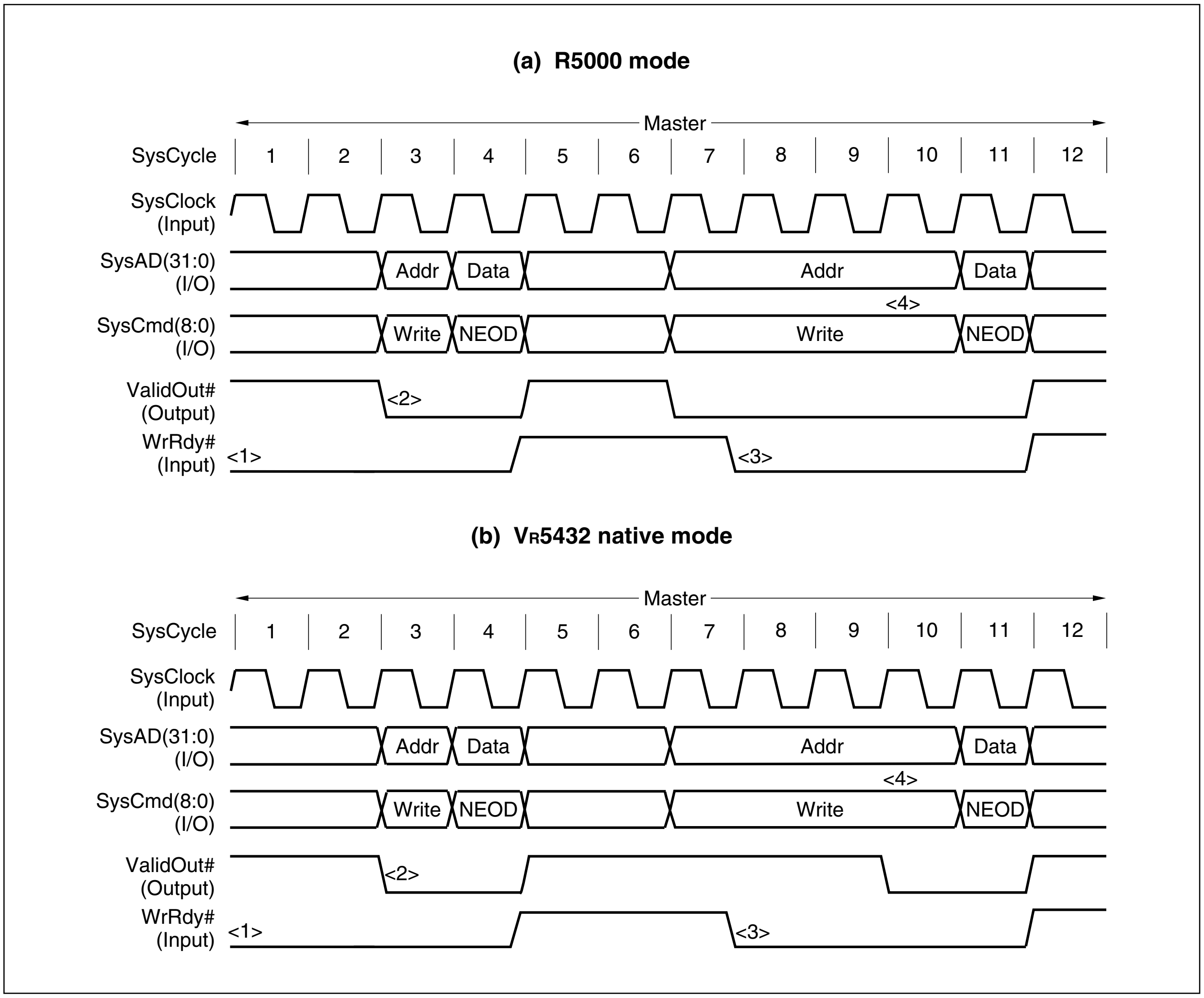
<2> The processor asserts the ValidOut# signal, and drives a write command onto the SysCmd bus and a write address onto the SysAD bus.

<3> The second write request is delayed until the WrRdy# signal is asserted again.

<4> If the WrRdy# signal is active two cycles before, an address cycle is issued in response to the processor write request. This completes the issuance of the write request.

Remark The timing of the SysADC bus is the same as that of the SysAD bus.

Figure 14-5. Timing When Second Processor Write Request Is Delayed



14.1.4 Timing mode of processor request

The VR5500 has three timing modes: VR4000-compatible mode, write re-issuance mode, and pipeline write mode.

- VR4000-compatible mode
  If single write requests are successively issued, the processor inserts two unused cycles after the data cycle so that an address cycle is issued once every 4 system cycles.

- Write re-issuance mode
  If the WrRdy# signal is deasserted in the address cycle of a write request, that request is discarded, but the processor issues the same write request again.

- Pipeline write mode
  Even if the WrRdy# signal is deasserted in the address cycle of a write request, the processor assumes that it has issued that request.

(1) VR4000-compatible mode

With the VR5500 processor interface, the WrRdy# signal must be asserted two system clocks before issuance of a write cycle. If the WrRdy# signal is deasserted immediately after the external agent has received a write request that fills the buffer, the subsequent write requests are kept waiting for the duration of 4 system cycles in the VR4000 non-block-write-compatible mode. The processor inserts at least two unused system cycles after a write address/data pair, giving the external agent the time to keep the next write request waiting.

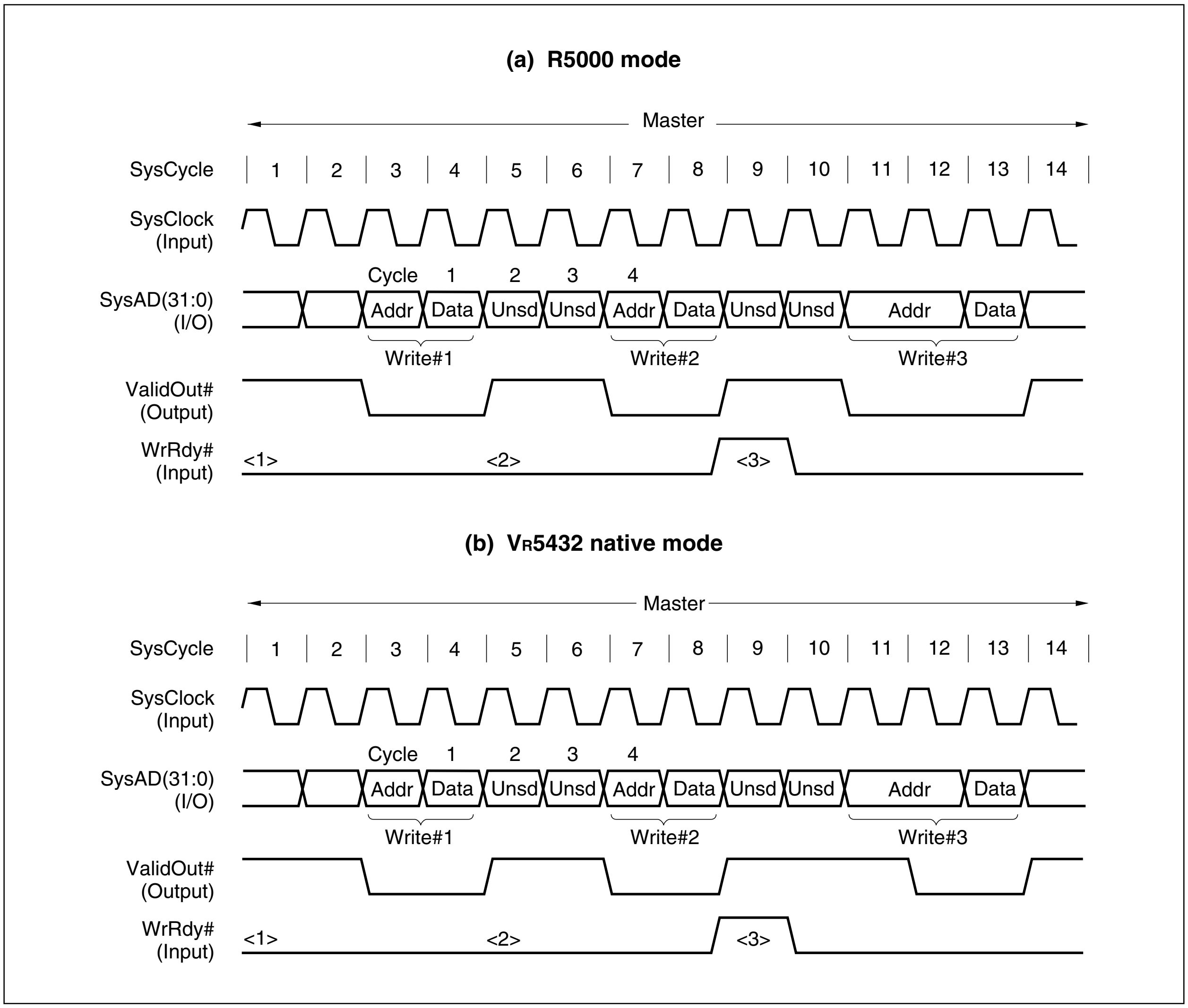
Figure 14-6 shows a back-to-back write cycle in the VR4000-compatible mode (the numbers in the explanation below correspond to the numbers in the figure).

<1> The external agent asserts the WrRdy# signal to indicate that it is ready to issue a write cycle.

<2> The WrRdy# signal remains active. This indicates that the external agent can acknowledge another write request.

<3> The WrRdy# signal is deasserted. This indicates that the external agent cannot acknowledge any more write requests, and that issuance of the next write request is stalled.

Figure 14-6. Timing of VR4000-Compatible Back-to-Back Write Cycle



(a) R5000 mode

(b) VR5432 native mode

(2) Write re-issuance mode

Figure 14-7 shows the write re-issuance protocol (the numbers in the explanation below correspond to the numbers in the figure).

A write request is issued when the WrRdy# signal is asserted two cycles before the address cycle and in the address cycle.

<1> The external agent asserts the WrRdy# signal to indicate that it is ready to acknowledge a write request.

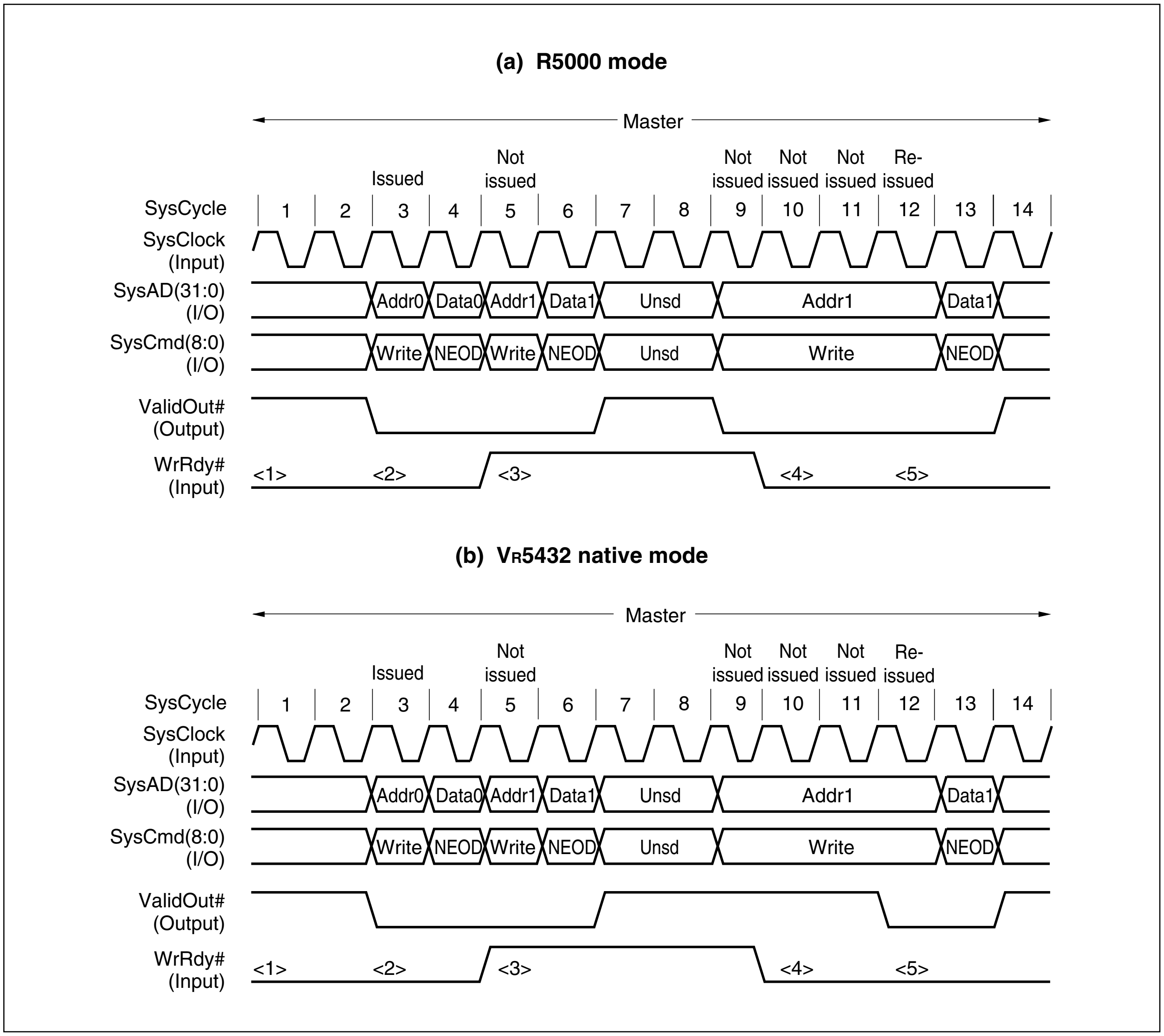
<2> The WrRdy# signal remains active even when the write request has been issued. This indicates that the external agent can acknowledge another write request.

<3> The WrRdy# signal is deasserted in the address cycle. This write cycle is aborted.

<4> The external agent asserts the WrRdy# signal, indicating that it is ready to acknowledge a write request. In response, the write request aborted in <3> is re-issued.

<5> Even if a write request is issued, the WrRdy# signal remains active. This indicates that the external agent can acknowledge another write request.

Figure 14-7. Write Re-Issuance


(a) R5000 mode

(b) VR5432 native mode

(3) Pipeline write mode

Figure 14-8 shows the pipeline write protocol (the numbers in the explanation below correspond to the numbers in the figure). If the WrRdy# signal is issued two cycles before the address cycle, a write request is issued. After the WrRdy# signal has been deasserted, the external agent must acknowledge one more write request.

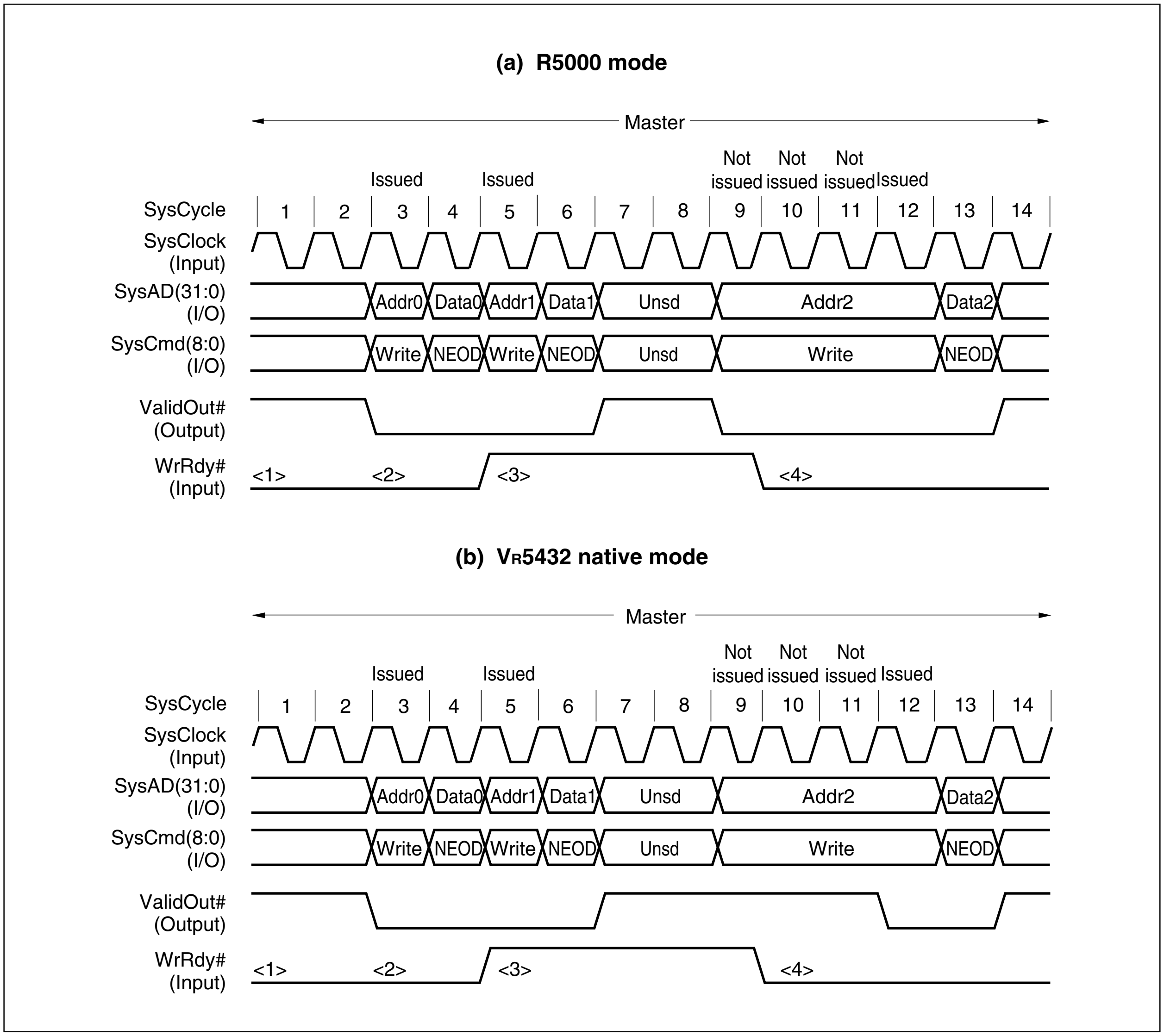
<1> The external agent asserts the WrRdy# signal to indicate that it is ready to acknowledge a write request.

<2> Even when the write request has been issued, the WrRdy# signal remains active. This indicates that the external agent can acknowledge one more write request.

<3> The WrRdy# signal is deasserted. This indicates that the external agent can acknowledge no more write requests. However, this write request is acknowledged.

<4> The external agent asserts the WrRdy# signal, indicating that it can acknowledge a write request.

Figure 14-8. Pipeline Write



(a) R5000 mode

(b) VR5432 native mode

14.2 Protocol of External Request

An external request can be issued only when the system interface is in the slave status. Arbitration that changes the status of the system interface from master to slave is realized by using the handshake signals of the system interface (ExtRqst# and Release#).

This section explains the following external request protocols, as well as the arbitration protocol.

- Null
- Write
- Read response

14.2.1 External arbitration protocol

To issue an external request, assert the ExtRqst# signal to arbitrate the system interface. Then wait until the processor asserts the Release# signal and releases the system interface to the slave status. When the system interface is already in the slave status, i.e., when the processor previously executed an uncompelled transition of the system interface to the slave status, the external agent can immediately start issuing an external request.

After issuing an external request, the external agent must return the right to control the system interface to the processor.

If the external agent does not have any more external requests that must be processed, it must deassert the ExtRqst# signal two cycles after the Release# signal was asserted. To issue two or more requests in a row, the ExtRqst# signal must be kept active until the last request cycle. If the last request cycle lasts for two cycles or more after the Release# signal was asserted, deassert the ExtRqst# signal.

While the ExtRqst# signal is active, the processor continues processing the external request. However, the processor cannot release the system interface to process the next external request until processing of the current request is finished. While the ExtRqst# signal is active, two or more successive external requests cannot be interrupted by a processor request.

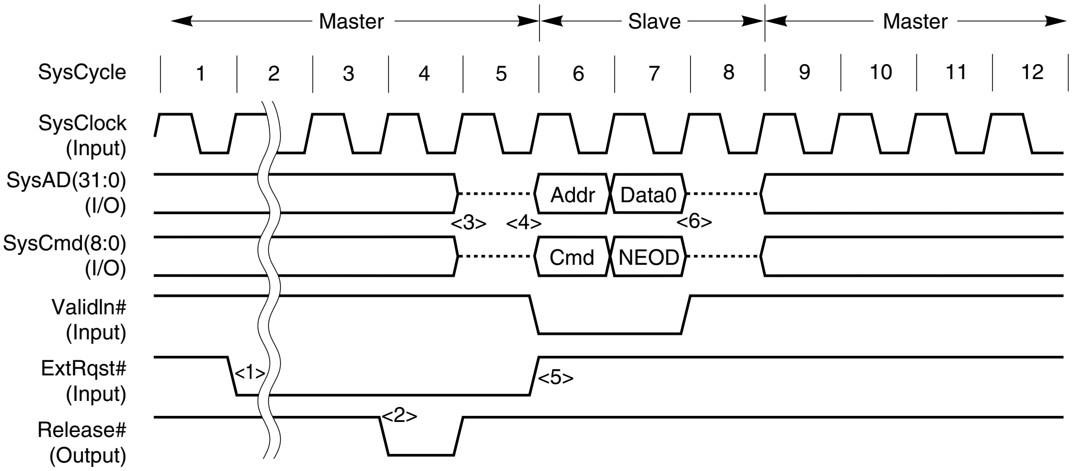
Figure 14-9 shows the arbitration protocol of an external request issued by the external agent. The following sequence explains the arbitration protocol (the numbers in the explanation below correspond to the numbers in the figure).

<1> The external agent continues asserting the ExtRqst# signal to issue an external request.
<2> The processor asserts the Release# signal for 1 cycle when it is ready to process the external request.
<3> The processor makes the SysAD and SysCmd buses go into a high-impedance state.
<4> The external agent must drive the SysAD and SysCmd buses at least two cycles after the Release# signal was asserted.
<5> The external agent must deassert the ExtRqst# signal two cycles after the Release# signal was asserted, except when it executes another external request.
<6> The external agent must make the SysAD and SysCmd buses go into a high-impedance state on completion of the external request.

Remarks 1. The processor can issue a request one cycle after the external agent has set the system interface to a high-impedance state.

2. The timing of the SysADC bus is the same as that of the SysAD bus.

Figure 14-9. External Request Arbitration Protocol



Remark The dotted line indicates high impedance.

14.2.2 External null request protocol

The processor supports an external null request. This request only returns the system interface from the slave status to the master status, and does not have any other influence on the processor.

Figure 14-10 shows the timing of the external null request (the numbers in the explanation below correspond to the numbers in the figure).

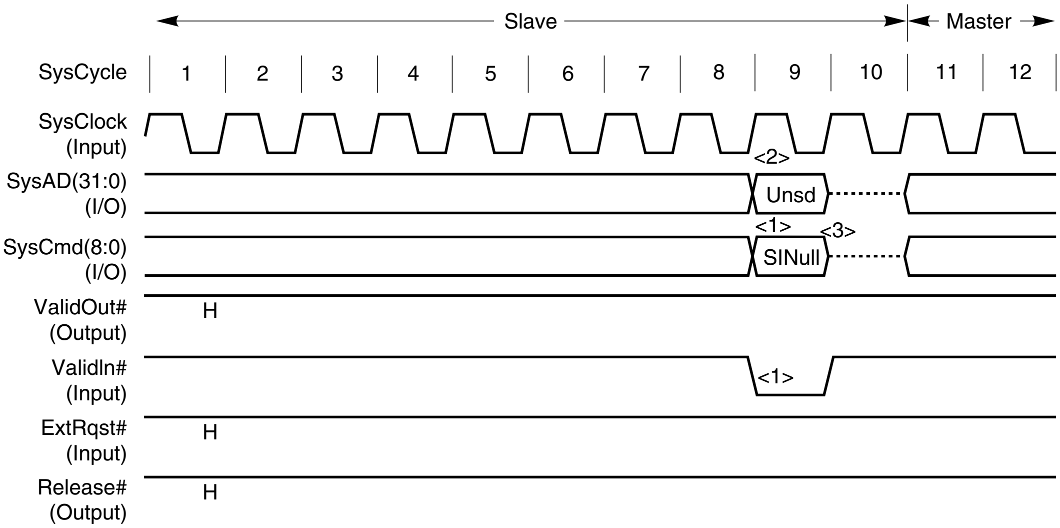
<1> The external agent drives an external null request command onto the SysCmd bus and asserts the ValidIn# signal for one cycle. This returns the right to control the system interface to the processor.

<2> The SysAD bus is not used in the address cycle corresponding to the external null request (the bus does not hold valid data).

<3> When the address cycle is issued, the null request is completed.

The external null request returns the system interface to the master status when the external agent has released the SysCmd and SysAD buses.

Figure 14-10. External Null Request Protocol



Remark The dotted line indicates high impedance.

14.2.3 External write request protocol

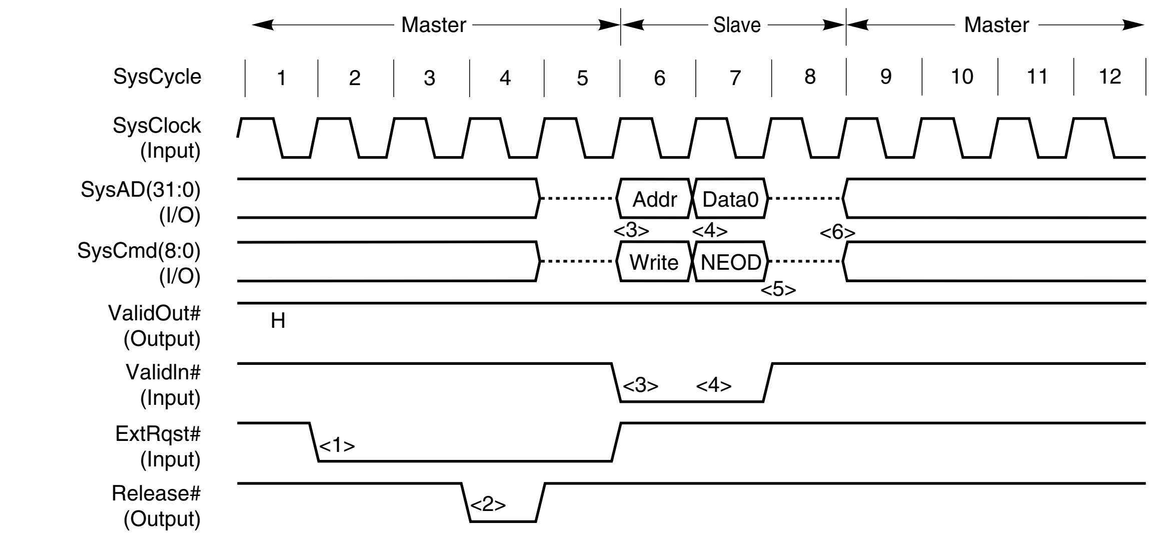
The external write request performs an operation close to the processor single write request, except that it asserts the ValidIn# signal, instead of the ValidOut# signal.

Figure 14-11 shows the timing of the external write request (the numbers in the explanation below correspond to the numbers in the figure).

<1> The external agent asserts the ExtRqst# signal to arbitrate the system interface.
<2> The processor asserts the Release# signal to release the system interface to the slave status.
<3> The external agent asserts the ValidIn# signal and drives a write command onto the SysCmd bus and a write address onto the SysAD bus.
<4> The external agent asserts the ValidIn# signal and drives a data identifier onto the SysCmd bus and data onto the SysAD bus.
<5> The data identifier corresponding to the data cycle must contain an indication of the last data cycle.
<6> When the data cycle is issued, the write request is completed. The external agent makes the SysCmd and SysAD buses go into a high-impedance state, and returns the system interface to the master status.

The external write request can only write word data to the processor. If a data element other than a word is specified for the external write request, the operation of the processor is undefined.

Figure 14-11. External Write Request Protocol



Remark The dotted line indicates high impedance.

14.2.4 Read response protocol

The external agent must return data to the processor by using a read response protocol, in response to a processor read request. The following sequence explains the read response protocol (the numbers in the explanation below correspond to the numbers in Figures 14-12 and 14-13).

<1> The external agent waits until the processor puts the system interface in the uncompelled slave status.
<2> The processor returns data via a single data cycle or a series of data cycles.
<3> When the last data cycle is issued, the read response is completed, and the external agent makes the SysCmd and SysAD buses go into a high-impedance state.
<4> The system interface returns to the master status.

Remark When the read request is issued, the processor always puts the system interface in the uncompelled slave status.

<5> The data identifier of the data cycle must indicate that this data is response data.
<6> The data identifier corresponding to the last data cycle must contain an indication of the last data cycle.

If the read response is for a block read request, the response data does not have to identify the initial cache status. The processor automatically allocates the cache to the clean status.

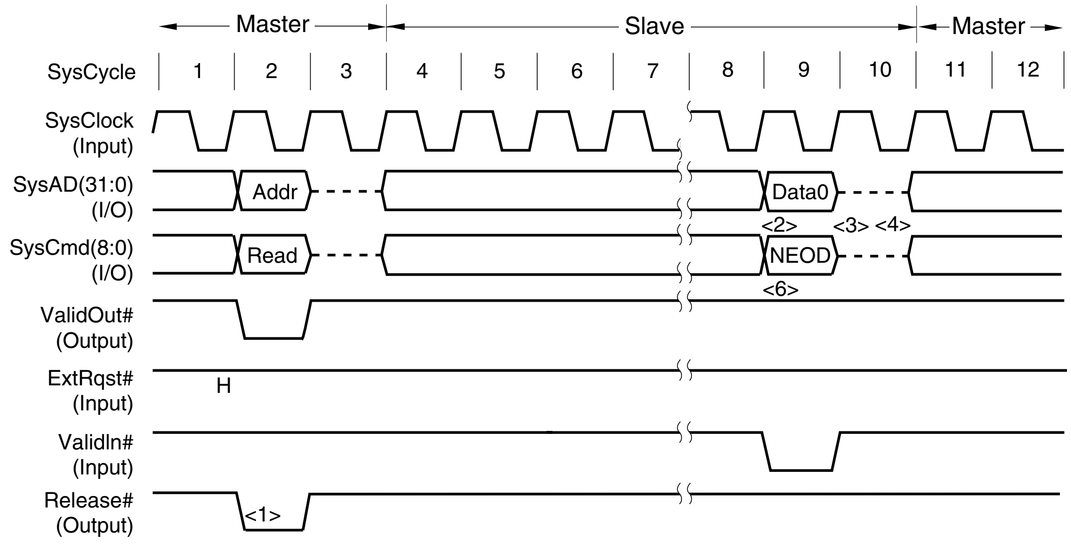
The data identifier corresponding to the data cycle can indicate that the data transferred in that cycle has an error. Even if data may have an error, however, the external agent must return a data block of the correct size.

Only when there is a pending processor read request, read response data is passed to the processor. The operation of the processor is undefined if there is no pending processor read request when a read response is received.

Figure 14-12 shows a processor word request and the word read response that follows. Figure 14-13 shows the read response to a processor block read request when the system interface is already in the slave status.

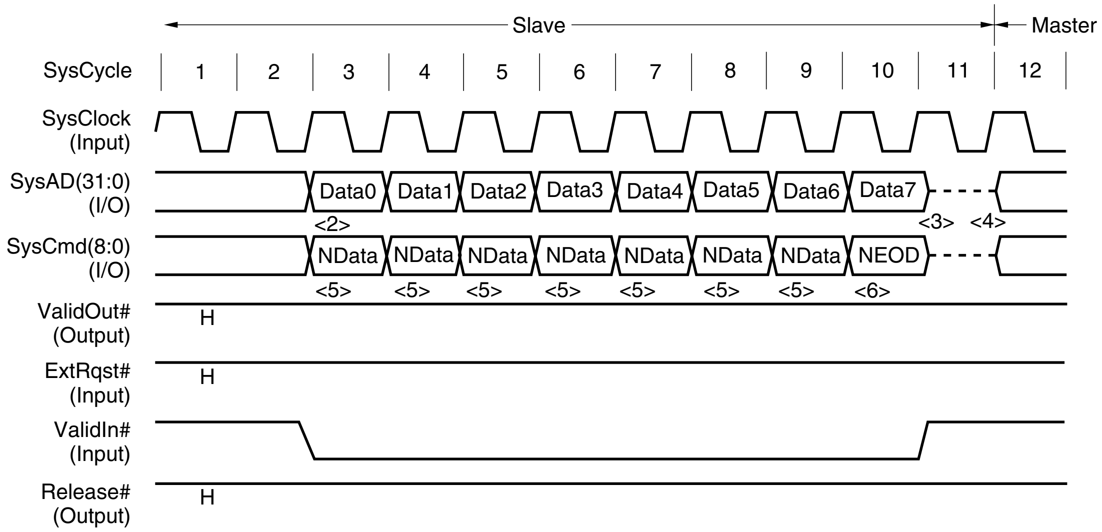
Remark The timing of the SysADC bus is the same as that of the SysAD bus.

Figure 14-12. Protocol of Read Request and Read Response



Remark The dotted line indicates high impedance.

Figure 14-13. Block Read Response in Slave Status



Remark The dotted line indicates high impedance.

14.2.5 SysADC(3:0) protocol for block read response

When a block read response is issued, SysADC(3:0) must be used in compliance with the following rules.

- Only the first two words of transfer data are checked. If the data has an error (SysCmd5 = 1), the cache line is invalidated, and a bus error exception occurs in the processor.
- A parity error of the first two words is detected when a request is issues, and a cache error exception occurs. At this time, the cache line is in the Invalid status. A parity error of a subsequent word is detected again when that data is used.
- The error bits in six subsequent words of data are ignored. The parity of each word is written to the cache, but is not checked until the data is referenced.
- If a memory error occurs during a block read operation, the SysADC bit must be changed to an illegal parity during a read response operation for all the bytes that are affected by the memory error. However, even if SysCmd5 is set to 1 during data transfer other than the first two words, a bus error exception does not occur. If the SysADC bit has been changed to an illegal parity, a cache error exception occurs when any of the remaining six words is referenced.

14.3 Data Flow Control

The system interface supports a data rate of 1 word per cycle.

14.3.1 Data rate control

The external agent can send data to the processor at the maximum data rate of the system interface.

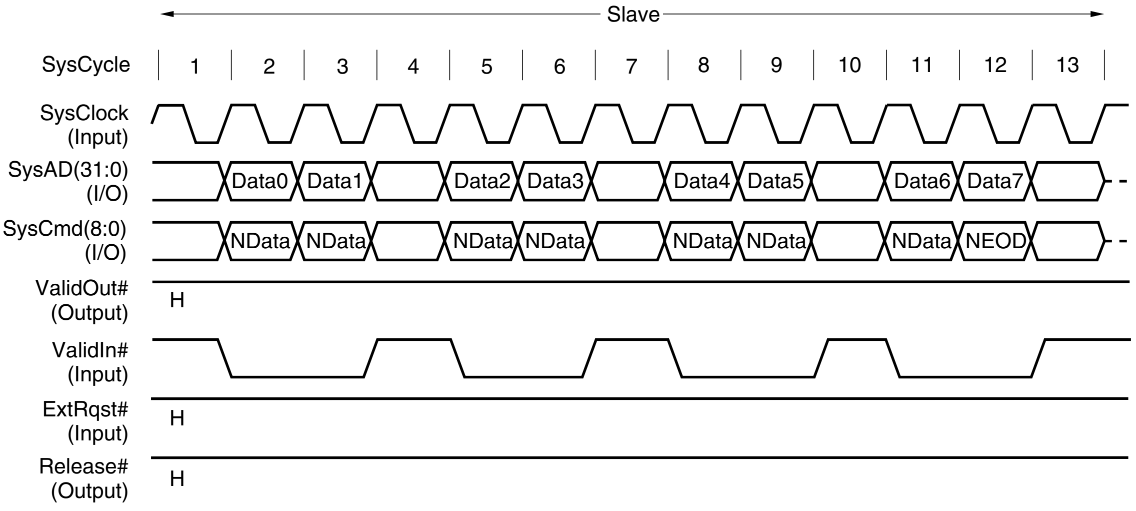
The rate at which data is to be sent to the processor can be controlled on the external agent side. The transfer rate from the external agent is not limited. The external agent asserts the ValidIn# signal in the cycle in which it transfers data.

When the ValidIn# signal has been asserted and as long as a data identifier is on the SysCmd bus, the processor acknowledges the cycle as valid. It then goes on acknowledging data until it receives a data word with NEOD.

The operation of the processor is undefined if data is sent in a pattern of other than 1 cycle for single data, and other than 2 or 8 cycles for block data.

Figure 14-14 shows the timing of the read response where the data rate pattern is DDx.

Figure 14-14. Read Response with Data Rate Pattern DDx



Remark The dotted line indicates high impedance.

14.3.2 Block write data transfer pattern

The rate at which the processor transfers block write data to the external agent can be set by the EP bit of the Config register after reset. The data pattern is indicated by characters D and x that indicate the array of data cycle and unused cycle at each data rate. D indicates a data cycle, and x indicates an unused cycle. For example, Dxx data pattern indicates a data rate of 1 word in every 3 cycles.

Table 14-1 shows the maximum data rate that can be set after reset.

Table 14-1. Transfer Data Rate and Data Pattern

Maximum Data Rate	Data Pattern
1 word/1 cycle	DDDDDDDD
2 words/3 cycles	DDxDDxDDxDDx
2 words/4 cycles	DDxxDDxxDDxxDDxx
1 word/2 cycles	DxDxDxDxDxDxDxDx
2 words/5 cycles	DDxxxDDxxxDDxxxDDxxx
2 words/6 cycles	DDxxxxDDxxxxDDxxxxDDxxxx
1 word/3 cycles	DxxDxxDxxDxxDxxDxxDxxDxx
2 words/8 cycles	DDxxxxxxDDxxxxxxDDxxxxxxDDxxxxxx
1 word/4 cycles	DxxxDxxxDxxxDxxxDxxxDxxxDxxxDxxx

14.3.3 Word transfer sequence

The VR5500 transfers a 32-bit address in one address cycle and 32-bit data in one data cycle. It takes two system cycles to transfer each doubleword as a block. Data is transferred in these two cycles in the following sequence.

- The lower 4 bytes (lower word) are transferred in the first data cycle in the little-endian mode, and in the second data cycle in the big-endian mode.
- The higher 4 bytes (higher word) are transferred in the second data cycle in the little-endian mode, and in the first data cycle in the big-endian mode.

The VR5500 can transfer a word or an unaligned word in one system cycle.

The table below shows the transfer sequence in both the little-endian and big-endian modes to write a block, doubleword, unaligned doubleword, word, and unaligned word.

Table 14-2. Data Write Sequence

Transfer Type	Little Endian	Big Endian
Block	1. A(31:0) 2. D0(31:0) 3. D0(63:32) 4. D1(31:0) 5. D1(63:32) 6. D2(31:0) 7. D2(63:32) 8. D3(31:0) 9. D3(63:32)	1. A(31:0) 2. D0(63:32) 3. D0(31:0) 4. D1(63:32) 5. D1(31:0) 6. D2(63:32) 7. D2(31:0) 8. D3(63:32) 9. D3(31:0)
Doubleword (in R5000 mode)	1. A(31:0) 2. D(31:0) 3. A(31:0) 4. D(63:32)	1. A(31:0) 2. D(63:32) 3. A(31:0) 4. D(31:0)
Doubleword (in VR5432 native mode)	1. A(31:0) 2. D(31:0) 3. D(63:32)	1. A(31:0) 2. D(63:32) 3. D(31:0)
Word or unaligned word	1. A(31:0) 2. W(31:0)	1. A(31:0) 2. W(31:0)

Remark A: Address, D: Doubleword, W: Word
Dn: n+1th doubleword in block data (n = 0 to 3)
Dn(31:0): Lower word of doubleword data Dn(63:0)
Dn(63:32): Higher word of doubleword data Dn(63:0)

With the VR5500, a doubleword is read in accordance with the sub-block order (refer to **APPENDIX A SUB-BLOCK ORDER**) when a cache line is obtained from the external agent and replaced. Doubleword transfer in this case is treated as 2-word transfer in sub-block order. The other doublewords, unaligned doublewords, words, and unaligned words are read in the same sequence as when they are written.

The table below shows the transfer sequence in both the little-endian and big-endian modes to read a block, doubleword, unaligned doubleword, word, and unaligned word.

Table 14-3. Data Read Sequence (1/2)

Transfer Type	Little Endian	Big Endian
Block (when A(4:3) = 00)	1. D0(31:0) 2. D0(63:32) 3. D1(31:0) 4. D1(63:32) 5. D2(31:0) 6. D2(63:32) 7. D3(31:0) 8. D3(63:32)	1. D0(63:32) 2. D0(31:0) 3. D1(63:32) 4. D1(31:0) 5. D2(63:32) 6. D2(31:0) 7. D3(63:32) 8. D3(31:0)
Block (when A(4:3) = 01)	1. D1(31:0) 2. D1(63:32) 3. D0(31:0) 4. D0(63:32) 5. D3(31:0) 6. D3(63:32) 7. D2(31:0) 8. D2(63:32)	1. D1(63:32) 2. D1(31:0) 3. D0(63:32) 4. D0(31:0) 5. D3(63:32) 6. D3(31:0) 7. D2(63:32) 8. D2(31:0)
Block (when A(4:3) = 10)	1. D2(31:0) 2. D2(63:32) 3. D3(31:0) 4. D3(63:32) 5. D0(31:0) 6. D0(63:32) 7. D1(31:0) 8. D1(63:32)	1. D2(63:32) 2. D2(31:0) 3. D3(63:32) 4. D3(31:0) 5. D0(63:32) 6. D0(31:0) 7. D1(63:32) 8. D1(31:0)

Remark A: Address, D: Doubleword, W: Word
Dn: n+1th doubleword in block data (n = 0 to 3)
Dn(31:0): Lower word of doubleword data Dn(63:0)
Dn(63:32): Higher word of doubleword data Dn(63:0)

Table 14-3. Data Read Sequence (2/2)

Transfer Type	Little Endian	Big Endian
Block (when A(4:3) = 11)	1. D3(31:0) 2. D3(63:32) 3. D2(31:0) 4. D2(63:32) 5. D1(31:0) 6. D1(63:32) 7. D0(31:0) 8. D0(63:32)	1. D3(63:32) 2. D3(31:0) 3. D2(63:32) 4. D2(31:0) 5. D1(63:32) 6. D1(31:0) 7. D0(63:32) 8. D0(31:0)
Doubleword (VR5432 native mode)	1. D(31:0) 2. D(63:32)	1. D(63:32) 2. D(31:0)
Word, unaligned word	1. W(31:0)	1. W(31:0)

Remarks 1. Doubleword read requests are not supported in R5000 mode.

2. A: Address, D: Doubleword, W: Word
Dn: n+1th doubleword in block data (n = 0 to 3)
Dn(31:0): Lower word of doubleword data Dn(63:0)
Dn(63:32): Higher word of doubleword data Dn(63:0)

The external agent can write 1 word of data to the VR5500 at a time (refer to **Figure 14-11**). Therefore, it takes the external agent 1 system cycle to transfer a word to the VR5500.

14.3.4 System endianness

The endianness of the system is set by the BigEndian pin after reset. The set endianness is indicated by the BE bit of the Config register.

14.4 Independent Transfer with SysAD Bus

For general applications, the SysAD bus connects the processor and a bidirectional register type transceiver in the external agent between two points. For such applications, only the processor and external agent can be connected to the SysAD bus.

For specific applications, other drivers and receivers are connected to the SysAD bus so that transfer can be performed independently of the processor on the SysAD bus. This is called independent transfer. To execute independent transfer, the external agent must adjust the right to control the SysAD bus by using the arbitration handshake signals and external null request.

The procedure of independent transfer of the SysAD bus is as follows.

<1> The external agent requests the right to control the SysAD bus by asserting the ExtRqst# signal to issue an external request.

<2> The processor releases the system interface to the slave status by asserting the Release# signal.

<3> In this way, the external agent can execute independent transfer on the SysAD bus. The ValidIn# signal must not be asserted during transfer.

<4> When transfer is completed, the external agent releases and returns the system interface to the master status by issuing an external null request.

14.5 System Interface Cycle Time

Because processor requests are restricted by the system interface protocol, the number of request cycles is checked by the protocol. Because external requests have the following two types of wait times, the number of request cycles differs depending on these wait times.

- Standby time until the processor releases the system interface to the slave status in response to an external request (release wait time)
- Response time of the external request that requires a response (external response wait time)

While an external request is being issued, the release wait time differs depending on the status of the system interface. When the external request is detected, the system interface is released to the external agent after the cycle under processing.

The external response time of the VR5500 is kept to the minimum. Data that is written is immediately loaded.

14.6 System Interface Commands and Data Identifiers

A system interface command defines the type and attribute of a system interface request. This definition is indicated in the address cycle of a request.

The system interface data identifier defines the attribute of the data transferred in the system interface data cycle.

This section explains the syntax of the commands and data identifiers of the system interface, i.e., coding in bit units.

Set the reserved bits and reserved area in the commands and data identifiers of the system interface related to external requests to 1.

The reserved bits and reserved area in the commands and data identifiers of the system interface related to processor requests are undefined.

14.6.1 Syntax of commands and data identifiers

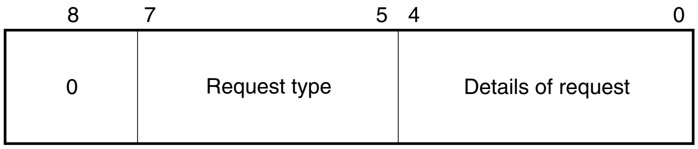
The commands and data identifiers of the system interface are coded in 9-bit units, and transferred from the processor to the external agent, or vice versa, via the SysCmd bus in the address cycle and data cycle.

SysCmd8 (most significant bit) determines whether the current contents of the SysCmd bus are a command (address cycle) or data identifier (data cycle). If they are a command, clear SysCmd8 to 0; if they are a data identifier, set it to 1.

14.6.2 Syntax of command

This section explains the coding of the SysCmd bus when a system interface command is used. Figure 14-15 shows the common code used for all the system interface commands.

Figure 14-15. Bit Definition of System Interface Command



Be sure to clear SysCmd8 to 0 when a system interface command is used.

SysCmd(7:5) define the types of system interface requests such as read, write, and null.

Table 14-4. Code of System Interface Command SysCmd(7:5)

Bit	Contents
SysCmd(7:5)	Command 0: Read request 1: Reserved 2: Write request 3: Null request 4 to 7: Reserved

SysCmd(4:0) are determined according to the type of request. A definition of each request is given below.

(1) Read request

The code of the SysCmd bus related to a read request is shown below.

Figure 14-16 shows the format of the command when a read request is issued.

Tables 14-5 to 14-7 show the code of the read attribute of the SysCmd(4:0) bits related to the read request.

Figure 14-16. Bit Definition of SysCmd Bus During Read Request

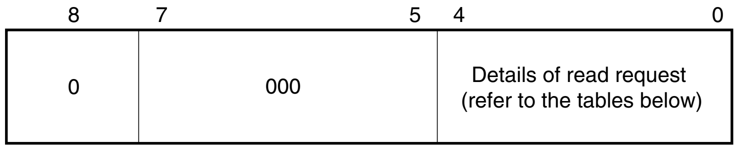


Table 14-5. Code of SysCmd(4:3) During Read Request

Bit	Contents
SysCmd(4:3)	Read attribute 0, 1: Reserved 2: Block read 3: Single read

Table 14-6. Code of SysCmd(2:0) During Block Read Request

Bit	Contents
SysCmd2	Reserved
SysCmd(1:0)	Size of read block 0: 2 words (in VR5432 native mode only) 1: 8 words 2, 3: Reserved

Table 14-7. Code of SysCmd(2:0) During Single Read Request

Bit	Contents
SysCmd2	Reserved
SysCmd(1:0)	Read data size 0: 1 byte is valid (byte). 1: 2 bytes are valid (halfword). 2: 3 bytes are valid. 3: 4 bytes are valid (word).

(2) Write request

The code of the SysCmd bus related to a write request is shown below.

Figure 14-17 shows the format of the command when a write request is issued.

Tables 14-8 to 14-10 show the code of the write attribute of the SysCmd(4:0) bits related to the write request.

Figure 14-17. Bit Definition of SysCmd Bus During Write Request

8	7 5	4 0
0	010	Details of write request (refer to the tables below)

Table 14-8. Code of SysCmd(4:3) During Write Request

Bit	Contents
SysCmd(4:3)	Write attribute 0, 1: Reserved 2: Block write 3: Single write

Table 14-9. Code of SysCmd(2:0) During Block Write Request

Bit	Contents
SysCmd2	Update of cache line 0: Replaced 1: Retained
SysCmd(1:0)	Size of write block 0: 2 words (in VR5432 native mode only) 1: 8 words 2, 3: Reserved

Table 14-10. Code of SysCmd(2:0) During Single Write Request

Bit	Contents
SysCmd2	Reserved
SysCmd(1:0)	Write data size 0: 1 byte is valid (byte). 1: 2 bytes are valid (halfword). 2: 3 bytes are valid. 3: 4 bytes are valid (word).

(3) Null request

Figure 14-18 shows the format of the command when a null request is used.

Figure 14-18. Bit Definition of SysCmd Bus During Null Request

8	7 5	4 0
0	011	Details of null request (refer to the table below)

Table 14-11 shows the code of the SysCmd(4:3) bits related to the null request.

For the null request, the SysCmd(2:0) bits are reserved.

Table 14-11. Code of SysCmd(4:3) During Null Request

Bit	Contents
SysCmd(4:3)	Null attribute 0: Released 1 to 3: Reserved

14.6.3 Syntax of data identifier

This section explains coding of the SysCmd bus when a system interface data identifier is used.

Figure 14-19 shows the common code used for all system interface data identifiers.

Figure 14-19. Bit Definition of System Interface Data Identifier

8	7	6	5	4	3 0
1	Indication of last data	Indication of response data	Indication of error data	Data check enable	Reserved

Be sure to set SysCmd8 of the system interface data identifier to 1.

A definition of the SysCmd(7:0) bits is given below.

SysCmd7: Indicates whether the data element is the last one.

SysCmd6: Indicates whether the data is response data. Response data is returned in response to a read request.

SysCmd5: Indicates whether the data element contains an error. The error indicated in the data cannot be corrected. If this data is returned to the processor, a bus error exception occurs. In the case of a response block, send the entire line to the processor regardless of the degree of error. The external agent should ignore this bit in a processor data identifier because no error is indicated.

SysCmd4: This bit in an external data identifier indicates whether the data of the data element and check bit are checked. This bit in a processor data identifier is reserved.

SysCmd(3:0): These bits are reserved.

Table 14-12 indicates the codes of SysCmd(7:5) of a processor data identifier, and Table 14-13 shows the codes of SysCmd(7:4) of an external data identifier.

Table 14-12. Codes of SysCmd(7:5) of Processor Data Identifier

Bit	Contents
SysCmd7	Indication of last data element 0: Last data element 1: Not last data element
SysCmd6	Indication of response data 0: Response data 1: Not response data
SysCmd5	Indication of error data 0: Error occurred 1: No error occurred

Table 14-13. Codes of SysCmd(7:4) of External Data Identifier

Bit	Contents
SysCmd7	Indication of last data element 0: Last data element 1: Not last data element
SysCmd6	Indication of response data 0: Response data 1: Not response data
SysCmd5	Indication of error data 0: Error occurred 1: No error occurred
SysCmd4	Data check enables 0: Data and check bit checked 1: Data and check bit not checked

14.7 System Interface Address

The system interface address is a 32-bit physical address and is output in the address cycle, using all the bits of the SysAD bus.

14.7.1 Address specification rules

An address related to transferring data such as a word and an unaligned word is aligned in accordance with the size of the data element. The system uses the following address rules.

- An address related to the request of a block is aligned at the requested doubleword boundary. Therefore, the lower 3 bits of the address are 0.
- The lower 3 bits of an address for a doubleword request are cleared to 0.
- The lower 2 bits of an address for a word request are cleared to 0.
- The least significant bit of an address for a halfword request cleared to 0.
- Each request of 1 and 3 bytes uses a byte address.

14.7.2 Sub-block ordering

The order of the data returned in response to a processor block read request is sub-block ordering. With sub-block ordering, the processor outputs the address of the doubleword required in a block. The external agent must return a block that starts with the specified doubleword, by using sub-block ordering (for details, refer to **APPENDIX A SUB-BLOCK ORDER**).

For a block write request, the processor always outputs the address of the first doubleword in the block. It sequentially outputs the doublewords in the block, starting from the first doubleword of the block.

Remark The sequence of the data in a doubleword differs depending on the endianness (refer to **Tables 14-2** and **14-3**).

In the data cycle, whether the byte line of an aligned doubleword (or byte, halfword, 3 bytes, or word) is valid or not depends on the position of the data. In the little-endian mode, for example, SysAD(7:0) of a byte request where lower 3 address bits are 0 are valid in the data cycle.

For the byte lane that is used when an unaligned word in big endian and little endian is transferred, refer to Figure 3-3 Byte Specification Related to Load/Store Instruction.

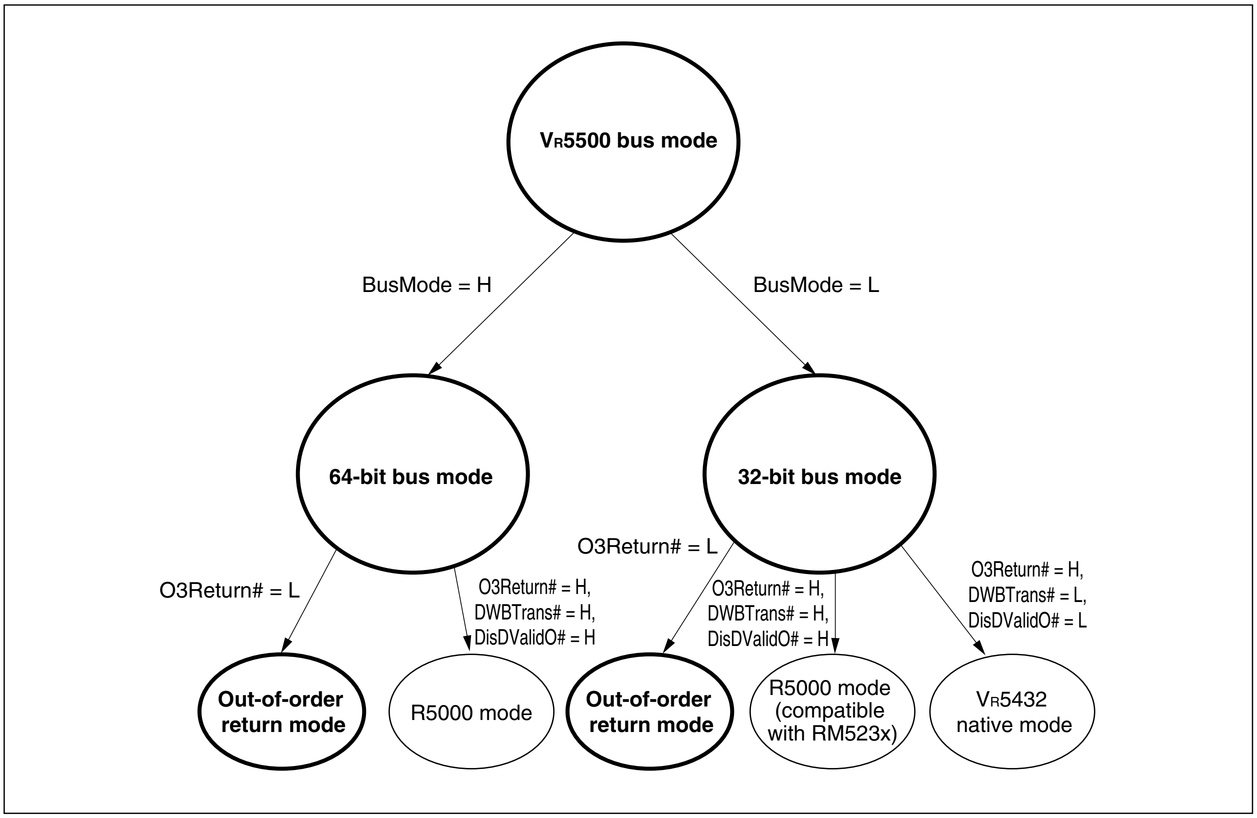
14.7.3 Processor internal address map

For an external write, the external agent accesses the internal resources of the processor. When an external write request is made, the processor decodes the SysAD(6:4) bits of the address that is output, to determine which of the resources of the processor is to be accessed. The only internal resource of the processor that can be accessed by an external write request is the interrupt register. Access the interrupt register by an external write access, by specifying an address that clears SysAD(6:4) to 000.

CHAPTER 15 SYSTEM INTERFACE (OUT-OF-ORDER RETURN MODE)

This chapter explains the request protocol of the system interface in the 64-/32-bit out-of-order return mode.

The system interface of the VR5500 enters the out-of-order return mode when a low level is input to the O3Return# pin before a power-on reset.



For the protocol in the normal mode (R5000 mode (operation mode compatible with the VR5000 Series and RM523x) and VR5432 native mode), refer to **CHAPTER 13 SYSTEM INTERFACE (64-BIT BUS MODE)** and **CHAPTER 14 SYSTEM INTERFACE (32-BIT BUS MODE)**.

15.1 Overview

In the out-of-order return mode, the external agent can return a response to a processor read request regardless of the order in which the request has been issued. Each request is issued with an identification number attached. If the external agent returns response data along with this identification number, the processor verifies the returned data and request.

The out-of-order return mode supports the following functions.

- Two timing modes
  Select either pipeline mode or re-issuance mode.
- Response queue of up to five entries
  Up to one instruction and four data entries can be managed.

The request cycles, basic operation of the protocol, and events that generate requests in the out-of-order return mode are the same as those in the normal mode. For details of these, refer to **CHAPTER 13 SYSTEM INTERFACE (64-BIT BUS MODE)** and **CHAPTER 14 SYSTEM INTERFACE (32-BIT BUS MODE)**.

15.1.1 Timing mode

The out-of-order return mode has two timing modes: re-issuance mode and pipeline mode. These modes can be selected by using the EM0 bit of the Config register in CP0. In the out-of-order return mode, the setting of the EM1 bit of the Config register is ignored.

- Pipeline mode
  The pipeline mode is selected when the EM0 bit of the Config register is cleared to 0.
  In this mode, even if the RdRdy#/WrRdy# signal is deasserted in the address cycle of a request, it is assumed that the request has been acknowledged.

- Re-issuance mode
  The re-issuance mode is selected when the EM0 bit of the Config register is set to 1.
  In this mode, a request is discarded if the RdRdy#/WrRdy# signal is deasserted in the address cycle of the request, and the same request is re-issued when the RdRdy#/WrRdy# signal is asserted.

15.1.2 Master status and slave status

In the out-of-order return mode, the system interface changes its status from master to slave in the following cases.

- When the maximum five requests are stored in the response queue and the processor has no write request to issue.
- The processor has no requests after it has issued a read request.

Remark The processor cannot issue a request in the following cases.

- When the processor has no requests.
- When the processor has a read request but the RdRdy# signal is inactive.
- When the processor has a write request but the WrRdy# signal is inactive.

When the system interface enters the slave status, the Release# signal is asserted. Therefore, the external agent must wait until the Release# signal is asserted, and then obrain the right to control the system interface to start driving response data.

Even when the system interface is in the slave status, the processor can request the right to control the system interface by asserting the PReq# signal.

When the active level of the PReq# signal is detected, the external agent can return the right to control the system interface to the processor by issuing a null request. At this time, the RdRdy#/WrRdy# signal must also be asserted, so that the processor can issue the subsequent request. If the RdRdy#/WrRdy# signal remains inactive, the system interface enters the slave status again even if it has entered the master status when the external agent issues the null request, without the processor issuing a request.

Even if the maximum five requests are stored in the response queue, the PReq# signal is asserted if read/write requests are accumulated in the processor. The external agent must process the processor requests by issuing a null request before the number of requests waiting for a request reaches five. Even if the external agent issues a null request when five requests are waiting for a response, processing of the requests does not proceed, and only the right to control the system interface is transferred.

15.1.3 Identifying request

The VR5500 uses the SysID(2:0) signals to identify the contents of a read request issued in the out-of-order return mode. The SysID0 signal indicates whether reading an instruction or data is requested, and the SysID(2:1) signals indicate the request sequence (number). When reading an instruction is requested, the SysID(2:1) signals are always 00 (for details, refer to **15.4 Request Identifier**).

The status of the SysID(2:0) signals is undefined when a write request is made.

15.2 Protocol of Out-of-Order Return Mode

This section explains the protocol of out-of-order return in the 64-bit bus mode. When using the 32-bit bus mode, read the SysAD bus width as 32 bits.

The data shown in Table 15-1 is driven onto the SysAD, SysCmd, and SysID buses. The symbols in this table are used in the timing chart shown later.

Table 15-1. System Interface Bus Data

Range	Symbol	Meaning
Common	Unsd	Unused
SysAD(64:0)	Addr<n>	Physical address of ID<n> request
	Data<n><m>	(m+1th element of) data of request of ID<n>
SysCmd(8:0)	Read	Read request command of processor or external agent
	Write	Write request command of processor or external agent
	Null	External null request command
	EOD	Data identifier of last data element
	Data	Data identifier of data element other than last data element
SysID(2:0)	ID<n>	Read request identifier

15.2.1 Successive read requests

This section explains the protocol used in each mode when three processor read requests are issued in a row.

(1) When processor read/write request follows in pipeline mode

In the pipeline mode, the external agent must acknowledge a request even if the RdRdy# signal goes high in the address cycle.

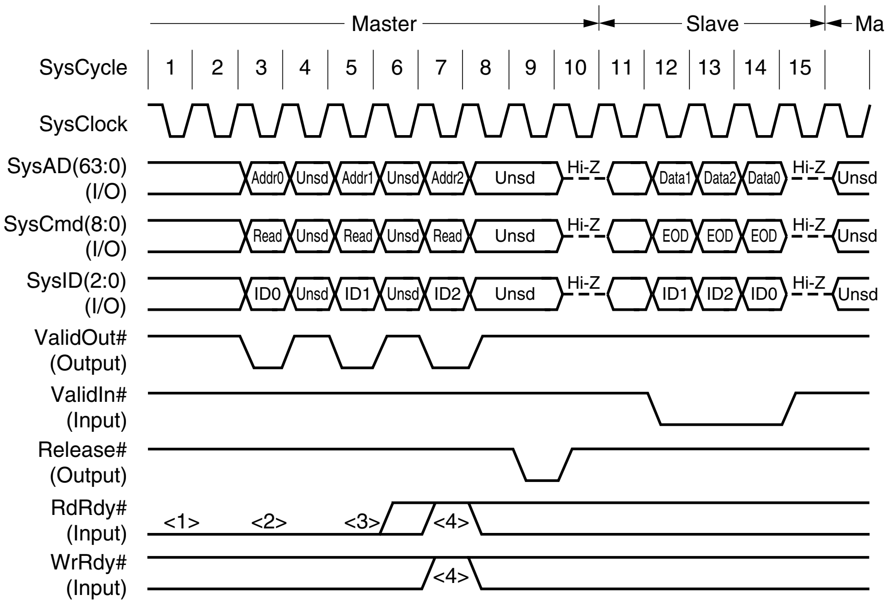
<1> to <3> in Figure 15-1 indicate that the external agent makes the RdRdy# signal low, indicating that it is ready to acknowledge a read request.

In response, the processor successively issues read requests in <2> to <4>. At this time, request identifiers are also driven onto the SysID bus.

In <4>, the external agent makes the RdRdy#/WrRdy# signal high, indicating that it can acknowledge no more read/write requests. However, the processor assumes that the request in the address cycle <4> has been acknowledged.

The external agent can return a response from a request for which data has been prepared. When driving response data, also drive the corresponding request identifier onto the SysID bus.

Figure 15-1. Successive Read Requests (in Pipeline Mode, with Subsequent Request)



(2) When processor read/write request does not follow in pipeline mode

In the pipeline mode, the external agent must acknowledge a request even if the RdRdy# signal goes high in the address cycle.

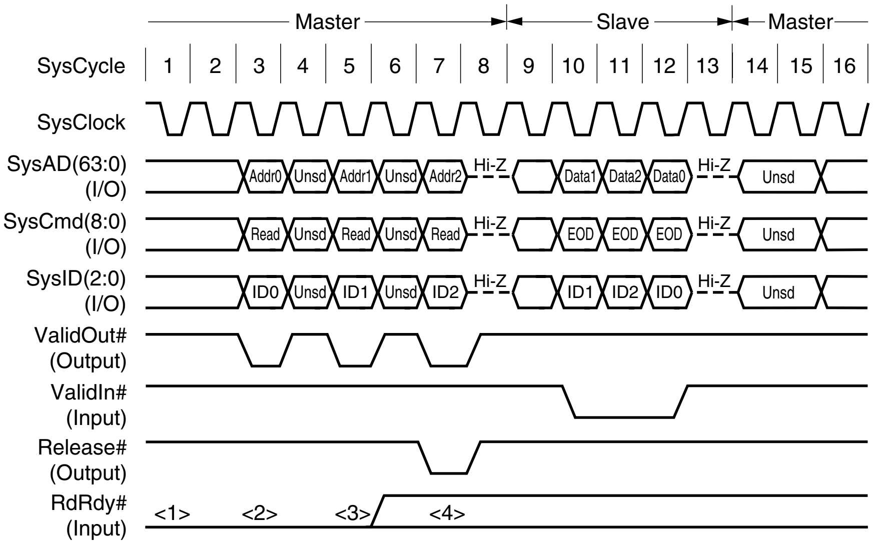
<1> to <3> in Figure 15-2 indicate that the external agent makes the RdRdy# signal low, indicating that it is ready to acknowledge a read request.

In response, the processor successively issues read requests in <2> to <4>. At this time, request identifiers are also driven onto the SysID bus.

Even if the external agent makes the RdRdy# signal high in the address cycle <4>, indicating that it cannot acknowledge a read request, the processor assumes that this request has been acknowledged.

The external agent can return a response from a request for which data has been prepared. When driving response data, also drive the corresponding request identifier onto the SysID bus.

Figure 15-2. Successive Read Requests (in Pipeline Mode, Without Subsequent Request)



(3) In re-issuance mode

If the RdRdy# signal goes high in the address cycle in the re-issuance mode, the processor discards the request and re-issues it when it returns to the master status.

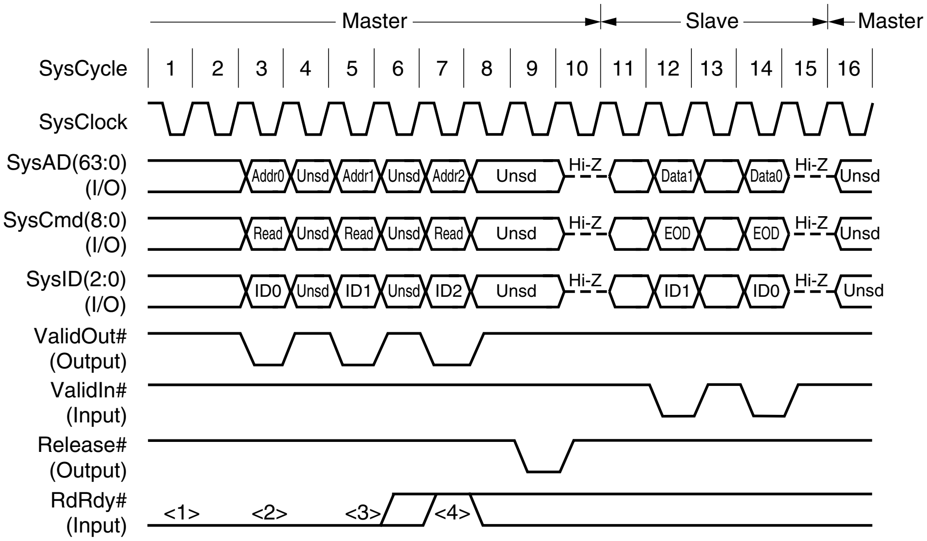
<1> to <3> in Figure 15-3 indicate that the external agent makes the RdRdy# signal low, indicating that it is ready to acknowledge a read request.

In response, the processor successively issues read requests in <2> to <4>. At this time, request identifiers are also driven onto the SysID bus.

If the external agent makes the RdRdy# signal high in the address cycle <4>, indicating that it cannot acknowledge a read request, the processor discards this request. When the processor later returns to the master status, it re-issues the request.

The external agent can return a response from a request for which data has been prepared. When driving response data, also drive the corresponding request identifier onto the SysID bus.

Figure 15-3. Successive Read Requests (in Re-Issuance Mode)



15.2.2 Successive write requests

This section explains the protocol used in each mode when processor write requests are issued in a row.

(1) In pipeline mode

In the pipeline mode, the external agent must acknowledge a request even if the WrRdy# signal goes high in the address cycle.

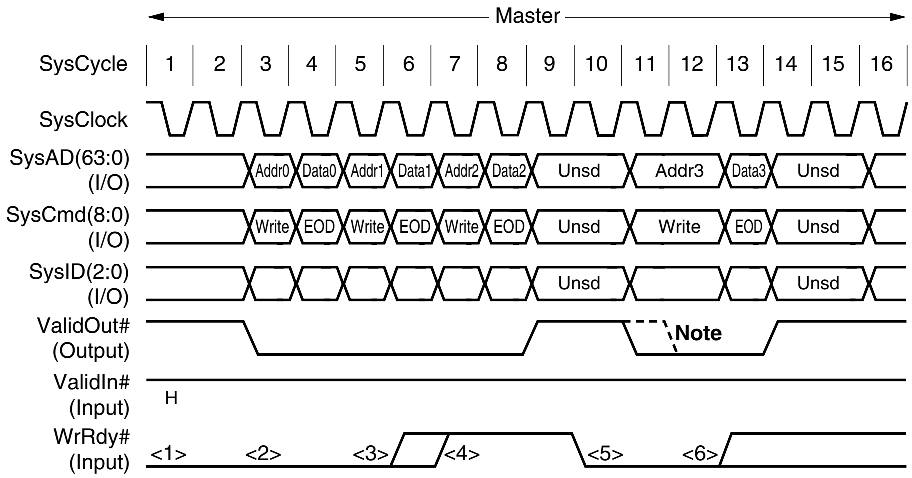
<1> to <3> in Figure 15-4 indicate that the external agent makes the WrRdy# signal low, indicating that it is ready to acknowledge a write request.

In response, the processor successively issues write requests in <2> to <4>. At this time, the status of the SysID bus is undefined.

Even if the external agent makes the WrRdy# signal high in the address cycle <4>, indicating that it cannot acknowledge a write request, the processor assumes that this request has been acknowledged.

When the external agent makes the WrRdy# signal low in <5>, the processor completes issuance of the write request in <6>.

Figure 15-4. Successive Write Requests (in Pipeline Mode)



Note When the DisDValidO# signal is low level

(2) In re-issuance mode

If the WrRdy# signal goes high in the address cycle in the re-issuance mode, the processor discards the request and re-issues it when the WrRdy# signal goes low.

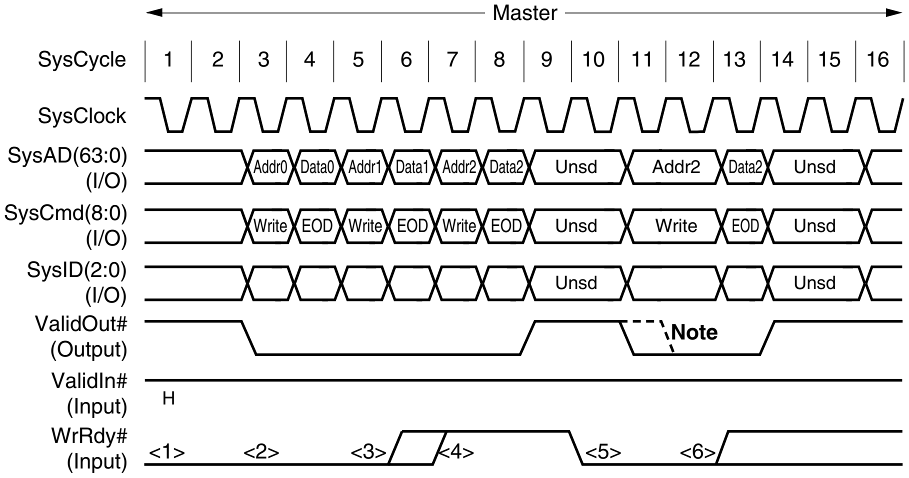
<1> to <3> in Figure 15-5 indicate that the external agent makes the WrRdy# signal low, indicating that it is ready to acknowledge a write request.

In response, the processor successively issues write requests in <2> to <4>. At this time, the status of the SysID bus is undefined.

If the external agent makes the WdRdy# signal high in the address cycle <4>, indicating that it cannot acknowledge a write request, the processor discards this request.

When the external agent makes the WrRdy# signal low in <5>, the processor re-issues in <6> the request discarded in <4>, and completes issuance of the write request.

Figure 15-5. Successive Write Requests (in Re-Issuance Mode)



Note When the DisDValidO# signal is low level

15.2.3 Write request following read request

This section explains the protocol when a processor write request is issued immediately after a processor read request.

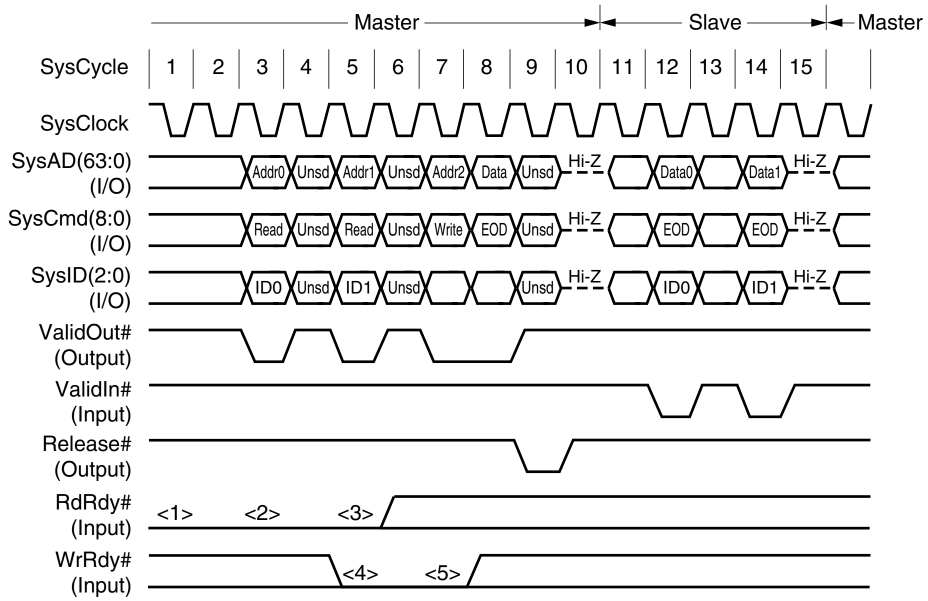
<1> and <2> in Figure 15-6 indicate that the external agent makes the RdRdy# signal low, indicating that it is ready to acknowledge a read request.

In response, the processor successively issues read requests in <2> and <3>. At this time, the request identifier is also driven onto the SysID bus.

In <4>, the external agent makes the WrRdy# signal low, indicating that it is ready to acknowledge a write request.

In response, the processor issues a write request in <5>. At this time, the status of the SysID bus is undefined.

Figure 15-6. Write Request Following Read Request



15.2.4 Bus arbitration of processor

This section explains the protocol in each mode when an external read response is aborted by asserting the PReq# signal.

(1) When processor read/write request follows in pipeline mode

In the pipeline mode, the external agent must acknowledge a request even if the RdRdy# signal goes high in the address cycle.

<1> and <2> in Figure 15-7 indicate that the external agent makes the RdRdy# signal low, indicating that it is ready to acknowledge a read request.

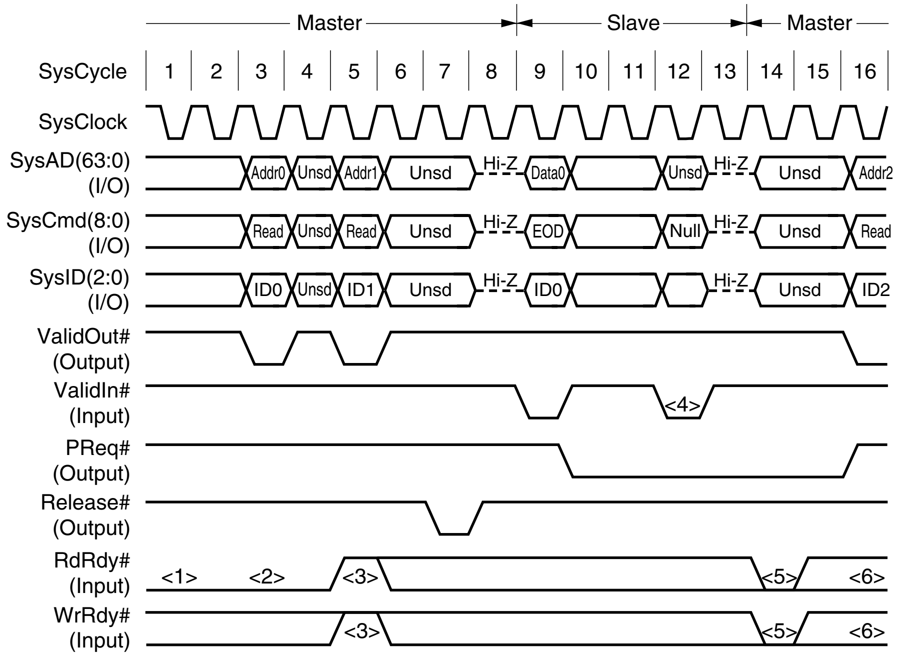
In response, the processor successively issues read requests in <2> and <3>. At this time, request identifiers are also driven onto the SysID bus.

In <3>, the external agent makes the RdRdy#/WrRdy# signal high, indicating that it can acknowledge no more read/write requests. However, the processor assumes that the request in the address cycle <3> has been acknowledged.

If the processor makes the PReq# signal low while a response cycle is delayed because it takes time to prepare response data, the external agent can issue a null request (<4>) and return the right to control the system interface to the processor. By transferring the right of control in this way before the number of requests waiting for a response reaches five, requests can be efficiently processed.

When the external agent makes the RdRdy#/WrRdy# signal low in <5>, the processor completes issuance of the read/write request in <6>.

Figure 15-7. Bus Arbitration of Processor (in Pipeline Mode, with Subsequent Request)



(2) When processor read/write request does not follow in pipeline mode

In the pipeline mode, the external agent must acknowledge a request even if the RdRdy# signal goes high in the address cycle.

<1> and <2> in Figure 15-8 indicate that the external agent makes the RdRdy# signal low, indicating that it is ready to acknowledge a read request.

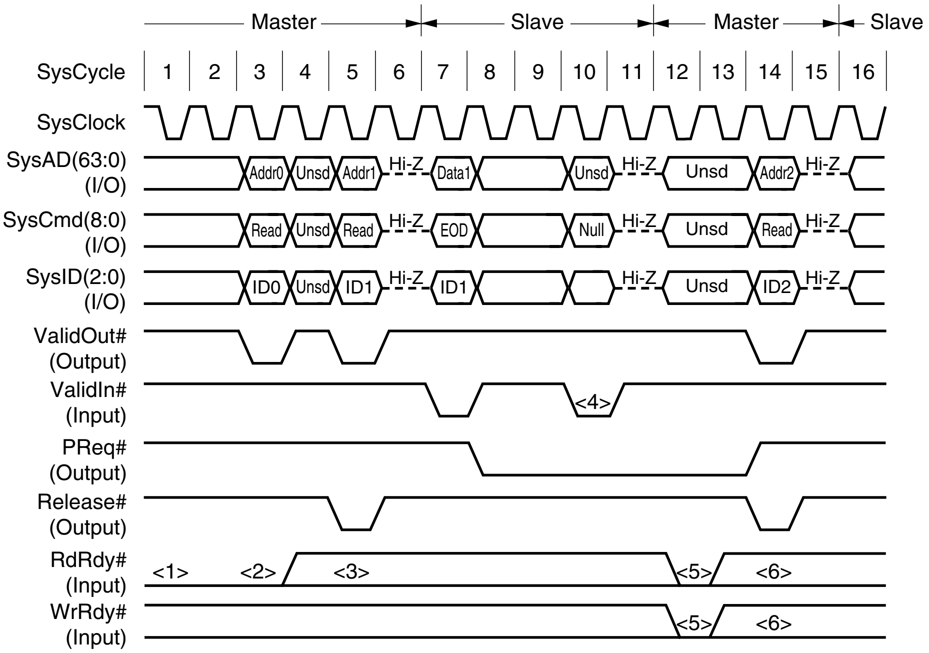
In response, the processor successively issues read requests in <2> and <3>. At this time, request identifiers are also driven onto the SysID bus.

Even if the external agent makes the RdRdy#/WrRdy# signal high in the address cycle <3>, indicating that it cannot acknowledge a read/write request, the processor assumes that this request has been acknowledged.

If the processor makes the PReq# signal low while a response cycle is delayed because it takes time to prepare response data, the external agent can issue a null request (<4>) and return the right to control the system interface to the processor. By transferring the right of control in this way before the number of requests waiting for a response reaches five, requests can be efficiently processed.

When the external agent makes the RdRdy#/WrRdy# signal low in <5>, the processor completes issuance of the read/write request in <6>.

Figure 15-8. Bus Arbitration of Processor (in Pipeline Mode, Without Subsequent Request)



(3) In re-issuance mode

If the RdRdy# signal goes high in the address cycle in the re-issuance mode, the processor discards the request and re-issues it when it returns to the master status.

<1> and <2> in Figure 15-9 indicate that the external agent makes the RdRdy# signal low, indicating that it is ready to acknowledge a read request.

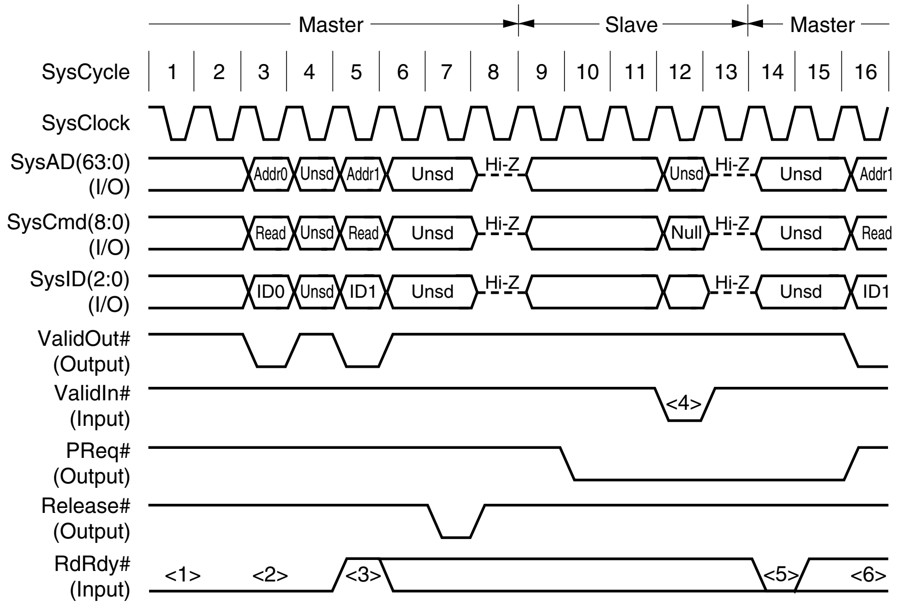
In response, the processor successively issues read requests in <2> and <3>. At this time, request identifiers are also driven onto the SysID bus.

If the external agent makes the RdRdy# signal high in the address cycle <3>, indicating that it cannot acknowledge a read request, the processor discards this request.

If the processor makes the PReq# signal low while a response cycle is delayed because it takes time to prepare response data, the external agent can issue a null request (<4>) and return the right to control the system interface to the processor. By transferring the right of control in this way before the number of requests waiting for a response reaches five, requests can be efficiently processed.

When the external agent makes the RdRdy# signal low in <5>, the processor completes issuance of the read request in <6>.

Figure 15-9. Bus Arbitration of Processor (in Re-Issuance Mode)



15.2.5 Single read request following block read request

This section explains the protocol in each mode when a processor single read request is issued immediately after a processor block read request.

(1) When processor read/write request follows in pipeline mode

In the pipeline mode, the external agent must acknowledge a request even if the RdRdy# signal goes high in the address cycle.

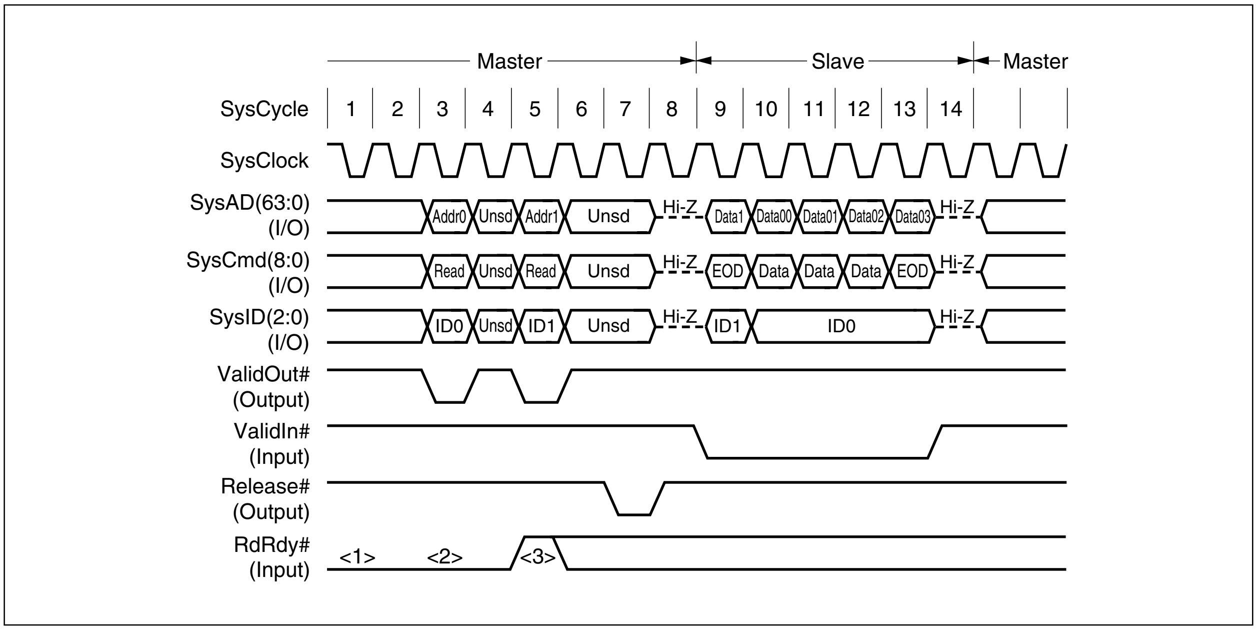
<1> and <2> in Figure 15-10 indicate that the external agent makes the RdRdy# signal low, indicating that it is ready to acknowledge a read request.

In response, the processor successively issues read requests in <2> and <3>. At this time, request identifiers are also driven onto the SysID bus.

Even if the external agent makes the RdRdy# signal high in the address cycle <3>, indicating that it cannot acknowledge a read request, the processor assumes that this request has been acknowledged.

The external agent can return a response from a request for which data has been prepared. When driving response data, also drive the corresponding request identifier onto the SysID bus.

Figure 15-10. Single Read Request Following Block Read Request (in Pipeline Mode, with Subsequent Request)



(2) When processor read/write request does not follow in pipeline mode

In the pipeline mode, the external agent must acknowledge a request even if the RdRdy# signal goes high in the address cycle.

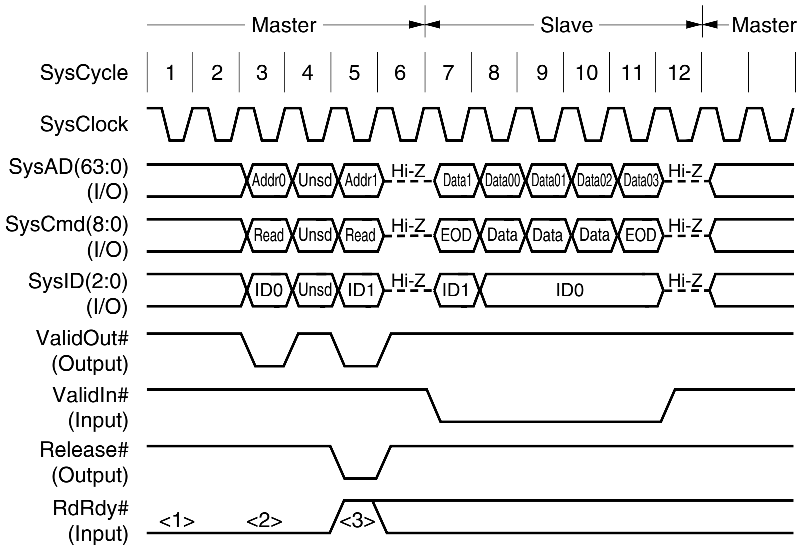
<1> and <2> in Figure 15-11 indicate that the external agent makes the RdRdy# signal low, indicating that it is ready to acknowledge a read request.

In response, the processor successively issues read requests in <2> and <3>. At this time, request identifiers are also driven onto the SysID bus.

Even if the external agent makes the RdRdy# signal high in the address cycle <3>, indicating that it cannot acknowledge a read request, the processor assumes that this request has been acknowledged.

The external agent can return a response from a request for which data has been prepared. When driving response data, also drive the corresponding request identifier onto the SysID bus.

Figure 15-11. Single Read Request Following Block Read Request (in Pipeline Mode, Without Subsequent Request)



(3) In re-issuance mode

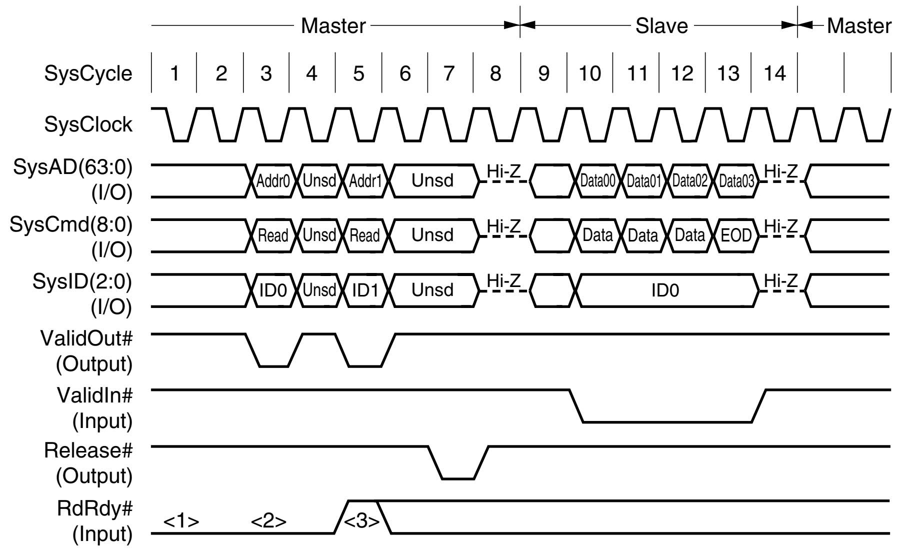
If the RdRdy# signal goes high in the address cycle in the re-issuance mode, the processor discards the request and re-issues it when it returns to the master status.

<1> and <2> in Figure 15-12 indicate that the external agent makes the RdRdy# signal low, indicating that it is ready to acknowledge a read request.

In response, the processor successively issues read requests in <2> and <3>. At this time, request identifiers are also driven onto the SysID bus.

If the external agent makes the RdRdy# signal high in the address cycle <4>, indicating that it cannot acknowledge a read request, the processor discards this request.

Figure 15-12. Single Read Request Following Block Read Request (in Re-Issuance Mode)



15.2.6 Unaligned 2-word read request

This section explains the protocol when a read request of unaligned 2-word data is issued in the 32-bit bus mode.

Remark Unaligned 2-word data is data of 5 to 8 bytes that is divided into 1 word and 1 to 4 bytes when processed.

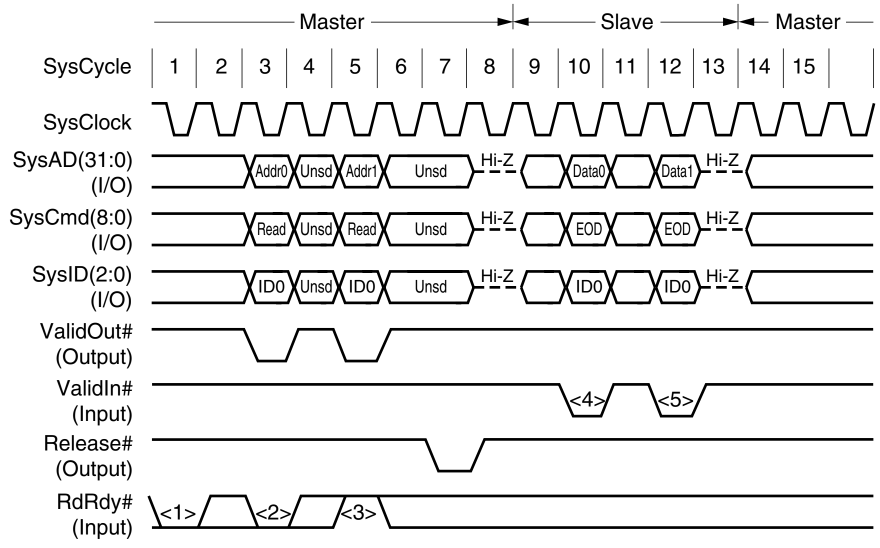
To read unaligned 2-word data, two read requests are successively issued, and the same request identifier is driven onto the SysID bus. The external agent must return response data in the same sequence as the corresponding request.

In <1> and <2> in Figure 15-13, the external agent makes the RdRdy# signal low, indicating that it is ready to acknowledge a read request.

In response, the processor successively issues read requests in <2> and <3>. At this time, the same request identifier is driven twice onto the SysID bus.

In <4> and <5>, the external agent must return the response data for which data has been prepared in the same sequence as the requests. When the response data is driven, the corresponding request identifier must also be driven onto the SysID bus.

Figure 15-13. Unaligned 2-Word Read (in Pipeline Mode, with Subsequent Request)



15.3 System Interface Commands and Data Identifiers

A system interface command defines the type and attribute of a system interface request. This definition is indicated in the address cycle of a request.

The system interface data identifier defines the attribute of the data transferred in the system interface data cycle.

This section explains the syntax of the commands and data identifiers of the system interface (coding in bit units) in the out-of-order return mode.

Set the reserved bits and reserved area in the commands and data identifiers of the system interface related to external requests to 1.

The reserved bits and reserved area in the commands and data identifiers of the system interface related to processor requests are undefined.

15.3.1 Syntax of commands and data identifiers

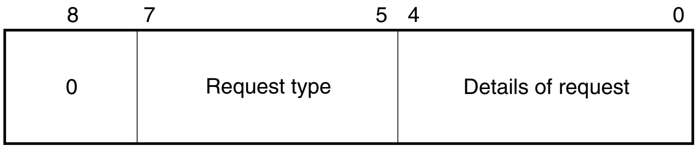
The commands and data identifiers of the system interface are coded in 9-bit units, and transferred from the processor to the external agent, or vice versa, via the SysCmd bus in the address cycle and data cycle.

SysCmd8 (most significant bit) determines whether the current contents of the SysCmd bus are a command (address cycle) or data identifier (data cycle). If they are a command, clear SysCmd8 to 0; if they are a data identifier, set it to 1.

15.3.2 Syntax of command

This section explains the coding of the SysCmd bus when a system interface command is used. Figure 15-14 shows the common code used for all the system interface commands.

Figure 15-14. Bit Definition of System Interface Command



Be sure to clear SysCmd8 to 0 when a system interface command is used.

SysCmd(7:5) define the types of system interface requests such as read, write, and null.

Table 15-2. Code of System Interface Command SysCmd(7:5)

Bit	Contents
SysCmd(7:5)	Command 0: Read request 1: Reserved 2: Write request 3: Null request 4 to 7: Reserved

SysCmd(4:0) are determined according to the type of request. A definition of each request is given below.

(1) Read request

The code of the SysCmd bus related to a read request is shown below.

Figure 15-15 shows the format of the command when a read request is issued.

Tables 15-3 to 15-5 show the code of the read attribute of the SysCmd(4:0) bits related to the read request.

Figure 15-15. Bit Definition of SysCmd Bus During Read Request

8	7 5	4 0
0	000	Details of read request (refer to the tables below)

Table 15-3. Code of SysCmd(4:3) During Read Request

(a) In 64-bit bus mode

Bit	Contents
SysCmd(4:3)	Read attribute 0: Reserved 1: Reserved 2: Block read 3: Single read

(b) In 32-bit bus mode

Bit	Contents
SysCmd(4:3)	Read attribute 0: Reserved 1: Unaligned 2-word read[Note] 2: Block read 3: Single read

Note When an unaligned 2-word read request is issued, the processor drives the same request identifier twice onto the SysID bus. The external agent must return the response data to the unaligned 2-word read request in the same sequence as the request.

Table 15-4. Code of SysCmd(2:0) During Block Read Request

(a) In 64-bit bus mode

Bit	Contents
SysCmd2	Reserved
SysCmd(1:0)	Size of read block 0: Reserved 1: 8 words 2, 3: Reserved

(b) In 32-bit bus mode

Bit	Contents
SysCmd2	Reserved
SysCmd(1:0)	Size of read block 0: 2 words (only when the DWBTrans# signal is low level) 1: 8 words 2, 3: Reserved

Table 15-5. Code of SysCmd(2:0) During Single Read Request

(a) In 64-bit bus mode

Bit	Contents
SysCmd(2:0)	Read data size 0: 1 byte is valid (byte). 1: 2 bytes are valid (halfword). 2: 3 bytes are valid. 3: 4 bytes are valid (word). 4: 5 bytes are valid. 5: 6 bytes are valid. 6: 7 bytes are valid. 7: 8 bytes are valid (doubleword).

(b) In 32-bit bus mode

Bit	Contents
SysCmd2	Reserved
SysCmd(1:0)	Read data size 0: 1 byte is valid (byte). 1: 2 bytes are valid (halfword). 2: 3 bytes are valid. 3: 4 bytes are valid (word).

(2) Write request

The code of the SysCmd bus related to a write request is shown below.

Figure 15-16 shows the format of the command when a write request is issued.

Tables 15-6 to 15-8 show the code of the write attribute of the SysCmd(4:0) bits related to the write request.

Figure 15-16. Bit Definition of SysCmd Bus During Write Request

8	7 5	4 0
0	010	Details of write request (refer to the tables below)

Table 15-6. Code of SysCmd(4:3) During Write Request

(a) In 64-bit bus mode

Bit	Contents
SysCmd(4:3)	Write attribute 0: Reserved 1: Reserved 2: Block write 3: Single write

(b) In 32-bit bus mode

Bit	Contents
SysCmd(4:3)	Write attribute 0: Reserved 1: Unaligned 2-word write 2: Block write 3: Single write

Table 15-7. Code of SysCmd(2:0) During Block Write Request

(a) In 64-bit bus mode

Bit	Contents
SysCmd2	Update of cache line 0: Replaced 1: Retained
SysCmd(1:0)	Size of write block 0: Reserved 1: 8 words 2, 3: Reserved

(b) In 32-bit bus mode

Bit	Contents
SysCmd2	Update of cache line 0: Replaced 1: Retained
SysCmd(1:0)	Size of write block 0: 2 words (only when the DWBTrans# signal is low level) 1: 8 words 2, 3: Reserved

Table 15-8. Code of SysCmd(2:0) During Single Write Request

(a) In 64-bit bus mode

Bit	Contents
SysCmd(2:0)	Write data size 0: 1 byte is valid (byte). 1: 2 bytes are valid (halfword). 2: 3 bytes are valid. 3: 4 bytes are valid (word). 4: 5 bytes are valid. 5: 6 bytes are valid. 6: 7 bytes are valid. 7: 8 bytes are valid (doubleword).

(b) In 32-bit bus mode

Bit	Contents
SysCmd2	Reserved
SysCmd(1:0)	Write data size 0: 1 byte is valid (byte). 1: 2 bytes are valid (halfword). 2: 3 bytes are valid. 3: 4 bytes are valid (word).

(3) Null request

Figure 15-17 shows the format of the command when a null request is used.

Table 15-9 shows the code of the SysCmd(4:3) bits related to the null request.

For the null request, the SysCmd(2:0) bits are reserved.

Figure 15-17. Bit Definition of SysCmd Bus During Null Request

8	7 5	4 0
0	011	Details of null request (refer to the table below)

Table 15-9. Code of SysCmd(4:3) During Null Request

Bit	Contents
SysCmd(4:3)	Null attribute 0: Released 1 to 3: Reserved

15.3.3 Syntax of data identifier

This section explains coding of the SysCmd bus when a system interface data identifier is used.

Figure 15-18 shows the common code used for all system interface data identifiers.

Figure 15-18. Bit Definition of System Interface Data Identifier

8	7	6	5	4	3 0
1	Indication of last data	Indication of response data	Indication of error data	Data check enable	Reserved

Be sure to set SysCmd8 of the system interface data identifier to 1.

A definition of the SysCmd(7:0) bits is given below.

SysCmd7: Indicates whether the data element is the last one.

SysCmd6: Indicates whether the data is response data. Response data is returned in response to a read request.

SysCmd5: Indicates whether the data element contains an error. The error indicated in the data cannot be corrected. If this data is returned to the processor, a bus error exception occurs. In the case of a response block, send the entire line to the processor regardless of the degree of error. The external agent should ignore this bit in a processor data identifier because no error is indicated.

SysCmd4: This bit in an external data identifier indicates whether the data of the data element and check bit are checked. This bit in a processor data identifier is reserved.

SysCmd(3:0): These bits are reserved.

Table 15-10 indicates the codes of SysCmd(7:5) of a processor data identifier, and Table 15-11 shows the codes of SysCmd(7:4) of an external data identifier.

Table 15-10. Codes of SysCmd(7:5) of Processor Data Identifier

Bit	Contents
SysCmd7	Indication of last data element 0: Last data element 1: Not last data element
SysCmd6	Indication of response data 0: Response data 1: Not response data
SysCmd5	Indication of error data 0: Error occurred 1: No error occurred

Table 15-11. Codes of SysCmd(7:4) of External Data Identifier

Bit	Contents
SysCmd7	Indication of last data element 0: Last data element 1: Not last data element
SysCmd6	Indication of response data 0: Response data 1: Not response data
SysCmd5	Indication of error data 0: Error occurred 1: No error occurred
SysCmd4	Data check enables 0: Data and check bit checked 1: Data and check bit not checked

Remark To enable data check, clear the DE bit of the Status register in CP0 to 0.

15.4 Request Identifier

In the out-of-order return mode, the processor drives a request identifier onto the SysID bus.

The request identifier defines the target of the read request and sequence of issuance (ID number). This definition is indicated in the address cycle of the request. The SysID bus is in an undefined state when a write request is issued.

SysID0 (least significant bit) determines whether the data targeted to the current request is an instruction or data.

SysID(2:1) defines the ID number of the read request.

Tables 15-12 to 15-14 show the code of the request identifier.

Table 15-12. Code of Request Identifier SysID0

Bit	Contents
SysID0	Request target 0: Instruction 1: Data

Table 15-13. Code of SysID(2:1) During Instruction Read

Bit	Contents
SysID(2:1)	Request issuance sequence 0: ID0 (first) 1 to 3: Reserved

Table 15-14. Code of SysID(2:1) During Data Read

Bit	Contents
SysID(2:1)	Request issuance sequence 0: ID0 (first) 1: ID1 (second) 2: ID2 (third) 3: ID3 (fourth)

This chapter explains the following four types of interrupts in the VR5500.

(1) Non-maskable interrupt (NMI): 1 source
(2) External ordinary interrupt: 6 sources (of which one is exclusive with a timer interrupt)
(3) Software interrupt: 2 sources
(4) Timer interrupt: 1 source (which is exclusive with one external ordinary interrupt)

16.1 Interrupt Request Type

16.1.1 Non-maskable interrupt (NMI)

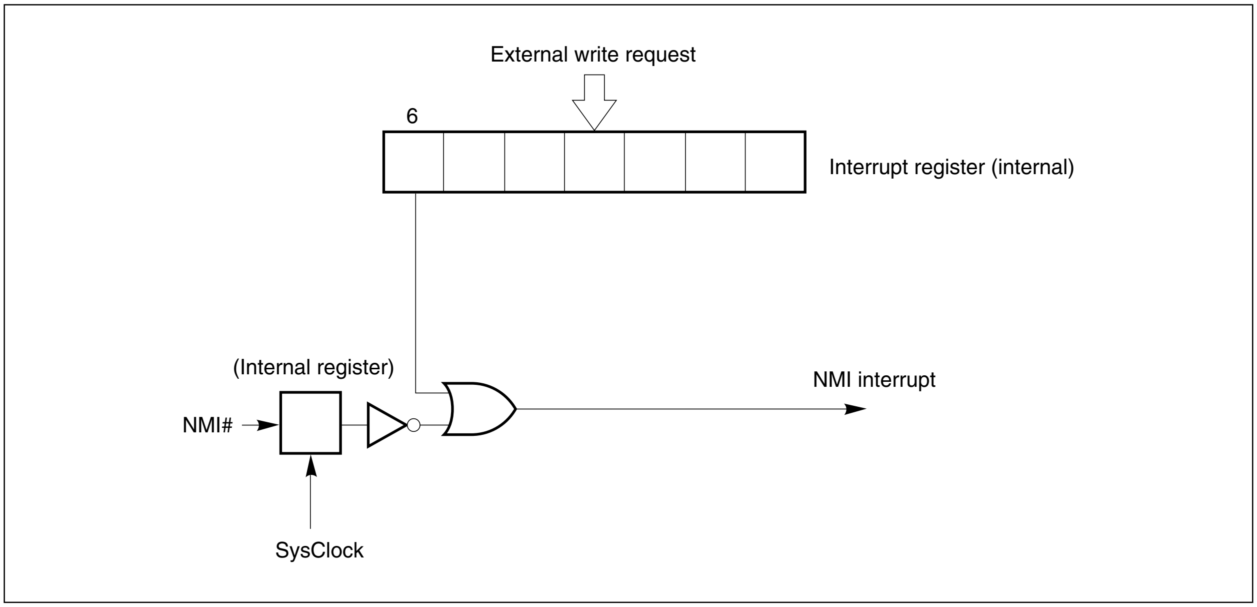
The NMI request is acknowledged when the NMI# signal is asserted, and execution branches to the reset exception vector. The NMI# signal is latched by an internal register at the rising edge of the SysClock signal as shown in Figure 16-1. This signal is edge-triggered.

This interrupt request can also be set by an external write request via the SysAD bus. In the data cycle, SysAD6 serves as an NMI request bit (1: Request), and SysAD22 serves as the write enable bit (1: Enable) corresponding to SysAD6.

An NMI cannot be masked.

Figure 16-1 shows the internal processing of the NMI# signal. A low-level signal input to the NMI# pin is latched to an internal register at the rising edge of SysClock. The latched NMI# signal is inverted and ORed with bit 6 of the internal register, and transmitted to the internal units as an NMI request.

Figure 16-1. NMI# Signal



16.1.2 External ordinary interrupt

This interrupt is acknowledged when the Int(5:0)# signals are made low, which sets the IP(7:2) bits of the Cause register. The Int(5:0)# signals are level-triggered. Keep these signals low until an interrupt exception occurs. After the interrupt exception has occurred, make high the signals that were low by the time execution returns to the normal routine, or before multiple interrupts are enabled.

An external ordinary interrupt request can also be set by an external write request via the SysAD bus. In the data cycle, SysAD(5:0) serve as external interrupt request bits (1: Request), and SysAD(21:16) serve as write enable bits (1: Enable) corresponding to SysAD(5:0). After an interrupt exception has occurred, issue the external write request again before execution returns to the ordinary routine or multiple interrupts are enabled, and clear the corresponding bit of the interrupt register to 0.

The interrupt request executed by Int5# signal or SysAD5 is acknowledged exclusively to the timer interrupt. If a low level is input to TIntSel pin before a power-on reset, the interrupt request by Int5# or SysAD5 becomes valid.

An external ordinary interrupt request can be masked by the IM(7:2), IE, EXL, and ERL bits of the Status register.

16.1.3 Software interrupts

Software interrupt requests are acknowledged when bits 1 and 0 of the IP (interrupt pending) field in the Cause register are set. These must be written by software; there is no hardware mechanism to set or clear these bits.

After the occurrence of an interrupt exception, the corresponding bit of the IP field in the Cause register must be cleared (0) before returning to the ordinary routine or before multiple interrupts are enabled.

A software interrupt request can be masked by the IM(1:0), IE, EXL, and ERL bits of the Status register.

16.1.4 Timer interrupt

This interrupt request uses bit 7 in the IP (interrupt pending) area of the Cause register. The IP7 bit is automatically set and the interrupt request is acknowledged if the value of the Count register becomes equal to that of the Compare register or if the performance counter overflows.

The timer interrupt is acknowledged exclusively to the interrupt request executed by the Int5# signal or SysAD5. If a high level is input to TIntSel pin before power-on reset, the timer interrupt request becomes valid.

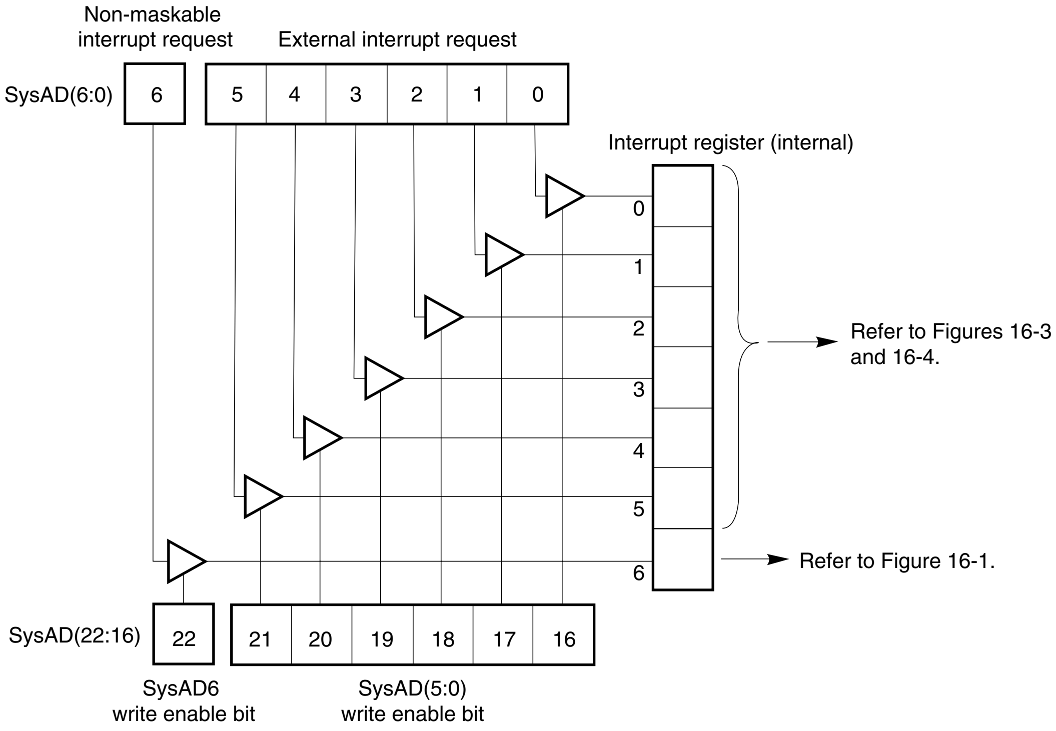
An timer interrupt request can be masked by the IM7, IE, EXL, and ERL bits of the Status register.

16.2 Acknowledging Interrupt Request Signal

If the external agent issues an external write request that makes SysAD(6:4) = 000, it is written to the interrupt register. This register can be used in the external write cycle but cannot be used in the external read cycle. When a request 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 registers.

In the data cycle, each bit of SysAD(22:16) enables a write access to the corresponding bit of the interrupt register, allowing the values of SysAD(6:0) to be written to the bits of the interrupt register. Therefore, bits 0 to 6 of the interrupt register can be set or cleared by issuing an external write request only once. This mechanism is illustrated in Figure 16-2, along with the NMI described above.

Figure 16-2. Bits of Interrupt Register and Enable Bits


Non-maskable interrupt request
External interrupt request
SysAD(6:0)
6
5
4
3
2
1
0
Interrupt register (internal)
0
1
2
3
4
5
6
Refer to Figures 16-3 and 16-4.
Refer to Figure 16-1.
SysAD(22:16)
22
21
20
19
18
17
16
SysAD6 write enable bit
SysAD(5:0) write enable bit

Bit	Function	Setting
SysAD(5:0)	External interrupt request	For each bit 1: Request 0: No request
SysAD(21:16)	Write enable bits of SysAD(5:0)	For each bit 1: Enabled 0: Disabled
SysAD6	Non-maskable interrupt request	1: Request 0: No request
SysAD22	Write enable bit of SysAD6	1: Enabled 0: Disabled

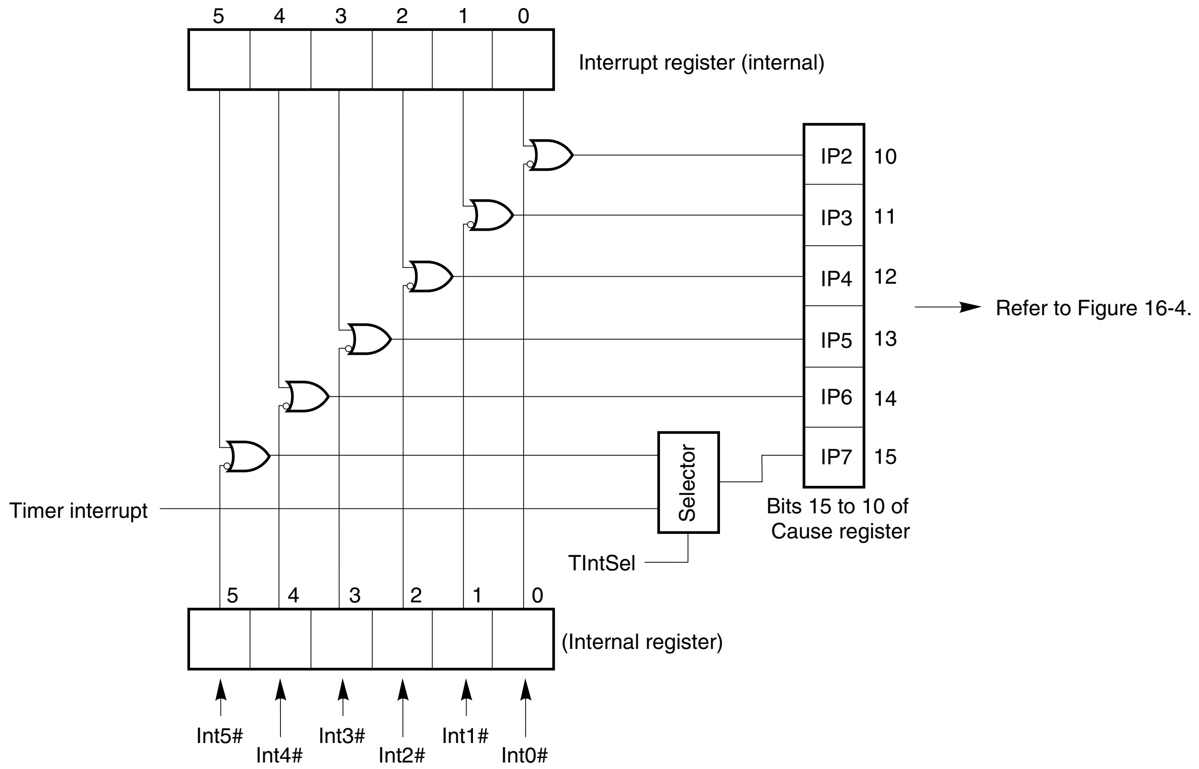
16.2.1 Detecting hardware interrupt

Figure 16-3 illustrates how a hardware interrupt request is detected by using the Cause register.

- Bit 15 (IP7) of the Cause register is directly checked for the timer interrupt request.
- Bits 15 to 10 (IP(7:2)) of the Cause register are directly checked for external ordinary interrupt requests (Int(5:0)# and SysAD(5:0)).
- Whether IP7 indicates the timer interrupt request or interrupt request executed by Int5# or SysAD5 is determined according to the status of the TIntSel pin before a power-on reset. If this pin is high, it indicates the timer interrupt. If it is low, it indicates the interrupt request executed by Int5# or SysAD5.

IP0 and IP1 of the Cause register are used for software interrupt requests (for details, refer to **CHAPTER 6 EXCEPTION PROCESSING**). Software interrupts cannot be set or cleared by hardware.

Figure 16-3. Hardware Interrupt Request Signal

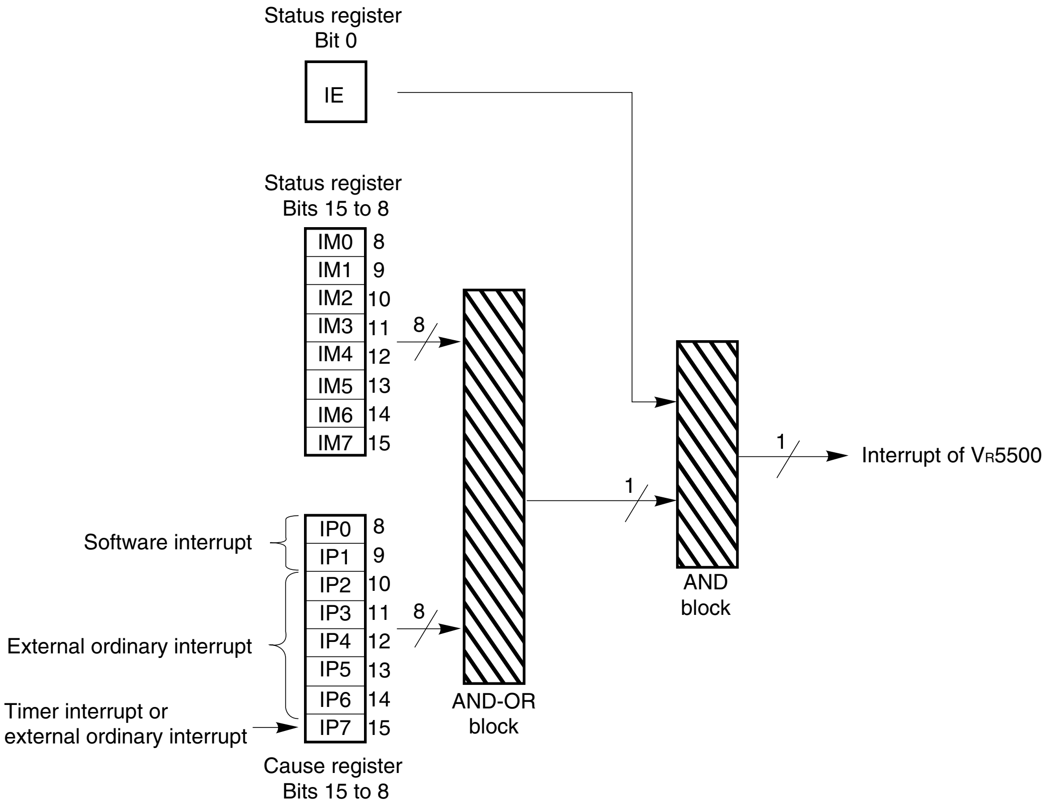


16.2.2 Masking interrupt signal

Figure 16-4 illustrates how an interrupt signal is masked.

- Bits 15 to 8 (IP(7:0)) of the Cause register are connected to the interrupt mask bits (bits 15 to 8, i.e., IM(7:0)) of the Status register by an AND-OR logic block, masking each interrupt request signal.
- Bit 0 of the Status register is a global interrupt enable (IE) bit. The output of this bit is ANDed with the output of the AND-OR logic block to generate the interrupt request signals of the VR5500. In addition, these interrupts are enabled by the EXL and ERL bits of the Status register.

Figure 16-4. Masking Interrupt Signal



Bit	Function	Setting
IE	Enables all interrupts.	1: Enables 0: Disables
IM(7:0)	Interrupt mask	For each bit 1: Enabled 0: Disabled
IP(7:0)	Interrupt request	For each bit 1: Request pending 0: Not pending

CHAPTER 17 CPU INSTRUCTION SET

This chapter provides a detailed description of the operation of the 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 18 FPU INSTRUCTION SET**.

17.1 Instruction Notation Conventions

In this chapter, all variable subfields in an instruction format (such as *rs*, *rt*, *immediate*, etc.) are shown in lowercase names. The instruction names (e.g. ADD and SUB) are indicated by upper-case characters. For the sake of clarity, we sometimes use an alias for a variable subfield in the formats of specific instructions. For example, we use *base* instead of *rs* in the format for load and store instructions. Such an alias is always lower case, since it refers to a variable subfield.

The architecture level at which the instruction was defined first is indicated on the right of the instruction format. The product name is also shown for instructions that may be incorporated differently depending on the product.

Figures with the actual bit encoding for all the mnemonics are located at the end of this chapter (**17.4 CPU Instruction Opcode Bit Encoding**), and the bit encoding also accompanies each instruction.

In the instruction descriptions that follow, the operation section describes the operation performed by each instruction using a high-level language notation. The VR5500 can operate as either a 32- or 64-bit microprocessor and the operation for both modes is included with the instruction description.

Special symbols used in the notation are described in Table 17-1.

Table 17-1. CPU Instruction Operation Notations

Symbol	Meaning
←	Assignment
||	Bit string concatenation
x^y	Replication of bit value *x* into a *y*-bit string. *x* is always a single-bit value
$x_{y..z}$	Selection of bits *y* to *z* of bit string *x*. Little-endian bit notation is always used. If *y* is less than *z*, this expression is an empty (zero length) bit string
+	2’s complement or floating-point addition
–	2’s complement or floating-point subtraction
*	2’s complement or floating-point multiplication
div	2’s complement integer division
mod	2’s complement modulo
/	Floating-point division
<	2’s complement less than comparison
and	Bit-wise logical AND
or	Bit-wise logical OR
xor	Bit-wise logical XOR
nor	Bit-wise logical NOR
GPR[*x*]	General-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	Big-endian mode as configured at reset (0 → Little, 1 → Big). Specifies the endianness of the memory interface (see **Table 17-2 Load and Store Common Functions**), and the endianness in kernel and supervisor mode. The status of the BE bit of the Config register is reflected.
ReverseEndian	Signal to reverse the endianness of load and store instructions. The status of bit 25 of the Status register is reflected. This value is always 0 in the V$_{R}$5500.
BigEndianCPU	The endianness for load and store instructions (0 → Little, 1 → Big). This variable is computed as BigEndianMem XOR ReverseEndian.
T + *i*:	Indicates the time steps between operations. Each of the statements within a time step are defined to be executed in sequential order (as modified by conditional and loop constructs). Operations which are marked *T + i*: are executed at instruction cycle *i* relative to the start of execution of the instruction. Thus, an instruction which starts at time *j* executes operations marked T + *i:* at time *i + j*. The interpretation of the order of execution between two instructions or two operations that execute at the same time should be pessimistic; the order is not defined.

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

Example 1:

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

Sixteen zero bits are concatenated with an immediate value (typically 16 bits), and the 32-bit string is assigned to general-purpose register *rt*.

Example 2:

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

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

17.2 Cautions on Using CPU Instructions

17.2.1 Load and store instructions

The instruction immediately after a load instruction can use the contents of a register that has been loaded, but execution of that instruction may be delayed. The VR5500 can cover the load delay using an out-of-order mechanism, but it is recommended to schedule the load delay slot to improve the performance.

With the VR5500, two special instructions, a load link instruction and a conditional store instruction, can be used. However, these instructions are used in a carefully programmed sequence when one of the synchronous primitives (such as test & set, lock of bit level, semaphore, and sequencer/event counter) is executed. These instructions are defined in the VR5500 to maintain compatibility with the other processors.

In the load and store descriptions, the functions listed below are used to summarize the handling of virtual addresses and physical memory.

Table 17-2. Load and Store Common Functions

Function	Meaning
AddressTranslation	Uses the TLB to find the physical address given the virtual address. The function fails and a TLB refill exception occurs if the required translation is not present in the TLB.
LoadMemory	Uses the cache and main memory to find the contents of the word containing the specified physical address. The lower 6 bits of the address and the Access Type field indicate which of each of the four bytes within the data word need to be returned. If the cache is enabled for this access, the entire word is returned and loaded to the cache. If the specified data is short of word length, the data position to which the contents of the specified data is stored is determined considering the endian mode and reverse-endian mode.
StoreMemory	Uses the cache, write buffer, and main memory to store the word or part of word specified as data in the word containing the specified physical address. The lower 3 bits of the address and the Access Type field indicate which of each of the four bytes within the data word should be stored. If the specified data is short of word length, the data position to which the contents of the specified data is stored is determined considering the endian mode and reverse-endian mode.

The Access Type field indicates the size of the data item to be loaded or stored. Regardless of access type or byte-numbering order (endian), the address specifies the byte that has the smallest byte address in the addressed field. The access type field is the leftmost byte in a big-endian system, and includes a 2's complement sign value. This field is the rightmost byte in a little-endian system.

Table 17-3. Access Type Specifications for Loads/Stores

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 addressed doubleword that are used can be determined directly from the access type and the lower 3 bits of the address.

17.2.2 Jump and branch instructions

The jump and branch instructions have a branch delay slot. A jump or branch instruction cannot be used in a delay slot. If used, the error is not detected and the results of such an operation are undefined.

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

Because jump and branch instructions may be restarted after exceptions or interrupts, they must be restartable. Therefore, when a jump or branch instruction stores a return link value, CPU general-purpose register *r31* (the register in which the link is stored) may not be used as a source register.

Since instructions must be word-aligned, a Jump Register or Jump and Link Register instruction must use a register which contains a content (address) whose lower 2 bits are zero. If the lower 2 bits are not zero, an address error exception will occur when the jump target instruction is subsequently fetched.

17.2.3 Coprocessor instructions

The coprocessor is an alternate execution unit and has a register file independent of that of the CPU. The MIPS architecture allows four coprocessor units to be defined. Each of these coprocessors has two register spaces, and each register space has thirty-two 32-bit registers. The coprocessor instructions modify the registers in either of the spaces.

- Coprocessor general-purpose registers are allocated in the first space. These registers directly load/store data from/in the main memory. They can also be used to transfer data between coprocessors.
- Coprocessor control registers are allocated in the second space. These registers can transfer their contents only between coprocessors.

17.2.4 System control coprocessor (CP0) instructions

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

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

17.3 CPU Instruction

This section describes the functions of CPU instructions in detail for both 32-bit address mode and 64-bit address mode.

The exception that may occur by executing each instruction is shown in the last of each instruction's description. For details of exceptions and their processes, see **CHAPTER 6 EXCEPTION PROCESSING**.

ADD

Add

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

Format:

ADD rd, rs, rt

MIPS I

Purpose:

Adds 32-bit integers. A trap is performed if an overflow occurs.

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*. In 64-bit mode, the operands must be valid 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 destination register *rd* is not modified when an integer overflow exception occurs.

Operation:

```
32    T:    $GPR[rd] \leftarrow GPR[rs] + GPR[rt]$

64    T:    $temp \leftarrow GPR[rs] + GPR[rt]$
            $GPR[rd] \leftarrow (temp_{31})^{32} \,||\, temp_{31..0}$
```

Exceptions:

Integer overflow exception

ADDI

Add Immediate

31 … 26	25 … 21	20 … 16	15 … 0
ADDI 001000	rs	rt	immediate

Format:

ADDI rt, rs, immediate **MIPS I**

Purpose:

Adds a 32-bit integer to a constant. A trap is performed if an overflow occurs.

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.* In 64-bit mode, the operand must be valid sign-extended, 32-bit values.
An integer overflow exception occurs if carries out of bits 30 and 31 differ (2's complement overflow). The destination register *rt* is not modified when an integer overflow exception occurs.

Operation:

```
32    T:    $GPR[rt] \leftarrow GPR[rs] + (immediate_{15})^{16} \| immediate_{15..0}$

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

Exceptions:

Integer overflow exception

ADDIU

Add Immediate Unsigned

31 26	25 21	20 16	15 0
ADDIU 001001	rs	rt	immediate

Format:

ADDIU rt, rs, immediate **MIPS I**

Purpose:

Adds a 32-bit integer to a constant.

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.* No integer overflow exception occurs under any circumstances. In 64-bit mode, the operand must be valid sign-extended, 32-bit values.

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

Operation:

```
32    T:    $GPR[rt] \leftarrow GPR[rs] + (immediate_{15})^{16} \,||\, immediate_{15..0}$

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

Exceptions:

None

ADDU

Add Unsigned

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	ADDU 100001

Format:

ADDU rd, rs, rt

MIPS I

Purpose:

Adds 32-bit integers.

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*. No integer overflow exception occurs under any circumstances. In 64-bit mode, the operands must be valid sign-extended, 32-bit values.

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

Operation:

```
32    T:    GPR[rd] ← GPR[rs] + GPR[rt]

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

Exceptions:

None

AND

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	AND 100100

Format:

AND rd, rs, rt **MIPS I**

Purpose:

Performs a bit-wise logical AND operation.

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

31 26	25 21	20 16	15 0
ANDI 001100	rs	rt	immediate

Format:

ANDI rt, rs, immediate **MIPS I**

Purpose:

Performs a bit-wise logical AND operation with a constant.

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 and $GPR[rs]_{15..0}$)

64    T:    $GPR[rt] \leftarrow 0^{48}$ || (immediate and $GPR[rs]_{15..0}$)
```

Exceptions:

None

BC0F

Branch on Coprocessor 0 False

31 26	25 21	20 16	15 0
COP0 010000	BC 01000	BCF 00000	offset

Format:

BC0F offset

MIPS I

Purpose:

Tests the CP0 condition code and executes a PC relative condition branch.

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 contents of CP0's condition signal (CpCond), as sampled during the previous instruction, is false, then the program branches to the target address with a delay of one instruction. Because the condition line is sampled during the previous instruction, there must be at least one instruction between this instruction and a coprocessor instruction that changes the condition line.

Remark The condition branch range of this instruction is ±128 KB because an 18-bit signed offset is used. To branch to an address outside this range, use the J or JR instruction.

Operation:

```
32      T – 1:  condition ← not COP0
        T:      target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
        T + 1:  if condition then
                  PC ← PC + target
                endif


64      T – 1:  condition ← not COP0
        T:      target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
        T + 1:  if condition then
                  PC ← PC + target
                endif
```

Exceptions:

Coprocessor unusable exception

BC0FL

Branch on Coprocessor 0 False Likely

31 26	25 21	20 16	15 0
COP0 010000	BC 01000	BCFL 00010	offset

Format:

BC0FL offset

MIPS II

Purpose:

Tests the CP0 condition code and executes a PC relative condition branch. Executes a delay slot only when a given branch condition is satisfied.

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of CP0's condition (CpCond) line, as sampled during the previous instruction, is false, the target address is branched to with a delay of one instruction.
If the conditional branch is not taken, the instruction in the branch delay slot is discarded.

Remarks 1. The condition branch range of this instruction is ±128 KB because an 18-bit signed offset is used. To branch to an address outside this range, use the J or JR instruction.

2. Use this instruction only when it is expected with a high probability (98% or higher) that a given branch condition is satisfied. If the branch condition is not satisfied or if the branch destination is not known, use the BC0F instruction.

Operation:

```
32      T – 1:  condition ← not COP0
        T:      target ← $(offset_{15})^{14}$ || offset || $0^{2}$
        T + 1:  if condition then
                  PC ← PC + target
                else
                  NullifyCurrentInstruction
                endif


64      T – 1:  condition ← not COP0
        T:      target ← $(offset_{15})^{46}$ || offset || $0^{2}$
        T + 1:  if condition then
                  PC ← PC + target
                else
                  NullifyCurrentInstruction
                endif
```

Exceptions:

Coprocessor unusable exception

BC0T

Branch on Coprocessor 0 True

31 26	25 21	20 16	15 0
COP0 010000	BC 01000	BCT 00001	offset

Format:

BC0T offset

MIPS I

Purpose:

Tests the CP0 condition code and executes a PC relative condition branch.

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of CP0's condition signal (CpCond) that is sampled during the previous instruction is true, then the program branches to the target address, with a delay of one instruction.

Remark The condition branch range of this instruction is ±128 KB because an 18-bit signed offset is used. To branch to an address outside this range, use the J or JR instruction.

Operation:

```
32      T – 1:  condition ← COP0
        T:      target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
        T + 1:  if condition then
                  PC ← PC + target
                endif


64      T – 1:  condition ← COP0
        T:      target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
        T + 1:  if condition then
                  PC ← PC + target
                endif
```

Exceptions:

Coprocessor unusable exception

BC0TL

Branch on Coprocessor 0 True Likely

31 26	25 21	20 16	15 0
COP0 010000	BC 01000	BCTL 00011	offset

Format:

BC0TL offset

MIPS II

Purpose:

Tests the CP0 condition code and executes a PC relative condition branch. Executes a delay slot only when a given branch condition is satisfied.

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of CP0's condition (CpCond) line, as sampled during the previous instruction, is true, the target address is branched to with a delay of one instruction.
If the conditional branch is not taken, the instruction in the branch delay slot is discarded.

Remarks 1. The condition branch range of this instruction is ±128 KB because an 18-bit signed offset is used. To branch to an address outside this range, use the J or JR instruction.

2. Use this instruction only when it is expected with a high probability (98% or higher) that a given branch condition is satisfied. If the branch condition is not satisfied or if the branch destination is not known, use the BC0T instruction.

Operation:

```
32     T – 1:  condition ← COP0
       T:      target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
       T + 1:  if condition then
                 PC ← PC + target
               else
                 NullifyCurrentInstruction
               endif

64     T – 1:  condition ← COP0
       T:      target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
       T + 1:  if condition then
                 PC ← PC + target
               else
                 NullifyCurrentInstruction
               endif
```

Exceptions:

Coprocessor unusable exception

BEQ

Branch on Equal

31 26	25 21	20 16	15 0
BEQ 000100	rs	rt	offset

Format:

BEQ rs, rt, offset

MIPS I

Purpose:

Compares general-purpose registers and executes a PC relative condition branch.

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. The contents of general-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 target address, with a delay of one instruction.

Remark The condition branch range of this instruction is ±128 KB because an 18-bit signed offset is used. To branch to an address outside this range, use the J or JR instruction.

Operation:

```
32      T:       $target \leftarrow (offset_{15})^{14} \| offset \| 0^{2}$
                 $condition \leftarrow (GPR[rs] = GPR[rt])$
        T + 1:   if condition then
                   $PC \leftarrow PC + target$
                 endif


64      T:       $target \leftarrow (offset_{15})^{46} \| offset \| 0^{2}$
                 $condition \leftarrow (GPR[rs] = GPR[rt])$
        T + 1:   if condition then
                   $PC \leftarrow PC + target$
                 endif
```

Exceptions:

None

BEQL

Branch on Equal Likely

31 26	25 21	20 16	15 0
BEQL 010100	rs	rt	offset

Format:

BEQL rs, rt, offset

MIPS II

Purpose:

Compares general-purpose registers and executes a PC relative condition branch. Executes a delay slot only when a given branch condition is satisfied.

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit offset, shifted left two bits and sign-extended. The contents of general-purpose register *rs* and the contents of general-purpose register *rt* are compared. If the two registers are equal, the target address is branched to, with a delay of one instruction.

If the conditional branch is not taken, the instruction in the branch delay slot is discarded.

Remarks 1. The condition branch range of this instruction is ±128 KB because an 18-bit signed offset is used. To branch to an address outside this range, use the J or JR instruction.

2. Use this instruction only when it is expected with a high probability (98% or higher) that a given branch condition is satisfied. If the branch condition is not satisfied or if the branch destination is not known, use the BEQ instruction.

Operation:

```
32    T:      target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
              condition ← (GPR[rs] = GPR[rt])
      T + 1:  if condition then
                PC ← PC + target
              else
                NullifyCurrentInstruction
              endif


64    T:      target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
              condition ← (GPR[rs] = GPR[rt])
      T + 1:  if condition then
                PC ← PC + target
              else
                NullifyCurrentInstruction
              endif
```

Exceptions:

None

BGEZ

Branch on Greater Than or Equal to Zero

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BGEZ 00001	offset

Format:

BGEZ rs, offset

MIPS I

Purpose:

Tests a general-purpose register and executes a PC relative condition branch.

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of general-purpose register *rs* are zero or greater when compared to zero, then the program branches to the target address, with a delay of one instruction.

Remark The condition branch range of this instruction is ±128 KB because an 18-bit signed offset is used. To branch to an address outside this range, use the J or JR instruction.

Operation:

```
32    T:      target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
              condition ← $(\text{GPR[rs]}_{31} = 0)$
      T + 1:  if condition then
                PC ← PC + target
              endif

64    T:      target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
              condition ← $(\text{GPR[rs]}_{63} = 0)$
      T + 1:  if condition then
                PC ← PC + target
              endif
```

Exceptions:

None

BGEZAL

Branch on Greater Than or Equal to Zero and Link

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BGEZAL 10001	offset

Format:

BGEZAL rs, offset **MIPS I**

Purpose:

Tests a general-purpose register and executes a PC relative condition procedure call.

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset,* shifted left two bits and sign-extended. Unconditionally, the address of the instruction after the delay slot is stored in the link register, *r31*. If the contents of general-purpose register *rs* are zero or greater when compared to zero, then the program branches to the target address, with a delay of one instruction.

General-purpose register *r31* should not be specified as general-purpose register *rs*. If register *r31* is specified, restarting may be impossible due to the destruction of *rs* contents caused by storing a link address. Even such instructions are executed, an exception does not result.

Remark The condition branch range of this instruction is ±128 KB because an 18-bit signed offset is used. To branch to an address outside this range, use the J or JR instruction.

Operation:

```
32    T:      target ← $(\mathrm{offset}_{15})^{14}$ || offset || $0^{2}$
              condition ← $(\mathrm{GPR[rs]}_{31} = 0)$
              GPR[31] ← PC + 8
      T + 1:  if condition then
                PC ← PC + target
              endif


64    T:      target ← $(\mathrm{offset}_{15})^{46}$ || offset || $0^{2}$
              condition ← $(\mathrm{GPR[rs]}_{63} = 0)$
              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

(1/2)

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BGEZALL 10011	offset

Format:

BGEZALL rs, offset

MIPS II

Purpose:

Tests a general-purpose register and executes a PC relative condition procedure call. Executes a delay slot only when a given branch condition is satisfied.

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset,* shifted left two bits and sign-extended. Unconditionally, the address of the instruction after the delay slot is stored in the link register, *r31*. If the contents of general-purpose register *rs* are zero or greater when compared to zero, then the program branches to the target address, with a delay of one instruction.

If the conditional branch is not taken, the instruction in the branch delay slot is discarded.

General-purpose register *r31* should not be specified as general-purpose register *rs*. If register *r31* is specified, restarting may be impossible due to the destruction of *rs* contents caused by storing a link address. Even such instructions are executed, an exception does not result.

Remarks 1. The condition branch range of this instruction is ±128 KB because an 18-bit signed offset is used. To branch to an address outside this range, use the J or JR instruction.

2. Use this instruction only when it is expected with a high probability (98% or higher) that a given branch condition is satisfied. If the branch condition is not satisfied or if the branch destination is not known, use the BGEZAL instruction.

BGEZALL

Branch on Greater Than or Equal to Zero and Link Likely

(2/2)

Operation:

```
32    T:      target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
              condition ← $(\text{GPR[rs]}_{31} = 0)$
              GPR[31] ← PC + 8
      T + 1:  if condition then
                PC ← PC + target
              else
                NullifyCurrentInstruction
              endif

64    T:      target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
              condition ← $(\text{GPR[rs]}_{63} = 0)$
              GPR[31] ← PC + 8
      T + 1:  if condition then
                PC ← PC + target
              else
                NullifyCurrentInstruction
              endif
```

Exceptions:

None

BGEZL

Branch on Greater Than or Equal to Zero Likely

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BGEZL 00011	offset

Format:

BGEZL rs, offset

MIPS II

Purpose:

Tests a general-purpose register and executes a PC relative condition branch. Executes a delay slot only when a given branch condition is satisfied.

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of general-purpose register *rs* are zero or greater when compared to zero, then the program branches to the target address, with a delay of one instruction.
If the conditional branch is not taken, the instruction in the branch delay slot is discarded.

Remarks 1. The condition branch range of this instruction is ±128 KB because an 18-bit signed offset is used. To branch to an address outside this range, use the J or JR instruction.

2. Use this instruction only when it is expected with a high probability (98% or higher) that a given branch condition is satisfied. If the branch condition is not satisfied or if the branch destination is not known, use the BGEZ instruction.

Operation:

```
32    T:      target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
              condition ← $(\text{GPR[rs]}_{31} = 0)$
      T + 1:  if condition then
                PC ← PC + target
              else
                NullifyCurrentInstruction
              endif


64    T:      target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
              condition ← $(\text{GPR[rs]}_{63} = 0)$
      T + 1:  if condition then
                PC ← PC + target
              else
                NullifyCurrentInstruction
              endif
```

Exceptions:

None

BGTZ

Branch on Greater Than Zero

31 26	25 21	20 16	15 0
BGTZ 000111	rs	0 00000	offset

Format:

BGTZ rs, offset

MIPS I

Purpose:

Tests a general-purpose register and executes a PC relative condition branch.

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of general-purpose register *rs* are zero or greater when compared to zero, then the program branches to the target address, with a delay of one instruction.

Remark The condition branch range of this instruction is ±128 KB because an 18-bit signed offset is used. To branch to an address outside this range, use the J or JR instruction.

Operation:

```
32    T:      target ← $(\text{offset}_{15})^{14}$ || offset || $0^{2}$
              condition ← $(\text{GPR[rs]}_{31} = 0)$ and $(\text{GPR[rs]} \neq 0^{32})$
      T + 1:  if condition then
                PC ← PC + target
              endif


64    T:      target ← $(\text{offset}_{15})^{46}$ || offset || $0^{2}$
              condition ← $(\text{GPR[rs]}_{63} = 0)$ and $(\text{GPR[rs]} \neq 0^{64})$
      T + 1:  if condition then
                PC ← PC + target
              endif
```

Exceptions:

None

BGTZL

Branch on Greater Than Zero Likely

31 26	25 21	20 16	15 0
BGTZL 010111	rs	0 00000	offset

Format:

BGTZL rs, offset

MIPS II

Purpose:

Tests a general-purpose register and executes a PC relative condition branch. Executes a delay slot only when a given branch condition is satisfied.

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. The contents of general-purpose register *rs* are compared to zero. If the contents of general-purpose register *rs* are greater than zero, then the program branches to the target address, with a delay of one instruction.

If the conditional branch is not taken, the instruction in the branch delay slot is discarded.

Remarks 1. The condition branch range of this instruction is ±128 KB because an 18-bit signed offset is used. To branch to an address outside this range, use the J or JR instruction.

2. Use this instruction only when it is expected with a high probability (98% or higher) that a given branch condition is satisfied. If the branch condition is not satisfied or if the branch destination is not known, use the BGTZ instruction.

Operation:

```
32    T:      target ← $(\text{offset}_{15})^{14}$ || offset || $0^{2}$
              condition ← $(\text{GPR[rs]}_{31} = 0)$ and $(\text{GPR[rs]} \neq 0^{32})$
      T + 1:  if condition then
                PC ← PC + target
              else
                NullifyCurrentInstruction
              endif


64    T:      target ← $(\text{offset}_{15})^{46}$ || offset || $0^{2}$
              condition ← $(\text{GPR[rs]}_{63} = 0)$ and $(\text{GPR[rs]} \neq 0^{64})$
      T + 1:  if condition then
                PC ← PC + target
              else
                NullifyCurrentInstruction
              endif
```

Exceptions:

None

BLEZ

Branch on Less Than or Equal to Zero

31 26	25 21	20 16	15 0
BLEZ 000110	rs	0 00000	offset

Format:

BLEZ rs, offset

MIPS I

Purpose:

Tests a general-purpose register and executes a PC relative condition branch.

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. The contents of general-purpose register *rs* are compared to zero. If the contents of general-purpose register *rs* are zero or smaller than zero, then the program branches to the target address, with a delay of one instruction.

Remark The condition branch range of this instruction is ±128 KB because an 18-bit signed offset is used. To branch to an address outside this range, use the J or JR instruction.

Operation:

```
32    T:      target ← $(\text{offset}_{15})^{14}$ || 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 ← $(\text{offset}_{15})^{46}$ || offset || $0^{2}$
              condition ← (GPR[rs]$_{63}$ = 1) or (GPR[rs] = $0^{64}$)
      T + 1:  if condition then
                PC ← PC + target
              endif
```

Exceptions:

None

BLEZL

Branch on Less Than or Equal to Zero Likely

31 26	25 21	20 16	15 0
BLEZL 010110	rs	0 00000	offset

Format:

BLEZL rs, offset

MIPS II

Purpose:

Tests a general-purpose register and executes a PC relative condition branch. Executes a delay slot only when a given branch condition is satisfied.

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. The contents of general-purpose register *rs* is compared to zero. If the contents of general-purpose register *rs* are zero or smaller than zero, then the program branches to the target address, with a delay of one instruction.
If the conditional branch is not taken, the instruction in the branch delay slot is discarded.

Remarks 1. The condition branch range of this instruction is ±128 KB because an 18-bit signed offset is used. To branch to an address outside this range, use the J or JR instruction.

2. Use this instruction only when it is expected with a high probability (98% or higher) that a given branch condition is satisfied. If the branch condition is not satisfied or if the branch destination is not known, use the BLEZ instruction.

Operation:

```
32      T:      target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
                condition ← $(\text{GPR[rs]}_{31} = 1)$ or $(\text{GPR[rs]} = 0^{32})$
        T + 1:  if condition then
                  PC ← PC + target
                else
                  NullifyCurrentInstruction
                endif

64      T:      target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
                condition ← $(\text{GPR[rs]}_{63} = 1)$ or $(\text{GPR[rs]} = 0^{64})$
        T + 1:  if condition then
                  PC ← PC + target
                else
                  NullifyCurrentInstruction
                endif
```

Exceptions:

None

BLTZ

Branch on Less Than Zero

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BLTZ 00000	offset

Format:

BLTZ rs, offset

MIPS I

Purpose:

Tests a general-purpose register and executes a PC relative condition branch.

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of general-purpose register *rs* are smaller than zero, then the program branches to the target address, with a delay of one instruction.

Remark The condition branch range of this instruction is ±128 KB because an 18-bit signed offset is used. To branch to an address outside this range, use the J or JR instruction.

Operation:

```
32    T:      target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
              condition ← $(\text{GPR}[rs]_{31} = 1)$
      T + 1:  if condition then
                PC ← PC + target
              endif


64    T:      target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
              condition ← $(\text{GPR}[rs]_{63} = 1)$
      T + 1:  if condition then
                PC ← PC + target
              endif
```

Exceptions:

None

BLTZAL

Branch on Less Than Zero and Link

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BLTZAL 10000	offset

Format:

BLTZAL rs, offset

MIPS I

Purpose:

Tests a general-purpose register and executes a PC relative condition procedure call.

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset,* shifted left two bits and sign-extended. Unconditionally, the address of the instruction after the delay slot is stored in the link register, *r31*. If the contents of general-purpose register *rs* are smaller than zero when compared to zero, then the program branches to the target address, with a delay of one instruction.
General-purpose register *r31* should not be specified as general-purpose register *rs*. If register *r31* is specified, restarting may be impossible due to the destruction of *rs* contents caused by storing a link address. Even such instructions are executed, an exception does not result.

Remark The condition branch range of this instruction is ±128 KB because an 18-bit signed offset is used. To branch to an address outside this range, use the J or JR instruction.

Operation:

```
32    T:      $target \leftarrow (offset_{15})^{14} \| offset \| 0^{2}$
              $condition \leftarrow (GPR[rs]_{31} = 1)$
              $GPR[31] \leftarrow PC + 8$
      T + 1:  if condition then
                $PC \leftarrow PC + target$
              endif


64    T:      $target \leftarrow (offset_{15})^{46} \| offset \| 0^{2}$
              $condition \leftarrow (GPR[rs]_{63} = 1)$
              $GPR[31] \leftarrow PC + 8$
      T + 1:  if condition then
                $PC \leftarrow PC + target$
              endif
```

Exceptions:

None

BLTZALL

Branch on Less Than Zero and Link Likely

(1/2)

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BLTZALL 10010	offset

Format:

BLTZALL rs, offset **MIPS II**

Purpose:

Tests a general-purpose register and executes a PC relative condition procedure call. Executes a delay slot only when a given branch condition is satisfied.

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset,* shifted left two bits and sign-extended. Unconditionally, the address of the instruction after the delay slot is stored in the link register, *r31*. If the contents of general-purpose register *rs* are smaller than zero when compared to zero, then the program branches to the target address, with a delay of one instruction.

If the conditional branch is not taken, the instruction in the branch delay slot is discarded.

General-purpose register *r31* should not be specified as general-purpose register *rs*. If register *r31* is specified, restarting may be impossible due to the destruction of *rs* contents caused by storing a link address. Even such instructions are executed, an exception does not result.

Remarks 1. The condition branch range of this instruction is ±128 KB because an 18-bit signed offset is used. To branch to an address outside this range, use the J or JR instruction.

2. Use this instruction only when it is expected with a high probability (98% or higher) that a given branch condition is satisfied. If the branch condition is not satisfied or if the branch destination is not known, use the BLTZAL instruction.

BLTZALL

Branch on Less Than Zero and Link Likely

(2/2)

Operation:

```
32      T:      target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
                condition ← $(\text{GPR[rs]}_{31} = 1)$
                GPR[31] ← PC + 8
        T + 1:  if condition then
                  PC ← PC + target
                else
                  NullifyCurrentInstruction
                endif

64      T:      target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
                condition ← $(\text{GPR[rs]}_{63} = 1)$
                GPR[31] ← PC + 8
        T + 1:  if condition then
                  PC ← PC + target
                else
                  NullifyCurrentInstruction
                endif
```

Exceptions:

None

BLTZL

Branch on Less Than Zero Likely

31 26	25 21	20 16	15 0
REGIMM 000001	rs	BLTZL 00010	offset

Format:

BLTZ rs, offset

MIPS II

Purpose:

Tests a general-purpose register and executes a PC relative condition procedure call. Executes a delay slot only when a given branch condition is satisfied.

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset*, shifted left two bits and sign-extended. If the contents of general-purpose register *rs* are smaller than zero when compared to zero, then the program branches to the target address, with a delay of one instruction.
If the conditional branch is not taken, the instruction in the branch delay slot is discarded.

Remarks 1. The condition branch range of this instruction is ±128 KB because an 18-bit signed offset is used. To branch to an address outside this range, use the J or JR instruction.

2. Use this instruction only when it is expected with a high probability (98% or higher) that a given branch condition is satisfied. If the branch condition is not satisfied or if the branch destination is not known, use the BLTZ instruction.

Operation:

```
32      T:      $target \leftarrow (offset_{15})^{14} \| offset \| 0^{2}$
                $condition \leftarrow (GPR[rs]_{31} = 1)$
        T + 1:  if condition then
                  $PC \leftarrow PC + target$
                else
                  NullifyCurrentInstruction
                endif


64      T:      $target \leftarrow (offset_{15})^{46} \| offset \| 0^{2}$
                $condition \leftarrow (GPR[rs]_{63} = 1)$
        T + 1:  if condition then
                  $PC \leftarrow PC + target$
                else
                  NullifyCurrentInstruction
                endif
```

Exceptions:

None

BNE

Branch on Not Equal

31 26	25 21	20 16	15 0
BNE 000101	rs	rt	offset

Format:

BNE rs, rt, offset

MIPS I

Purpose:

Tests a general-purpose register and executes a PC relative condition branch.

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset,* shifted left two bits and sign-extended. The contents of general-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 target address, with a delay of one instruction.

Remark The condition branch range of this instruction is ±128 KB because an 18-bit signed offset is used. To branch to an address outside this range, use the J or JR instruction.

Operation:

```
32    T:      target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
              condition ← (GPR[rs] ≠ GPR[rt])
      T + 1:  if condition then
                PC ← PC + target
              endif

64    T:      target ← $(\text{offset}_{15})^{46}$ || 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

31 26	25 21	20 16	15 0
BNEL 010101	rs	rt	offset

Format:

BNEL rs, rt, offset **MIPS II**

Purpose:

Tests a general-purpose register and executes a PC relative condition branch. Executes a delay slot only when a given branch condition is satisfied.

Description:

A branch target address is computed from the sum of the address of the instruction in the delay slot and the 16-bit *offset,* shifted left two bits and sign-extended. The contents of general-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 target address, with a delay of one instruction.

If the conditional branch is not taken, the instruction in the branch delay slot is discarded.

Remarks 1. The condition branch range of this instruction is ±128 KB because an 18-bit signed offset is used. To branch to an address outside this range, use the J or JR instruction.

2. Use this instruction only when it is expected with a high probability (98% or higher) that a given branch condition is satisfied. If the branch condition is not satisfied or if the branch destination is not known, use the BNE instruction.

Operation:

```
32     T:      target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
               condition ← (GPR[rs] $\neq$ GPR[rt])
       T + 1:  if condition then
                 PC ← PC + target
               else
                 NullifyCurrentInstruction
               endif


64     T:      target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
               condition ← (GPR[rs] $\neq$ GPR[rt])
       T + 1:  if condition then
                 PC ← PC + target
               else
                 NullifyCurrentInstruction
               endif
```

Exceptions:

None

BREAK

Breakpoint

31 26	25 6	5 0
SPECIAL 000000	code	BREAK 001101

Format:

BREAK **MIPS I**

Purpose:

Generates a breakpoint exception.

Description:

A breakpoint exception occurs, immediately and unconditionally transferring control to the exception handler.
The code field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
32, 64    T:       BreakpointException
```

Exceptions:

Breakpoint exception

CACHE

Cache Operation

(1/4)

31 26	25 21	20 16	15 0
CACHE 101111	base	op	offset

Format:

CACHE op, offset (base) **MIPS III**

Description:

The 16-bit *offset* is sign-extended and added to the contents of general-purpose register *base* to generate a virtual address. The virtual address is translated to a physical address using the TLB, and the 5-bit sub-opcode specifies a cache operation for that address.

If CP0 is not usable (user or supervisor mode) and the CP0 enable bit in the Status register is clear, a coprocessor unusable exception is taken. The operation of this instruction on any operation/cache combination not listed below, or on a secondary cache that is not incorporated in VR5500, is undefined. The operation of this instruction on uncached addresses is also undefined.

The Index operation uses part of the virtual address to specify a cache block. For a cache of $2^{CACHEBITS}$ bytes with $2^{LINEBITS}$ bytes per tag, $vAddr_{CACHEBITS...LINEBITS}$ specifies the block. The way of the cache is specified by using bit 0 of the virtual address.

In Hit, Fill, and Fetch_and_Lock operations, the way of the cache is specified by using the LRU bit of the cache tag.

Index_Load_Tag also uses $vAddr_{LINEBITS...3}$ to select the doubleword for reading parity. If the CE bit of the Status register is set, $vAddr_{LINEBITS..3}$ is used for Hit_Write_Back_Invalidate, Index_Write_Back_Invalidate, and Fill operations to select the doubleword that includes the modified parity. This operation is unconditionally executed.

The Hit operation accesses the specified cache as normal data references, and performs the specified operation if the cache block contains valid data with the specified physical address (a hit). If the cache block is invalid or contains a different address (a miss), no operation is performed.

CACHE

Cache Operation

(2/4)

Write back from a cache goes to main memory. The main memory address to be written is specified by the cache tag and not the physical address translated using TLB.
TLB refill and TLB invalid exceptions can occur on any operation. For Index operations[Note] for addresses in the unmapped areas, unmapped addresses may be used to avoid TLB exceptions. Index operations never cause a TLB modified exception.

Note Physical addresses here are used to index the cache, and they do not need to match the cache tag.

Bits 17 and 16 of the instruction code specify the cache for which the operation is to be performed as follows.

$op_{1..0}$	Name	Cache
0	I	Instruction cache
1	D	Data cache
2	–	Reserved
3	–	Reserved

Bits 20 to 18 of this instruction specify the contents of cache operation. Details are provided from the next page.

CACHE

Cache Operation

(3/4)

op$_{4..2}$	Cache	Name	Operation
0	I	Index_Invalidate	Set the cache state of the cache block to Invalid.
0	D	Index_Write_ Back_Invalidate	Examine the cache state of the data cache block at the index specified by the virtual address. If the state is Dirty and not Invalid, then write back the block to 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 TagLo CP0 registers. At this time, a parity error is ignored. In addition, data is loaded from the doubleword for which the data parity was specified to the Parity Error register.
2	I, D	Index_Store_ Tag	Write the tag for the cache block at the specified index from the TagLo CP0 register.
3	D	Create_Dirty	This operation is used to avoid loading data needlessly from memory when writing new contents to an entire cache block. If the cache block does not contain the specified address, and the block is dirty, write it back to the memory. In all cases, set the cache state to Dirty. The specified physical address is set to the cache block tag in all cases and the cache status is set to Dirty.
4	I, D	Hit_Invalidate	If the cache block contains the specified address, mark the cache block Invalid.
5	I	Fill	Fill the instruction cache block from memory. If the CE bit of the Status register is set, the contents of the ECC register is used instead of the computed parity bits for addressed doubleword when written to the instruction cache.
5	D	Hit_Write_Back Invalidate	If the cache block contains the specified address, write back the data if it is Dirty, and mark the cache block Invalid.
6	D	Hit_Write_Back	If the cache block includes the specified address and if the cache status is Dirty, data is written back to the main memory and the cache status of that cache block is set to Clean.
7	I	Fetch_and_Lock	If the specified address is not included in the cache block, that block is filled with data from the main memory. In all cases, the specified physical address is set to the cache block tag and the cache status is locked.
7	D	Fetch_and_Lock	If the specified address is not included in the cache block and if that block is Dirty, the data is written back and the block is filled with data from the main memory. In all cases, the specified physical address is set to the cache block tag and the cache status is locked.

CACHE

Cache Operation

(4/4)

Operation:

```
32, 64   T:   $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0})+GPR[base]$
              $(pAddr,uncached) \leftarrow AddressTranslation (vAddr, DATA)$
              CacheOp (op, vAddr, pAddr)
```

Exceptions:

Coprocessor unusable exception
TLB refill exception
TLB invalid exception
Bus error exception
Address error exception
Cache error exception

CLO

Count Leading Ones in Word

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL2 011100	rs	rt	rd	0 00000	CLO 100001

Format:

CLO rd, rs

VR5500

Purpose:

Counts the number of 1s in 32-bit data.

Description:

This instruction scans the 32-bit contents of general-purpose register *rs* from the most significant bit toward the least significant bit, and stores the number of 1s in general-purpose register *rd*. If the value of register *rs* is all 1, 32 is stored in *rd*.

In the 64-bit mode, the operand must be a sign-extended 32-bit value; otherwise the result will be undefined.

Specify the same register as general-purpose register *rd* for general-purpose register *rt*.

Operation:

```
32, 64   T:   temp ← 32
              for i in 31..0
                if $GPR[rs]_i$ = 0 then
                     temp ← 31 – i
                     break
                endif
              endfor
              GPR[rd] ← $(temp_{31})^{32}$ || temp
```

Exceptions:

None

CLZ

Count Leading Zeros in Word

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL2 011100	rs	rt	rd	0 00000	CLZ 100000

Format:

CLZ rd, rs

VR5500

Purpose:

Counts the number of 0s in 32-bit data.

Description:

This instruction scans the 32-bit contents of general-purpose register *rs* from the most significant bit toward the least significant bit, and stores the number of 0s in general-purpose register *rd*. If the value of register *rs* is all 0, 32 is stored in *rd*.

In the 64-bit mode, the operand must be a sign-extended 32-bit value; otherwise the result will be undefined.

Specify the same register as general-purpose register *rd* for general-purpose register *rt*.

Operation:

```
32, 64   T:   temp ← 32
              for i in 31..0
                if $GPR[rs]_i$ = 1 then
                    temp ← 31 – i
                    break
                endif
              endfor
              GPR[rd] ← $(temp_{31})^{32}$ || temp
```

Exceptions:

None

COPz

Coprocessor z Operation

31 26	25	24 0
COPz 0100XX[Note]	CO 1	cofun

Format:

COPz cofun

MIPS I

Purpose:

Executes a coprocessor instruction.

Description:

This instruction executes a coprocessor instruction. This instruction can specify and reference an internal coprocessor register and can modify the status of the coprocessor. However, the status of the processor, cache, and main memory remains unchanged. For details of the coprocessor instructions, refer to **CHAPTER 18 FPU INSTRUCTION SET**.

Operation:

```
32, 64   T:    CoprocessorOperation (z, cofun)
```

Exceptions:

Coprocessor unusable exception

Floating-point operation exception (CP1 only)

Note See the opcode table below, or **17.4 CPU Instruction Opcode Bit Encoding**.

Opcode Table:

	31	30	29	28	27	26	25	0
COP0	0	1	0	0	0	0	1	

	31	30	29	28	27	26	25	0
COP1	0	1	0	0	0	1	1	

	31	30	29	28	27	26	25	0
COP2	0	1	0	0	1	0	1	

Opcode

Coprocessor sub-opcode

Coprocessor No.

Remark Coprocessor 2 is reserved in the VR5500.

DADD

Doubleword Add

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DADD 101100

Format:

DADD rd, rs, rt

MIPS III

Purpose:

Adds 64-bit integers. A trap is performed if an overflow occurs.

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 destination register *rd* is not modified when an integer overflow exception occurs.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
64     T:     GPR[rd] ← GPR[rs] + GPR[rt]
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Integer overflow exception

Reserved instruction exception (32-bit user/supervisor mode)

DADDI

Doubleword Add Immediate

31 26	25 21	20 16	15 0
DADDI 011000	rs	rt	immediate

Format:

DADDI rt, rs, immediate **MIPS III**

Purpose:

Adds a 64-bit integer to a constant. A trap is performed if an overflow occurs.

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 destination register *rt* is not modified when an integer overflow exception occurs. This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
64    T:    $GPR[rt] \leftarrow GPR[rs] + (immediate_{15})^{48} \| immediate_{15..0}$
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Integer overflow exception
Reserved instruction exception (32-bit user/supervisor mode)

DADDIU

Doubleword Add Immediate Unsigned

31 26	25 21	20 16	15 0
DADDIU 011001	rs	rt	immediate

Format:

DADDIU rt, rs, immediate

MIPS III

Purpose:

Adds a 64-bit integer to a constant.

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 defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

The only difference between this instruction and the DADDI instruction is that DADDIU never causes an integer overflow exception.

Operation:

```
64    T:    $GPR[rt] \leftarrow GPR[rs] + (immediate_{15})^{48} \mathbin{||} immediate_{15..0}$
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Reserved instruction exception (32-bit user/supervisor mode)

DADDU

Doubleword Add Unsigned

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DADDU 101101

Format:

DADDU rd, rs, rt

MIPS III

Purpose:

Adds 64-bit integers.

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 defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

The only difference between this instruction and the DADD instruction is that DADDU never causes an integer overflow exception.

Operation:

```
64      T:      GPR[rd] ← GPR[rs] + GPR[rt]
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Reserved instruction exception (32-bit user/supervisor mode)

DCLO

Count Leading Ones in Doubleword

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL2 011100	rs	rt	rd	0 00000	DCLO 100101

Format:

DCLO rd, rs **VR5500**

Purpose:

Counts the number of 1s in 64-bit data.

Description:

This instruction scans the 64-bit contents of general-purpose register *rs* from the most significant bit toward the least significant bit, and stores the number of 1s in general-purpose register *rd*. If the value of register *rs* is all 1, 64 is stored in *rd*.

Specify the same register as general-purpose register *rd* for general-purpose register *rt*.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
64      T:      temp ← 64
                for i in 63..0
                  if $GPR[rs]_i$ = 0 then
                        temp ← 63 – i
                        break
                  endif
                endfor
                GPR[rd] ← $(temp_{31})^{32}$ || temp
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Reserved instruction exception (32-bit user/supervisor mode)

DCLZ

Count Leading Zeros in Doubleword

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL2 011100	rs	rt	rd	0 00000	DCLZ 100100

Format:

DCLZ rd, rs

VR5500

Purpose:

Counts the number of 0s in 64-bit data.

Description:

This instruction scans the 64-bit contents of general-purpose register *rs* from the most significant bit toward the least significant bit, and stores the number of 0s in general-purpose register *rd*. If the value of register *rs* is all 0, 64 is stored in *rd*.

Specify the same register as general-purpose register *rd* for general-purpose register *rt*.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
64      T:      temp ← 64
                for i in 63..0
                  if $GPR[rs]_i$ = 1 then
                      temp ← 63 – i
                      break
                  endif
                endfor
                GPR[rd] ← $(temp_{31})^{32}$ || temp
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Reserved instruction exception (32-bit user/supervisor mode)

DDIV

Doubleword Divide

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 0000000000	DDIV 011110

Format:

DDIV rs, rt

MIPS III

Purpose:

Divides a 64-bit signed integer.

Description:

The contents of general-purpose register *rs* are divided by the contents of general-purpose register *rt,* treating both operands as signed values. No integer overflow exception occurs under any circumstances, and the result of this operation is undefined when the divisor is zero.

This instruction is typically followed by additional instructions to check for a zero divisor and for overflow.

When the operation completes, the quotient word of the double result is loaded to special register *LO*, and the remainder word of the double result is loaded to 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 between the MFHI or MFLO instruction and the DDIV instruction.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

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 The operation is the same in the 32-bit kernel mode.

Exceptions:

Reserved instruction exception (32-bit user/supervisor mode)

DDIVU

Doubleword Divide Unsigned

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 0000000000	DDIVU 011111

Format:

DDIVU rs, rt

MIPS III

Purpose:

Divides a 64-bit unsigned integer.

Description:

The contents of general-purpose register *rs* are divided by the contents of general-purpose register *rt,* treating both operands as unsigned values. No integer overflow exception occurs under any circumstances, and the result of this operation is undefined when the divisor is zero.

This instruction may be followed by additional instructions to check for a zero divisor, inserted by the programmer.

When the operation completes, the quotient word of the double result is loaded to special register *LO*, and the remainder word of the double result is loaded to 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 between the MFHI or MFLO instruction and the DDIVU instruction.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

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 The operation is the same in the 32-bit kernel mode.

Exceptions:

Reserved instruction exception (32-bit user/supervisor mode)

DIV

Divide

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 0000000000	DIV 011010

Format:

DIV rs, rt

MIPS I

Purpose:

Divides a 32-bit signed integer.

Description:

The contents of general-purpose register *rs* are divided by the contents of general-purpose register *rt,* treating both operands as signed values. No integer overflow exception occurs under any circumstances, and the result of this operation is undefined when the divisor is zero.

In 64-bit mode, the operands must be valid sign-extended, 32-bit values.

This instruction is typically followed by additional instructions to check for a zero divisor and for overflow.

When the operation completes, the quotient word of the double result is loaded to special register *LO*, and the remainder word of the double result is loaded to 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 between the MFHI or MFLO instruction and the DDIV instruction.

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:      $q \leftarrow GPR[rs]_{31..0}$ div $GPR[rt]_{31..0}$
              $r \leftarrow GPR[rs]_{31..0}$ mod $GPR[rt]_{31..0}$
              $LO \leftarrow (q_{31})^{32} \mid\mid q_{31..0}$
              $HI \leftarrow (r_{31})^{32} \mid\mid r_{31..0}$
```

Exceptions:

None

DIVU

Divide Unsigned

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 0000000000	DIVU 011011

Format:

DIVU rs, rt

MIPS I

Purpose:

Divides a 32-bit unsigned integer.

Description:

The contents of general-purpose register *rs* are divided by the contents of general-purpose register *rt,* treating both operands as unsigned values. No integer overflow exception occurs under any circumstances, and the result of this operation is undefined when the divisor is zero. In 64-bit mode, the operands must be valid sign-extended, 32-bit values.

This instruction is typically followed by additional instructions to check for a zero divisor.

When the operation completes, the quotient word of the double result is loaded to special register *LO*, and the remainder word of the double result is loaded to 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 between the MFHI or MFLO instruction and the DDIV instruction.

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

31 26	25 21	20 16	15 11	10 0
COP0 010000	DMF 00001	rt	rd	0 00000000000

Format:

DMFC0 rt, rd

MIPS III

Description:

The contents of coprocessor register *rd* of the CP0 are loaded to general-purpose register *rt.*

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode. The contents of the coprocessor register *rd* source are written to the 64-bit general-purpose register *rt* destination. The operation of DMFC0 on a 32-bit coprocessor 0 register is undefined.

Operation:

```
64     T:       data ← CPR[0, rd]
       T + 1:   GPR[rt] ← data
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Coprocessor unusable exception (64-/32-bit user/supervisor mode if CP0 is disabled)

Reserved instruction exception (32-bit user/supervisor mode)

DMTC0

Doubleword Move to System Control Coprocessor

31 26	25 21	20 16	15 11	10 0
COP0 010000	DMT 00101	rt	rd	0 00000000000

Format:

DMTC0 rt, rd

MIPS III

Description:

The contents of general-purpose register *rt* are loaded to coprocessor register *rd* of the CP0.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

The contents of the general-purpose register *rd* source are written to the 64-bit coprocessor register *rt* destination.

The operation of DMTC0 on a 32-bit coprocessor 0 register is undefined.

Because the state of the virtual address translation system may be altered by this instruction, the operation of load instructions, store instructions, and TLB operations immediately prior to and after this instruction are undefined.

Operation:

```
64      T:        data ← GPR[rt]
        T + 1:    CPR[0, rd] ← data
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Coprocessor unusable exception (64-/32-bit user/supervisor mode if CP0 is disabled)

Reserved instruction exception (32-bit user/supervisor mode)

DMULT

Doubleword Multiply

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 0000000000	DMULT 011100

Format:

DMULT rs, rt

MIPS III

Purpose:

Multiply 64-bit signed integers.

Description:

The contents of general-purpose registers *rs* and *rt* are multiplied, treating both operands as signed values. No integer overflow exception occurs under any circumstances.

When the operation completes, the lower word of the double result is loaded to special register *LO*, and the higher word of the double result is loaded to 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 instructions between the MFHI or MFLO instruction and the DMULT instruction.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
64      T – 2:  LO ← undefined
                HI ← undefined
        T – 1:  LO ← undefined
                HI ← undefined
        T:      t ← GPR[rs] * GPR[rt]
                LO ← $t_{63..0}$
                HI ← $t_{127..64}$
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Reserved instruction exception (32-bit user/supervisor mode)

DMULTU

Doubleword Multiply Unsigned

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 0000000000	DMULTU 011101

Format:

DMULTU rs, rt

MIPS III

Purpose:

Multiply 64-bit unsigned integers.

Description:

The contents of general-purpose registers *rs* and *rt* are multiplied, treating both operands as unsigned values. No integer overflow exception occurs under any circumstances.

When the operation completes, the lower word of the double result is loaded to special register *LO*, and the higher word of the double result is loaded to 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 instructions between the MFHI or MFLO instruction and the DMULTU instruction.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
64      T – 2:  LO ← undefined
                HI ← undefined
        T – 1:  LO ← undefined
                HI ← undefined
        T:      t ← (0 || GPR[rs]) * (0 || GPR[rt])
                LO ← $t_{63..0}$
                HI ← $t_{127..64}$
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Reserved instruction exception (32-bit user/supervisor mode)

DROR

Doubleword Rotate Right

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	1 00001	rt	rd	sa	DROR 111010

Format:

DROR rd, rt, sa

VR5500

Purpose:

Arithmetically shifts a doubleword to the right by the specific number of bits (0 to 31 bits).

Description:

This instruction shifts the contents of general-purpose register *rt* to the right by the number of bits specified by *sa*. The lower bit that is shifted out is inserted in the higher bit. The result is stored in general-purpose register *rd*. This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
64    T:    $GPR[rd] \leftarrow GPR[rt]_{sa-1..0} \,||\, GPR[rt]_{63..sa}$
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Reserved instruction exception (32-bit user/supervisor mode)

DROR32

Doubleword Rotate Right + 32

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	1 00001	rt	rd	sa	DROR32 111110

Format:

DROR32 rd, rt, sa

VR5500

Purpose:

Arithmetically shifts a doubleword to the right by the specific number of bits (32 to 63 bits).

Description:

This instruction shifts the contents of general-purpose register *rt 32 + sa* bits to the right. The lower bit that is shifted out is inserted in the higher bit. The result is stored in general-purpose register *rd*.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
32, 64   T:   $s \leftarrow sa + 32$
              $GPR[rd] \leftarrow GPR[rt]_{s-1..0} \| GPR[rt]_{63..s}$
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Reserved instruction exception (32-bit user/supervisor mode)

DRORV

Doubleword Rotate Right Variable

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	1 00001	DRORV 010110

Format:

DRORV rd, rt, rs **VR5500**

Purpose:

Arithmetically shifts a doubleword to the right by the specified number of bits.

Description:

This instruction shifts the contents of general-purpose register *rt* to the right by the number of bits specified by the lower 5 bits of general-purpose register *rs*. The lower bit that is shifted out is inserted in the higher bit. The result is stored in general-purpose register *rd*.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
32, 64   T:   $s \leftarrow GPR[rs]_{4..0}$
              $GPR[rd] \leftarrow GPR[rt]_{s-1..0} \,||\, GPR[rt]_{63..s}$
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Reserved instruction exception (32-bit user/supervisor mode)

DSLL

Doubleword Shift Left Logical

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSLL 111000

Format:

DSLL rd, rt, sa **MIPS III**

Purpose:

Shifts a doubleword to the left by the specific number of bits (0 to 31 bits).

Description:

The contents of general-purpose register *rt* are shifted left by the number of bits specified by *sa*, inserting zeros into the lower bits. The result is stored in general-purpose register *rd*.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
64    T:    $s \leftarrow 0 || sa$
            $GPR[rd] \leftarrow GPR[rt]_{(63-s)..0} || 0^{s}$
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Reserved instruction exception (32-bit user/supervisor mode)

DSLL32

Doubleword Shift Left Logical + 32

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSLL32 111100

Format:

DSLL32 rd, rt, sa

MIPS III

Purpose:

Shifts a doubleword to the left by the specific number of bits (32 to 63 bits).

Description:

The contents of general-purpose register *rt* are shifted left by *32* + *sa* bits, inserting zeros into the lower bits. The result is stored in general-purpose register *rd*.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
64      T:      $s \leftarrow 1 \| sa$
                $GPR[rd] \leftarrow GPR[rt]_{(63-s)..0} \| 0^{s}$
```

Exceptions:

Reserved instruction exception (32-bit user/supervisor mode)

DSLLV

Doubleword Shift Left Logical Variable

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DSLLV 010100

Format:

DSLLV rd, rt, rs **MIPS III**

Purpose:

Shifts a doubleword to the left by the specified number of bits.

Description:

The contents of general-purpose register *rt* are shifted left by the number of bits specified by the lower 6 bits contained in general-purpose register *rs*, inserting zeros into the lower bits. The result is stored in general-purpose register *rd*.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
64    T:    $s \leftarrow GPR[rs]_{5..0}$
            $GPR[rd] \leftarrow GPR[rt]_{(63-s)..0} \,||\, 0^{s}$
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Reserved instruction exception (32-bit user/supervisor mode)

DSRA

Doubleword Shift Right Arithmetic

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSRA 111011

Format:

DSRA rd, rt, sa

MIPS III

Purpose:

Arithmetically shifts a doubleword to the right by the specific number of bits (0 to 31 bits).

Description:

The contents of general-purpose register *rt* are shifted right by *sa* bits, sign-extending the higher bits. The result is stored in general-purpose register *rd*.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
64    T:    $s \leftarrow 0 \,||\, sa$
            $GPR[rd] \leftarrow (GPR[rt]_{63})^{s} \,||\, GPR[rt]_{63..s}$
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Reserved instruction exception (32-bit user/supervisor mode)

DSRA32

Doubleword Shift Right Arithmetic + 32

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSRA32 111111

Format:

DSRA32 rd, rt, sa

MIPS III

Purpose:

Arithmetically shifts a doubleword to the right by the specific number of bits (32 to 63 bits).

Description:

The contents of general-purpose register *rt* are shifted right by *32 + sa* bits, sign-extending the higher bits. The result is stored in general-purpose register *rd*.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
64    T:    $s \leftarrow 1 \,||\, sa$
            $GPR[rd] \leftarrow (GPR[rt]_{63})^{s} \,||\, GPR[rt]_{63..s}$
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Reserved instruction exception (32-bit user/supervisor mode)

DSRAV

Doubleword Shift Right Arithmetic Variable

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DSRAV 010111

Format:

DSRAV rd, rt, rs

MIPS III

Purpose:

Arithmetically shifts a doubleword to the right by the specified number of bits.

Description:

The contents of general-purpose register *rt* are shifted right by the number of bits specified by the lower 6 bits of general-purpose register *rs*, sign-extending the higher bits. The result is stored in general-purpose register *rd*.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
64    T:    $s \leftarrow GPR[rs]_{5..0}$
            $GPR[rd] \leftarrow (GPR[rt]_{63})^{s} \,||\, GPR[rt]_{63..s}$
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Reserved instruction exception (32-bit user/supervisor mode)

DSRL

Doubleword Shift Right Logical

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSRL 111010

Format:

DSRL rd, rt, sa **MIPS III**

Purpose:

Logically shifts a doubleword to the right by the specific number of bits (0 to 31 bits).

Description:

The contents of general-purpose register *rt* are shifted right by *sa* bits, inserting zeros into the higher bits. The result is stored in general-purpose register *rd*.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
64    T:    $s \leftarrow 0 \| sa$
            $GPR[rd] \leftarrow 0^{s} \| GPR[rt]_{63..s}$
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Reserved instruction exception (32-bit user/supervisor mode)

DSRL32

Doubleword Shift Right Logical + 32

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	DSRL32 111110

Format:

DSRL32 rd, rt, sa **MIPS III**

Purpose:

Logically shifts a doubleword to the right by the specific number of bits (32 to 63 bits).

Description:

The contents of general-purpose register *rt* are shifted right by *32* + *sa* bits, inserting zeros into the higher bits. The result is stored in general-purpose register *rd*.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
64    T:    $s \leftarrow 1 \mid\mid sa$
            $GPR[rd] \leftarrow 0^{s} \mid\mid GPR[rt]_{63..s}$
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Reserved instruction exception (32-bit user/supervisor mode)

DSRLV

Doubleword Shift Right Logical Variable

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DSRLV 010110

Format:

DSRLV rd, rt, rs

MIPS III

Purpose:

Logically shifts a doubleword to the right by the specified number of bits.

Description:

The contents of general-purpose register *rt* are shifted right by the number of bits specified by the lower 6 bits of general-purpose register *rs,* inserting zeros into the higher bits. The result is stored in general-purpose register *rd.*

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
64      T:      $s \leftarrow GPR[rs]_{5..0}$
                $GPR[rd] \leftarrow 0^{s} \,||\, GPR[rt]_{63..s}$
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Reserved instruction exception (32-bit user/supervisor mode)

DSUB

Doubleword Subtract

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DSUB 101110

Format:

DSUB rd, rs, rt

MIPS III

Purpose:

Subtract a 64-bit integer. A trap is performed if an overflow occurs.

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 destination register *rd* is not modified when an integer overflow exception occurs.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
64      T:      GPR[rd] ← GPR[rs] – GPR[rt]
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Integer overflow exception

Reserved instruction exception (32-bit user/supervisor mode)

DSUBU

Doubleword Subtract Unsigned

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	DSUBU 101111

Format:

DSUBU rd, rs, rt

MIPS III

Purpose:

Subtract a 64-bit integer.

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 never causes an integer overflow.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
64      T:      GPR[rd] ← GPR[rs] – GPR[rt]
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Reserved instruction exception (32-bit user/supervisor mode)

ERET

Return from Exception

31 26	25	24 6	5 0
COP0 010000	CO 1	0 0000000000000000000	ERET 011000

Format:

ERET

MIPS III

Description:

The ERET instruction is the instruction for returning from an interrupt, exception, or error exception. Unlike a branch or jump instruction, ERET does not execute the next instruction.

The ERET instruction must not be placed in a branch delay slot.

If the ERL bit of the Status register is set (SR_2 = 1), the contents of the ErrorEPC register are loaded to the PC and the ERL bit is cleared (SR_2). Otherwise (SR_2 = 0), the contents of the PC are loaded from the EPC register, and the EXL bit of the Status register is cleared (SR_1 = 0).

Because the LL bit is cleared by the ERET instruction, an execution of ERET between the LL, LLD instructions and SC, SD instructions causes the SC instruction to fail.

Operation:

```
32, 64   T:   if $SR_2 = 1$ then
                PC $\leftarrow$ ErrorEPC
                SR $\leftarrow SR_{31..3}$ || 0 || $SR_{1..0}$
              else
                PC $\leftarrow$ EPC
                SR $\leftarrow SR_{31..2}$ || 0 || $SR_0$
              endif
              LLbit $\leftarrow$ 0
```

Exceptions:

Coprocessor unusable exception

J

Jump

31 26	25 0
J 000010	target

Format:

J target

MIPS I

Purpose:

Executes a branch in the area (256 MB) currently aligned.

Description:

The 26-bit target address is shifted left two bits and combined with the higher 4 bits of the address of the delay slot. The program unconditionally jumps to this calculated address with a delay of one instruction.

Operation:

```
32      T:       temp ← target
        T + 1:   PC ← $PC_{31..28}$ || temp || $0^2$

64      T:       temp ← target
        T + 1:   PC ← $PC_{63..28}$ || temp || $0^2$
```

Exceptions:

None

JAL

Jump and Link

31 26	25 0
JAL 000011	target

Format:

JAL target

MIPS I

Purpose:

Executes a procedure call in the area (256 MB) currently aligned.

Description:

The 26-bit target address is shifted left two bits and combined with the higher 4 bits of the address of the delay slot. The program unconditionally jumps to this calculated address with a delay of one instruction. The address of the instruction immediately after a 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

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	0 00000	rd	0 00000	JALR 001001

Format:

JALR rs
JALR rd, rs

MIPS I

Purpose:

Executes a procedure call to an instruction address in a register.

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 immediately after the delay slot is placed in general-purpose register *rd*. The default value of *rd*, if omitted in the assembly language instruction, is 31.

Register numbers *rs* and *rd* may not be equal, because such an instruction does not have the same effect when re-executed. Because storing a link address destroys the contents of *rs* if they are equal. Even such instructions are execute, an exception does not result, and the result of executing such an instruction is undefined.

The effective target address of general-purpose register *rs* must be aligned. If the lower 2 bits are not zero, an address error exception will occur when the jump target instruction is subsequently fetched.

Operation:

```
32, 64   T:      temp ← GPR[rs]
                 GPR[rd] ← PC + 8
         T + 1:  PC ← temp
```

Exceptions:

Address error exception

JR

Jump Register

31 26	25 21	20 6	5 0
SPECIAL 000000	rs	0 000000000000000	JR 001000

Format:

JR rs

MIPS I

Description:

The program unconditionally jumps to the address contained in general-purpose register *rs* with a delay of one instruction.

The effective target address of general-purpose register *rs* must be aligned. If the lower 2 bits are not zero, an address error exception will occur when the jump target instruction is subsequently fetched.

Operation:

```
32, 64  T:       temp ← GPR[rs]
        T + 1:   PC ← temp
```

Exceptions:

Address error exception

LB

Load Byte

31 26	25 21	20 16	15 0
LB 100000	base	rt	offset

Format:

LB rt, offset (base)

MIPS I

Purpose:

Loads 1 byte from memory as a signed value.

Description:

The 16-bit *offset* is sign-extended and added to the contents of general-purpose register *base* to generate a virtual address. The contents of the byte at the memory location specified by the effective address are sign-extended and loaded to general-purpose register *rt*.

Operation:

```
32    T:    $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
            $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
            $pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0}\ xor\ ReverseEndian^{3})$
            $mem \leftarrow LoadMemory\ (uncached, BYTE, pAddr, vAddr, DATA)$
            $byte \leftarrow vAddr_{2..0}\ xor\ BigEndianCPU^{3}$
            $GPR[rt] \leftarrow (mem_{7+8*byte})^{24} \| mem_{7+8*byte..8*byte}$

64    T:    $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
            $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr,DATA)$
            $pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0}\ xor\ ReverseEndian^{3})$
            $mem \leftarrow LoadMemory\ (uncached, BYTE, pAddr, vAddr, DATA)$
            $byte \leftarrow vAddr_{2..0}\ xor\ BigEndianCPU^{3}$
            $GPR[rt] \leftarrow (mem_{7+8*byte})^{56} \| mem_{7+8*byte..8*byte}$
```

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

LBU

Load Byte Unsigned

31 26	25 21	20 16	15 0
LBU 100100	base	rt	offset

Format:

LBU rt, offset (base)

MIPS I

Purpose:

Loads 1 byte from memory as an unsigned value.

Description:

The 16-bit *offset* is sign-extended and added to the contents of general-purpose register *base* to generate a virtual address. The contents of the byte at the memory location specified by the effective address are zero-extended and loaded to general-purpose register *rt*.

Operation:

```
32    T:    vAddr ←((offset$_{15}$)$^{16}$ || offset$_{15..0}$) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr,DATA)
            pAddr ← pAddr$_{PSIZE-1..3}$ || (pAddr$_{2..0}$ xor ReverseEndian$^3$)
            mem ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
            byte ← vAddr$_{2..0}$ xor BigEndianCPU$^3$
            GPR[rt] ← 0$^{24}$ || mem$_{7+8*byte..8*byte}$

64    T:    vAddr ← ((offset$_{15}$)$^{48}$ || offset$_{15..0}$) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr,DATA)
            pAddr ← pAddr$_{PSIZE-1..3}$ || (pAddr$_{2..0}$ xor ReverseEndian$^3$)
            mem ← LoadMemory (uncached, BYTE, pAddr, vAddr, DATA)
            byte ← vAddr$_{2..0}$ xor BigEndianCPU$^3$
            GPR[rt] ← 0$^{56}$ || mem$_{7+8*byte..8*byte}$
```

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

LD

Load Doubleword

31 26	25 21	20 16	15 0
LD 110111	base	rt	offset

Format:

LD rt, offset (base) **MIPS III**

Purpose:

Loads a doubleword from memory.

Description:

The 16-bit *offset* is sign-extended and added to the contents of general-purpose register *base* to generate a virtual address. The contents of the 64-bit doubleword at the memory location specified by the effective address are loaded to general-purpose register *rt*.

An address error exception occurs if the lower 3 bits of the effective address are not 0.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
64    T:    vAddr ← $((\text{offset}_{15})^{48} \mid\mid \text{offset}_{15..0}) + \text{GPR[base]}$
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
            GPR[rt] ← mem
```

Remark The higher 32 bits are ignored when a virtual address is generated in the 32-bit kernel mode.

Exceptions:

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

Reserved instruction exception (32-bit user/supervisor mode)

LDCz

Load Doubleword to Coprocessor z

(1/2)

31 26	25 21	20 16	15 0
LDCz 1101XX[Note]	base	rt	offset

Format:

LDCz rt, offset (base)

MIPS II

Purpose:

Loads a doubleword from memory to the coprocessor general-purpose register.

Description:

The 16-bit *offset* is sign-extended and added to the contents of general-purpose register *base* to generate a virtual address. The contents of the doubleword at the memory location specified by the effective address are loaded to CPz register *rt*. How to use data is defined for each processor.

An address error exception occurs if the lower 3 bits of the address are not 0.

This instruction set to CP0 is invalid.

If CP1 is specified and the FR bit of the status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a general-purpose register. If an odd number is specified, the operation is undefined. If the FR bit of the status bit is 1, both odd and even register numbers are valid.

Operation:

```
32    T:    vAddr ← $((\text{offset}_{15})^{16} \,||\, \text{offset}_{15..0})$ + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
            COPzLD (rt, mem)

64    T:    vAddr ← $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15..0})$ + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
            COPzLD (rt, mem)
```

Exceptions:

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

Coprocessor unusable exception

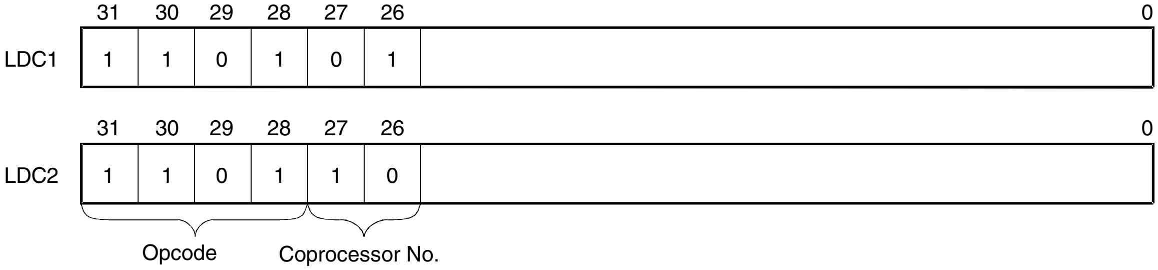
Note See the opcode table below, or **17.4 CPU Instruction Opcode Bit Encoding**.

LDCz

Load Doubleword to Coprocessor z

(2/2)

Opcode Table:



Remark Coprocessor 2 is reserved in the VR5500.

LDL

Load Doubleword Left

(1/3)

31 26	25 21	20 16	15 0
LDL 011010	base	rt	offset

Format:

LDL rt, offset (base)

MIPS III

Purpose:

Loads the most significant part of a doubleword from unaligned memory.

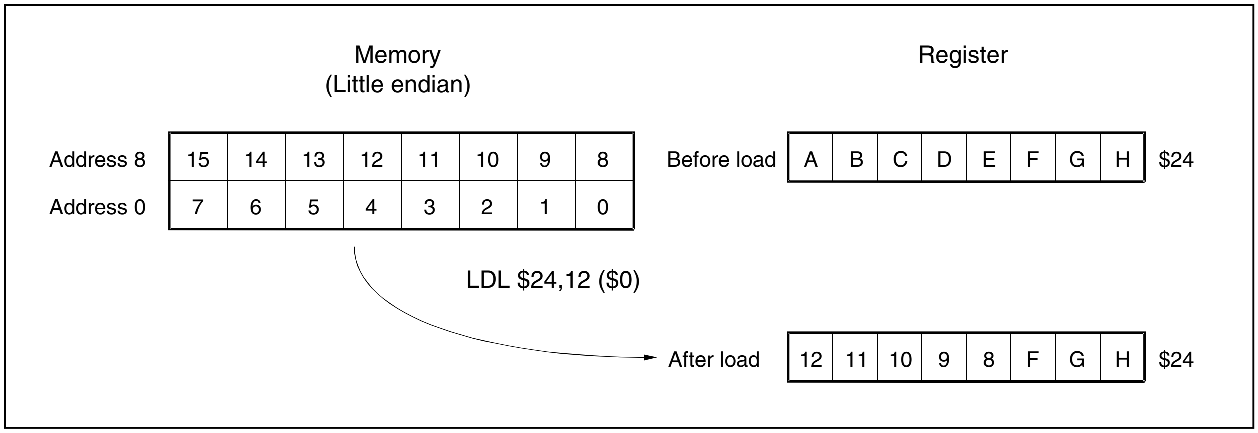
Description:

This instruction can be used in combination with the LDR instruction when loading a doubleword data in the memory that does not exist at a doubleword boundary to general-purpose register *rt*. The LDL instruction loads the higher word of the data, and the LDR instruction loads the lower word of the data to the register.

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 an arbitrary byte. Among the doubleword data in the memory whose most significant byte is the byte specified by the virtual address, only data at the same doubleword boundary as the target address is loaded and stored in the higher portion of general-purpose register *rt*. Other bits in general-purpose register *rt* will not be changed. The number of bytes to be loaded varies from one to eight depending on the byte specified.

In other words, the byte specified by the virtual address is stored in the most significant byte of general-purpose register *rt*. As long as there are lower bytes among the bytes at the same doubleword boundary, the operation to store the byte in the next byte of general-purpose register *rt* will be continued.

The lower byte of the register will not be changed.



Address 8	15	14	13	12	11	10	9	8
Address 0	7	6	5	4	3	2	1	0

Before load	A	B	C	D	E	F	G	H	$24
After load	12	11	10	9	8	F	G	H	$24

LDL

Load Doubleword Left

(2/3)

The contents of general-purpose register *rt* are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register *rt* and a following LDL (or LDR) instruction which also specifies register *rt*.

An address error exception caused by the specified address not being aligned at a doubleword boundary does not occur.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
64    T:    $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
            $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
            $pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0}\ xor\ ReverseEndian^{3})$
            if $BigEndianMem = 0$ then
              $pAddr \leftarrow pAddr_{PSIZE-1..3} \| 0^{3}$
            endif
            $byte \leftarrow vAddr_{2..0}\ xor\ BigEndianCPU^{3}$
            $mem \leftarrow LoadMemory\ (uncached, byte, pAddr, vAddr, DATA)$
            $GPR[rt] \leftarrow mem_{7+8*byte..0} \| GPR[rt]_{55-8*byte..0}$
```

Remark The higher 32 bits are ignored when a virtual address is generated in the 32-bit kernel mode.

LDL

Load Doubleword Left

(3/3)

The relationship between the address assigned to the LDL instruction and its result (each byte of the register) is shown below.

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

vAddr$_{2..0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	Destination	Type	Offset		Destination	Type	Offset	
			LEM	BEM			LEM	BEM
0	P B C D E F G H	0	0	7	I J K L M N O P	7	0	0
1	O P C D E F G H	1	0	6	J K L M N O P H	6	0	1
2	N O P D E F G H	2	0	5	K L M N O P G H	5	0	2
3	M N O P E F G H	3	0	4	L M N O P F G H	4	0	3
4	L M N O P F G H	4	0	3	M N O P E F G H	3	0	4
5	K L M N O P G H	5	0	2	N O P D E F G H	2	0	5
6	J K L M N O P H	6	0	1	O P C D E F G H	1	0	6
7	I J K L M N O P	7	0	0	P B C D E F G H	0	0	7

Remark *Type* AccessType (see **Figure 3-3 Byte Specification Related to Load and Store Instruction**) output to memory

Offset pAddr$_{2..0}$ output to memory

LEM Little-endian memory (BigEndianMem = 0)

BEM Big-endian memory (BigEndianMem = 1)

Exceptions:

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

Reserved instruction exception (32-bit user/supervisor mode)

LDR

Load Doubleword Right

(1/3)

31 26	25 21	20 16	15 0
LDR 011011	base	rt	offset

Format:

LDR rt, offset (base)

MIPS III

Purpose:

Loads the least significant part of a doubleword from unaligned memory.

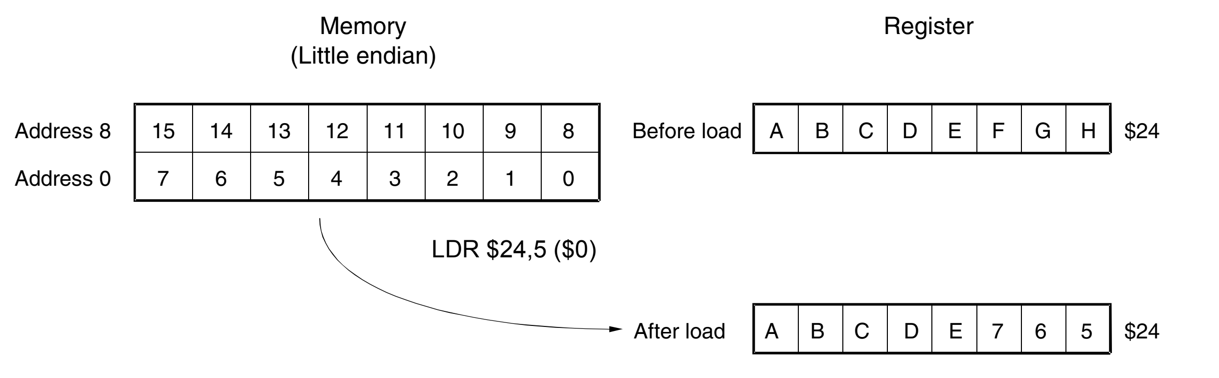
Description:

This instruction can be used in combination with the LDL instruction when loading a doubleword data in the memory that does not exist at a doubleword boundary to general-purpose register *rt*. The LDL instruction loads the higher word of the data, and the LDR instruction loads the lower word of the data to the register.

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 an arbitrary byte. Among the doubleword data in the memory whose least significant byte is the byte specified by the virtual address, only data at the same doubleword boundary as the target address is loaded and stored in the lower portion of general-purpose register *rt*. Other bits in general-purpose register *rt* will not be changed. The number of bytes to be loaded varies from one to eight depending on the byte specified.

In other words, the byte specified by the virtual address is stored in the least significant byte of general-purpose register *rt*. As long as there are higher bytes among the bytes at the same doubleword boundary, the operation to store the byte in the next byte of general-purpose register *rt* will be continued.

The higher byte of the register will not be changed.



LDR

Load Doubleword Right

(2/3)

The contents of general-purpose register *rt* are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register *rt* and a following LDR (or LDL) instruction which also specifies register *rt*.

An address error exception caused by the specified address not being aligned at a doubleword boundary does not occur.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
64      T:      vAddr ← ((offset$_{15}$)$^{48}$ || offset$_{15..0}$) + GPR[base]
                (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
                pAddr ← pAddr$_{PSIZE-1..3}$ || (pAddr$_{2..0}$ xor ReverseEndian$^{3}$)
                if BigEndianMem = 1 then
                  pAddr ← pAddr$_{PSIZE-1..3}$ || 0$^{3}$
                endif
                byte ← vAddr$_{2..0}$ xor BigEndianCPU$^{3}$
                mem ← LoadMemory (uncached, DOUBLEWORD - byte, pAddr, vAddr, DATA)
                GPR[rt] ← GPR[rt]$_{63..64-8*byte}$ || mem$_{63..8*byte}$
```

Remark The higher 32 bits are ignored when a virtual address is generated in the 32-bit kernel mode.

LDR

Load Doubleword Right

(3/3)

The relationship between the address assigned to the LDR instruction and its result (each byte of the register) is shown below.

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

vAddr$_{2..0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	Destination	Type	Offset		Destination	Type	Offset	
			LEM	BEM			LEM	BEM
0	I J K L M N O P	7	0	0	A B C D E F G I	0	7	0
1	A I J K L M N O	6	1	0	A B C D E F I J	1	6	0
2	A B I J K L M N	5	2	0	A B C D E I J K	2	5	0
3	A B C I J K L M	4	3	0	A B C D I J K L	3	4	0
4	A B C D I J K L	3	4	0	A B C I J K L M	4	3	0
5	A B C D E I J K	2	5	0	A B I J K L M N	5	2	0
6	A B C D E F I J	1	6	0	A I J K L M N O	6	1	0
7	A B C D E F G I	0	7	0	I J K L M N O P	7	0	0

Remark *Type* AccessType (see **Figure 3-3 Byte Specification Related to Load and Store Instruction**) output to memory

Offset pAddr$_{2..0}$ output to memory

LEM Little-endian memory (BigEndianMem = 0)

BEM Big-endian memory (BigEndianMem = 1)

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception
Reserved instruction exception (32-bit user/supervisor mode)

LH

Load Halfword

31 26	25 21	20 16	15 0
LH 100001	base	rt	offset

Format:

LH rt, offset (base)

MIPS I

Purpose:

Loads a halfword from memory as a signed value.

Description:

The 16-bit *offset* is sign-extended and added to the contents of general-purpose register *base* to generate a virtual address. The contents of the halfword at the memory location specified by the effective address are sign-extended and loaded to general-purpose register *rt*.

An address error exception occurs if the least-significant bit of the address is not 0.

Operation:

```
32    T:    vAddr ← ((offset$_{15}$)$^{16}$ || offset$_{15..0}$) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← pAddr$_{PSIZE–1..3}$ || (pAddr$_{2..0}$ xor (ReverseEndian$^{2}$ || 0 ))
            mem ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
            byte ← vAddr$_{2..0}$ xor (BigEndianCPU$^{2}$ || 0)
            GPR[rt] ← (mem$_{15+8*byte}$)$^{16}$ || mem$_{15+8*byte..8*byte}$

64    T:    vAddr ← ((offset$_{15}$)$^{48}$ || offset$_{15..0}$) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← pAddr$_{PSIZE–1..3}$ || (pAddr$_{2..0}$ xor (ReverseEndian$^{2}$ || 0 ))
            mem ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
            byte ← vAddr$_{2..0}$ xor (BigEndianCPU$^{2}$ || 0)
            GPR[rt] ← (mem$_{15+8*byte}$)$^{48}$ || mem$_{15+8*byte..8*byte}$
```

Exceptions:

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

LHU

Load Halfword Unsigned

31 26	25 21	20 16	15 0
LHU 100101	base	rt	offset

Format:

LHU rt, offset (base)

MIPS I

Purpose:

Loads a halfword from memory as an unsigned value.

Description:

The 16-bit *offset* is sign-extended and added to the contents of general-purpose register *base* to generate a virtual address. The contents of the halfword at the memory location specified by the effective address are zero-extended and loaded to general-purpose register *rt*.

An address error exception occurs if the least-significant bit of the address is not 0.

Operation:

```
32   T:   $vAddr \leftarrow ((offset_{15})^{16} \,||\, offset_{15..0}) + GPR[base]$
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          $pAddr \leftarrow pAddr_{PSIZE-1..3} \,||\, (pAddr_{2..0}\ xor\ (ReverseEndian_{2} \,||\, 0))$
          mem ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
          $byte \leftarrow vAddr_{2..0}\ xor\ (BigEndianCPU^{2} \,||\, 0)$
          $GPR[rt] \leftarrow 0^{16} \,||\, mem_{15+8*byte..8*byte}$

64   T:   $vAddr \leftarrow ((offset_{15})^{48} \,||\, offset_{15..0}) + GPR[base]$
          (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
          $pAddr \leftarrow pAddr_{PSIZE-1..3} \,||\, (pAddr_{2..0}\ xor\ (ReverseEndian^{2} \,||\, 0))$
          mem ← LoadMemory (uncached, HALFWORD, pAddr, vAddr, DATA)
          $byte \leftarrow vAddr_{2..0}\ xor\ (BigEndianCPU^{2} \,||\, 0)$
          $GPR[rt] \leftarrow 0^{48} \,||\, mem_{15+8*byte..8*byte}$
```

Exceptions:

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

LL

Load Linked

(1/2)

31 26	25 21	20 16	15 0
LL 110000	base	rt	offset

Format:

LL rt, offset (base) **MIPS II**

Purpose:

Loads a word from memory for atomic read-modify-write.

Description:

This instruction sign-extends a 16-bit *offset* and adds the result to the contents of general-purpose register *base* to generate a virtual address. It loads the contents of a word from the memory at a specified address to general-purpose register *rt*. In the 64-bit mode, the loaded word is sign-extended. In addition, the physical address of the specified memory is stored in the LLAddr register and the LL bit is set to 1. After that, the processor checks if the address stored in the LLAddr register has been rewritten by another processor or device.

Updating memory in a multi-processor system can be accurately performed by using the LL and SC instructions. These instructions are used as shown in the following example.

```
L1:
          LL     T1, (T0)
          ADDI   T2, T1, 1
          SC     T2, (T0)
          BEQ    T2, 0, L1
          NOP
```

In this example, the word addressed by T0 is automatically incremented. By replacing the ADDI instruction with the ORI instruction, the bit is automatically set.

This instruction can be used in all the modes and it is not necessary to enable CP0.

This instruction is defined to maintain compatibility with the other VR Series processors.

LL

Load Linked

(2/2)

The operation of the LL instruction is undefined if the specified address is in an uncached area. A cache miss that may occur between the LL and SC instructions prevents execution of the SC instruction. Therefore, do not use a load or store instruction between the LL and SC instructions. Otherwise, the operation of the SC instruction will not be guaranteed. If exceptions often occur, exceptions must be temporarily disabled because they also prevent execution of the SC instruction.

An address error exception occurs if the lower 2 bits of the address are not 0.

Operation:

```
32    T:    $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
            GPR[rt] ← mem
            LLbit ← 1
            LLAddr ← pAddr

64    T:    $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
            GPR[rt] ← mem
            LLbit ← 1
            LLAddr ← pAddr
```

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

LLD

Load Linked Doubleword

(1/2)

31 26	25 21	20 16	15 0
LLD 110100	base	rt	offset

Format:

LLD rt, offset (base)

MIPS III

Purpose:

Loads a doubleword from memory for atomic read-modify-write.

Description:

This instruction sign-extends a 16-bit *offset* and adds the result to the contents of general-purpose register *base* to generate a virtual address. It loads the contents of a doubleword from the memory at a specified address to general-purpose register *rt*. In addition, the physical address of the specified memory is stored in the LLAddr register and the LL bit is set to 1. After that, the processor checks if the address stored in the LLAddr register has been rewritten by another processor or device.

Updating memory in a multi-processor system can be accurately performed by using the LLD and SCD instructions. These instructions are used as shown in the following example.

```
L1:
          LLD    T1, (T0)
          DADDI  T2, T1, 1
          SCD    T2, (T0)
          BEQ    T2, 0, L1
          NOP
```

In this example, the doubleword addressed by T0 is automatically incremented. By replacing the DADDI instruction with the ORI instruction, the bit is automatically set.

This instruction is defined to maintain compatibility with the other VR Series processors.

LLD

Load Linked Doubleword

(2/2)

The operation of the LLD instruction is undefined if the specified address is in an uncached area. A cache miss that may occur between the LLD and SCD instructions prevents execution of the SCD instruction. 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 exceptions often occur, exceptions must be temporarily disabled because they also prevent execution of the SCD instruction.

An address error exception occurs if the lower 3 bits of the address are not 0.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
32    T:    vAddr ← $((\text{offset}_{15})^{16} \,||\, \text{offset}_{15..0})$ + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
            GPR[rt] ← mem
            LLbit ← 1
            LLAddr ← pAddr


64    T:    vAddr ← $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15..0})$ + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            mem ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
            GPR[rt] ← mem
            LLbit ← 1
            LLAddr ← pAddr
```

Remark The higher 32 bits are ignored when a virtual address is generated in the 32-bit kernel mode.

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception
Reserved instruction exception (32-bit user/supervisor mode)

LUI

Load Upper Immediate

31 26	25 21	20 16	15 0
LUI 001111	0 00000	rt	immediate

Format:

LUI rt, immediate

MIPS I

Purpose:

Loads a constant to the upper half of a word.

Description:

The 16-bit *immediate* is shifted left 16 bits and concatenated to 16 bits of zeros. The result is stored in general-purpose register *rt*. In 64-bit mode, the loaded word is sign-extended.

Operation:

```
32    T:    $GPR[rt] \leftarrow immediate \parallel 0^{16}$

64    T:    $GPR[rt] \leftarrow (immediate_{15})^{32} \parallel immediate \parallel 0^{16}$
```

Exceptions:

None

LW

Load Word

31 26	25 21	20 16	15 0
LW 100011	base	rt	offset

Format:

LW rt, offset (base)

MIPS I

Purpose:

Loads a word from memory as a signed value.

Description:

The 16-bit *offset* is sign-extended and added to the contents of general-purpose register *base* to generate a virtual address. The contents of the word at the memory location specified by the effective address are loaded to general-purpose register *rt*. In 64-bit mode, the loaded word is sign-extended.

An address error exception occurs if the lower 2 bits of the address are not 0.

Operation:

```
32    T:    vAddr ← $((\text{offset}_{15})^{16} \,||\, \text{offset}_{15..0})$ + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
            GPR[rt] ← mem


64    T:    vAddr ← $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15..0})$ + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
            GPR[rt] ← mem
```

Exceptions:

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

LWCz

Load Word to Coprocessor z

(1/2)

31 26	25 21	20 16	15 0
LWCz 1100XX[Note]	base	rt	offset

Format:

LWCz rt, offset (base)

MIPS I

Purpose:

Loads a word from memory to the coprocessor general-purpose register.

Description:

The 16-bit *offset* is sign-extended and added to the contents of general-purpose register *base* to generate a virtual address. The contents of the word at the memory location specified by the effective address are loaded to CPz register *rt*. How to use data is defined for each processor.

An address error exception occurs if the lower 2 bits of the address are not 0.

This instruction set to CP0 is invalid.

Operation:

```
32    T:    vAddr ← $((\text{offset}_{15})^{16} \,||\, \text{offset}_{15..0})$ + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← $\text{pAddr}_{\text{PSIZE}-1..3}$ || ($\text{pAddr}_{2..0}$ xor (ReverseEndian || $0^2$))
            mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
            byte ← $\text{vAddr}_{2..0}$ xor (BigEndianCPU || $0^2$)
            COPzLW (byte, rt, mem)

64    T:    vAddr ← $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15..0})$ + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← $\text{pAddr}_{\text{PSIZE}-1..3}$ || ($\text{pAddr}_{2..0}$ xor (ReverseEndian || $0^2$))
            mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
            byte ← $\text{vAddr}_{2..0}$ xor (BigEndianCPU || $0^2$)
            COPzLW (byte, rt, mem)
```

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception
Coprocessor unusable exception

Note See the opcode table below, or **17.4 CPU Instruction Opcode Bit Encoding**.

LWCz

Load Word to Coprocessor z

(2/2)

Opcode Table:

	31	30	29	28	27	26	0
LWC1	1	1	0	0	0	1	

	31	30	29	28	27	26	0
LWC2	1	1	0	0	1	0	
	Opcode				Coprocessor No.		

Remark Coprocessor 2 is reserved in the VR5500.

LWL

Load Word Left

(1/3)

31 26	25 21	20 16	15 0
LWL 100010	base	rt	offset

Format:

LWL rt, offset (base)

MIPS I

Purpose:

Loads the most significant part of a word from unaligned memory.

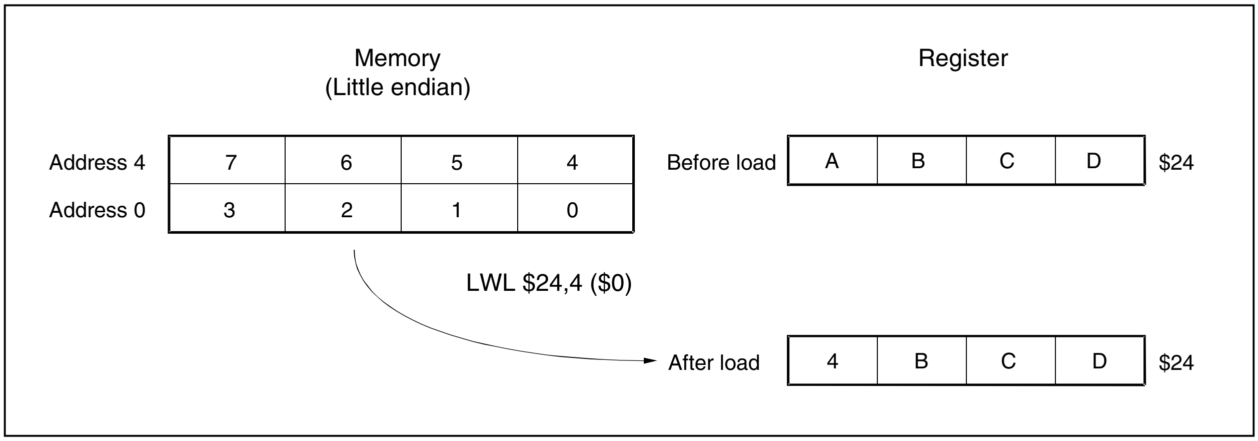
Description:

This instruction can be used in combination with the LWR instruction when loading a word data in the memory that does not exist at a word boundary to general-purpose register *rt*. The LWL instruction loads the higher word of the data, and the LWR instruction loads the lower word of the data to the register.

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 an arbitrary byte. Among the word data in the memory whose most significant byte is the byte specified by the virtual address, only data at the same word boundary as the target address is loaded and stored in the higher portion of general-purpose register *rt*. Other bits in general-purpose register *rt* will not be changed. The number of bytes to be loaded varies from one to four depending on the byte specified.

In other words, the byte specified by the virtual address is stored in the most significant byte of general-purpose register *rt*. As long as there are lower bytes among the bytes at the same word boundary, the operation to store the byte in the next byte of general-purpose register *rt* will be continued.

The lower byte of the register will not be changed.



LWL

Load Word Left

(2/3)

The contents of general-purpose register *rt* are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register *rt* and a following LWL (or LWR) instruction which also specifies register *rt*.

An address error exception caused by the specified address not being aligned at a word boundary does not occur.

Operation:

```
32    T:    vAddr ← $((\text{offset}_{15})^{16} \| \text{offset}_{15..0}) + \text{GPR[base]}$
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← $\text{pAddr}_{\text{PSIZE}-1..3} \| (\text{pAddr}_{2..0} \text{ xor ReverseEndian}^{3})$
            if BigEndianMem = 0 then
              pAddr ← $\text{pAddr}_{\text{PSIZE}-1..2} \| 0^{2}$
            endif
            byte ← $\text{vAddr}_{1..0} \text{ xor BigEndianCPU}^{2}$
            word ← $\text{vAddr}_{2} \text{ xor BigEndianCPU}$
            mem ← LoadMemory (uncached, byte, pAddr, vAddr, DATA)
            temp ← $\text{mem}_{32*\text{word}+8*\text{byte}+7..32*\text{word}} \| \text{GPR[rt]}_{23-8*\text{byte}..0}$
            GPR[rt] ← temp

64    T:    vAddr ← $((\text{offset}_{15})^{48} \| \text{offset}_{15..0}) + \text{GPR[base]}$
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← $\text{pAddr}_{\text{PSIZE}-1..3} \| (\text{pAddr}_{2..0} \text{ xor ReverseEndian}^{3})$
            if BigEndianMem = 0 then
              pAddr ← $\text{pAddr}_{\text{PSIZE}-1..2} \| 0^{2}$
            endif
            byte ← $\text{vAddr}_{1..0} \text{ xor BigEndianCPU}^{2}$
            word ← $\text{vAddr}_{2} \text{ xor BigEndianCPU}$
            mem ← LoadMemory (uncached, byte, pAddr, vAddr, DATA)
            temp ← $\text{mem}_{32*\text{word}+8*\text{byte}+7..32*\text{word}} \| \text{GPR[rt]}_{23-8*\text{byte}..0}$
            GPR[rt] ← $(\text{temp}_{31})^{32} \| \text{temp}$
```

LWL

Load Word Left

(3/3)

The relationship between the address assigned to the LWL instruction and its result (each byte of the register) is shown below.

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

vAddr$_{2..0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	Destination	Type	Offset		Destination	Type	Offset	
			LEM	BEM			LEM	BEM
0	S S S S P F G H	0	0	7	S S S S I J K L	3	4	0
1	S S S S O P G H	1	0	6	S S S S J K L H	2	4	1
2	S S S S N O P H	2	0	5	S S S S K L G H	1	4	2
3	S S S S M N O P	3	0	4	S S S S L F G H	0	4	3
4	S S S S L F G H	0	4	3	S S S S M N O P	3	0	4
5	S S S S K L G H	1	4	2	S S S S N O P H	2	0	5
6	S S S S J K L H	2	4	1	S S S S O P G H	1	0	6
7	S S S S I J K L	3	4	0	S S S S P F G H	0	0	7

Remark *Type* AccessType (see **Figure 3-3 Byte Specification Related to Load and Store Instruction**) output to memory

Offset pAddr$_{2..0}$ output to memory

LEM Little-endian memory (BigEndianMem = 0)

BEM Big-endian memory (BigEndianMem = 1)

S Bit 31 of destination sign-extended

Exceptions:

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

LWR

Load Word Right

(1/3)

31 26	25 21	20 16	15 0
LWR 100110	base	rt	offset

Format:

LWR rt, offset (base) **MIPS I**

Purpose:

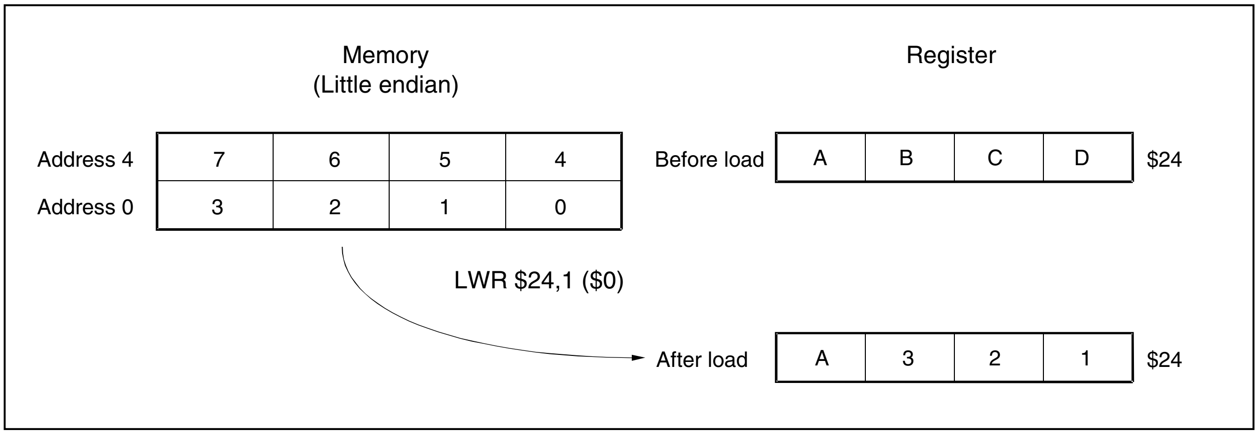
Loads the least significant part of a word from unaligned memory.

Description:

This instruction can be used in combination with the LWL instruction when loading a word data in the memory that does not exist at a word boundary to general-purpose register *rt*. The LWL instruction loads the higher word of the data, and the LWR instruction loads the lower word of the data to the register.

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 an arbitrary byte. Among the word data in the memory whose least significant byte is the byte specified by the virtual address, only data at the same word boundary as the target address is loaded and stored in the lower portion of general-purpose register *rt*. Other bits in general-purpose register *rt* will not be changed. The number of bytes to be loaded varies from one to four depending on the byte specified.

In other words, the byte specified by the virtual address is stored in the least significant byte of general-purpose register *rt*. As long as there are higher bytes among the bytes at the same word boundary, the operation to store the byte in the next byte of general-purpose register *rt* will be continued.



LWR

Load Word Right

(2/3)

The contents of general-purpose register *rt* are internally bypassed within the processor so that no NOP is needed between an immediately preceding load instruction which specifies register *rt* and a following LWR (or LWL) instruction which also specifies register *rt*.

An address error exception caused by the specified address not being aligned at a word boundary does not occur.

Operation:

```
32    T:    vAddr ← ((offset$_{15}$)$^{16}$ || offset$_{15..0}$) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← pAddr$_{PSIZE-1..3}$ || (pAddr$_{2..0}$ xor ReverseEndian$^{3}$)
            if BigEndianMem = 1 then
              pAddr ← pAddr$_{PSIZE-1..3}$ || 0$^{3}$
            endif
            byte ← vAddr$_{1..0}$ xor BigEndianCPU$^{2}$
            word ← vAddr$_{2}$ xor BigEndianCPU
            mem ← LoadMemory (uncached, 0 || byte, pAddr, vAddr, DATA)
            temp ← GPR[rt]$_{31..32-8*byte}$ || mem$_{31+32*word..32*word+8*byte}$
            GPR[rt] ← temp


64    T:    vAddr ← ((offset$_{15}$)$^{48}$ || offset$_{15..0}$) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← pAddr$_{PSIZE-1..3}$ || (pAddr$_{2..0}$ xor ReverseEndian$^{3}$)
            if BigEndianMem = 1 then
              pAddr ← pAddr$_{PSIZE-1..3}$ || 0$^{3}$
            endif
            byte ← vAddr$_{1..0}$ xor BigEndianCPU$^{2}$
            word ← vAddr$_{2}$ xor BigEndianCPU
            mem ← LoadMemory (uncached, 0 || byte, pAddr, vAddr, DATA)
            temp ← GPR[rt]$_{31..32-8*byte}$ || mem$_{31+32*word..32*word+8*byte}$
            GPR[rt] ← (temp$_{31}$)$^{32}$ || temp
```

LWR

Load Word Right

(3/3)

The relationship between the address assigned to the LWR instruction and its result (each byte of the register) is shown below.

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

$vAddr_{2..0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	Destination	Type	Offset		Destination	Type	Offset	
			LEM	BEM			LEM	BEM
0	S S S S M N O P	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* AccessType (see **Figure 3-3 Byte Specification Related to Load and Store Instruction**) output to memory

Offset $pAddr_{2..0}$ output to memory

LEM Little-endian memory (BigEndianMem = 0)

BEM Big-endian memory (BigEndianMem = 1)

S Bit 31 of destination sign-extended

X No change (32-bit mode)
Bit 31 of destination sign-extended (64-bit mode)

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception

LWU

Load Word Unsigned

31 26	25 21	20 16	15 0
LWU 100111	base	rt	offset

Format:

LWU rt, offset (base)

MIPS III

Purpose:

Loads a word from memory as an unsigned value.

Description:

The 16-bit *offset* is sign-extended and added to the contents of general-purpose register *base* to generate a virtual address. The contents of the word at the memory location specified by the effective address are loaded to general-purpose register *rt*. The loaded word is zero-extended.

An address error exception occurs if the lower 2 bits of the address are not 0.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
64    T:    vAddr ← ((offset$_{15}$)$^{48}$ || offset$_{15..0}$) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            mem ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
            GPR[rt] ← 0$^{32}$ || mem
```

Remark The higher 32 bits are ignored when a virtual address is generated in the 32-bit kernel mode.

Exceptions:

TLB refill exception

TLB invalid exception

Bus error exception

Address error exception

Reserved instruction exception (32-bit user/supervisor mode)

MACC

Multiply, Accumulate, and Move LO

31 26	25 21	20 16	15 11	10 0
SPECIAL 000000	rs	rt	rd	MACC 00101011000

Format:

MACC rd, rs, rt **VR5500**

Purpose:

Combines multiplication and addition of 32-bit signed integers for execution.

Description:

This instruction multiplies the contents of general-purpose register *rs* by the contents of general-purpose register *rt*. It treats both the operands as 32-bit signed integers.

The lower 32 bits of special register *HI* and the lower 32 bits of special register *LO* are combined and used as an accumulator.

The contents of this accumulator are added to the result of the multiplication as a 64-bit signed integer, and the result is stored in the accumulator. The lower 32 bits of the result are also stored in general-purpose register *rd*.

An integer overflow exception does not occur.

Operation:

```
32, 64   T:   $HI_{31..0} \| LO_{31..0} \leftarrow (HI_{31..0} \| LO_{31..0}) + (GPR[rs] * GPR[rt])$
              $GPR[rd]_{31..0} \leftarrow ((HI_{31..0} \| LO_{31..0}) + (GPR[rs] * GPR[rt]))_{31..0}$
```

Exceptions:

None

MACCHI

Multiply, Accumulate, and Move HI

31 26	25 21	20 16	15 11	10 0
SPECIAL 000000	rs	rt	rd	MACCHI 01101011000

Format:

MACCHI rd, rs, rt **VR5500**

Purpose:

Combines multiplication and addition of 32-bit signed integers for execution.

Description:

This instruction multiplies the contents of general-purpose register *rs* by the contents of general-purpose register *rt*. It treats both the operands as 32-bit signed integers.

The lower 32 bits of special register *HI* and the lower 32 bits of special register *LO* are combined and used as an accumulator.

The contents of this accumulator are added to the result of the multiplication as a 64-bit signed integer, and the result is stored in the accumulator. The higher 32 bits of the result are also stored in general-purpose register *rd*.

An integer overflow exception does not occur.

Operation:

```
32, 64   T:   $HI_{31..0} \| LO_{31..0} \leftarrow (HI_{31..0} \| LO_{31..0}) + (GPR[rs] * GPR[rt])$
              $GPR[rd]_{31..0} \leftarrow ((HI_{31..0} \| LO_{31..0}) + (GPR[rs] * GPR[rt]))_{63..32}$
```

Exceptions:

None

MACCHIU

Unsigned Multiply, Accumulate, and Move HI

31 26	25 21	20 16	15 11	10 0
SPECIAL 000000	rs	rt	rd	MACCHIU 01101011001

Format:

MACCHIU rd, rs, rt **VR5500**

Purpose:

Combines multiplication and addition of 32-bit unsigned integers for execution.

Description:

This instruction multiplies the contents of general-purpose register *rs* by the contents of general-purpose register *rt*. It treats both the operands as 32-bit unsigned integers.

The lower 32 bits of special register *HI* and the lower 32 bits of special register *LO* are combined and used as an accumulator.

The contents of this accumulator are added to the result of the multiplication as a 64-bit unsigned integer, and the result is stored in the accumulator. The higher 32 bits of the result are also stored in general-purpose register *rd*.

An integer overflow exception does not occur.

Operation:

```
32, 64  T:  $HI_{31..0} \| LO_{31..0} \leftarrow (HI_{31..0} \| LO_{31..0}) + (GPR[rs] * GPR[rt])$
            $GPR[rd]_{31..0} \leftarrow ((HI_{31..0} \| LO_{31..0}) + (GPR[rs] * GPR[rt]))_{63..32}$
```

Exceptions:

None

MACCU

Unsigned Multiply, Accumulate, and Move LO

31 26	25 21	20 16	15 11	10 0
SPECIAL 000000	rs	rt	rd	MACCU 00101011001

Format:

MACCU rd, rs, rt

VR5500

Purpose:

Combines multiplication and addition of 32-bit unsigned integers for execution.

Description:

This instruction multiplies the contents of general-purpose register *rs* by the contents of general-purpose register *rt*. It treats both the operands as 32-bit unsigned integers.

The lower 32 bits of special register *HI* and the lower 32 bits of special register *LO* are combined and used as an accumulator.

The contents of this accumulator are added to the result of the multiplication as a 64-bit unsigned integer, and the result is stored in the accumulator. The lower 32 bits of the result are also stored in general-purpose register *rd*.

An integer overflow exception does not occur.

Operation:

```
32, 64   T:   $HI_{31..0} \| LO_{31..0} \leftarrow (HI_{31..0} \| LO_{31..0}) + (GPR[rs] * GPR[rt])$
              $GPR[rd]_{31..0} \leftarrow ((HI_{31..0} \| LO_{31..0}) + (GPR[rs] * GPR[rt]))_{31..0}$
```

Exceptions:

None

MADD

Multiply and Add Word

31 26	25 21	20 16	15 6	5 0
SPECIAL2 011100	rs	rt	0 0000000000	MADD 000000

Format:

MADD rs, rt

VR5500

Purpose:

Combines multiplication and addition of 32-bit signed integers for execution.

Description:

This instruction multiplies the contents of general-purpose register *rs* by the contents of general-purpose register *rt*. It treats both the operands as signed integers. The result of this multiplication is added to a 64-bit value that combined special register *HI* and *LO*. The lower word of the 64-bit sum from this add operation is sign-extended and loaded to special register *LO* and the higher word is sign-extended and loaded to special register *HI*.

An integer overflow exception does not occur.

Operation:

```
32, 64   T:   temp1 ← GPR[rs] * GPR[rt]
              temp2 ← temp1 + ($HI_{31..0}$ || $LO_{31..0}$)
              LO ← $(temp2_{31})^{32}$ || $temp2_{31..0}$
              HI ← $(temp2_{63})^{32}$ || $temp2_{63..32}$
```

Exceptions:

None

MADDU

Multiply and Add Word Unsigned

31 26	25 21	20 16	15 6	5 0
SPECIAL2 011100	rs	rt	0 0000000000	MADDU 000001

Format:

MADDU rs, rt

VR5500

Purpose:

Combines multiplication and addition of 32-bit unsigned integers for execution.

Description:

This instruction multiplies the contents of general-purpose register *rs* by the contents of general-purpose register *rt*. It treats both the operands as unsigned integers. The result of this multiplication is added to a 64-bit value that combined special register *HI* and *LO*. The lower word of the 64-bit sum from this add operation is sign-extended and loaded to special register *LO* and the higher word is sign-extended and loaded to special register *HI*.
An integer overflow exception does not occur.

Operation:

```
32, 64   T:   $temp1 \leftarrow (0^{32} \| GPR[rs]) * (0^{32} \| GPR[rt])$
              $temp2 \leftarrow temp1 + (HI_{31..0} \| LO_{31..0})$
              $LO \leftarrow (temp2_{31})^{32} \| temp2_{31..0}$
              $HI \leftarrow (temp2_{63})^{32} \| temp2_{63..32}$
```

Exceptions:

None

MFC0

Move from System Control Coprocessor

31 26	25 21	20 16	15 11	10 0
COP0 010000	MF 00000	rt	rd	0 00000000000

Format:

MFC0 rt, rd

MIPS I

Description:

The contents of coprocessor register *rd* of the CP0 are loaded to 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] ← $(data_{31})^{32}$ || $data_{31..0}$
```

Exceptions:

Coprocessor unusable exception (64/32-bit user/supervisor mode if CP0 is disabled)

MFCz

Move from Coprocessor z

31 26	25 21	20 16	15 11	10 0
COPz 0100XX[Note]	MF 00000	rt	rd	0 00000000000

Format:

MFCz rt, rd

MIPS I

Description:

The contents of general-purpose register *rd* of the CPz are loaded to general-purpose register *rt*.

Operation:

```
32    T:       data ← CPR[z, rd]
      T + 1:   GPR[rt] ← data


64    T:       if $rd_0$ = 0 then
                 data ← CPR[z, $rd_{4..1}$ || 0]$_{31..0}$
               else
                 data ← CPR[z, $rd_{4..1}$ || 0]$_{63..32}$
               endif
      T + 1:   GPR[rt] ← $(data_{31})^{32}$ || data
```

Exceptions:

Coprocessor unusable exception

Note See the opcode table below, or **17.4 CPU Instruction Opcode Bit Encoding**.

Opcode Table:

	31	30	29	28	27	26	25	24	23	22	21	0
MFC0	0	1	0	0	0	0	0	0	0	0	0	

	31	30	29	28	27	26	25	24	23	22	21	0
MFC1	0	1	0	0	0	1	0	0	0	0	0	

	31	30	29	28	27	26	25	24	23	22	21	0
MFC2	0	1	0	0	1	0	0	0	0	0	0	

Opcode (bits 31–28) Coprocessor No. (bits 27–26) Coprocessor sub-opcode (bits 25–21)

Remark Coprocessor 2 is reserved in the V_R5500.

MFHI

Move from HI

31 26	25 16	15 11	10 6	5 0
SPECIAL 000000	0 0000000000	rd	0 00000	MFHI 010000

Format:

MFHI rd

MIPS I

Description:

The contents of special register *HI* are loaded to general-purpose register *rd*.

Operation:

```
32, 64    T:       GPR[rd] ← HI
```

Exceptions:

None

MFLO

Move from LO

31 26	25 16	15 11	10 6	5 0
SPECIAL 000000	0 0000000000	rd	0 00000	MFLO 010010

Format:

MFLO rd

MIPS I

Description:

The contents of special register *LO* are loaded to general-purpose register *rd*.

Operation:

```
32, 64    T:        GPR[rd] ← LO
```

Exceptions:

None

MFPC

Move from Performance Counter

31 26	25 21	20 16	15 11	10 6	5 1	0
COP0 010000	MF 00000	rt	CP0 25 11001	0 00000	reg	1 1

Format:

MFPC rt, reg

VR5500

Description:

This instruction loads the contents of performance counter *reg* of CP0 to general-purpose register *rt*. With the VR5500, only 0 and 1 are valid as *reg*.

Operation:

```
32      T:       data ← CPR[0, reg]
        T + 1:   GPR[rt] ← data

64      T:       data ← CPR[0, reg]
        T + 1:   GPR[rt] ← $(data_{31})^{32}$ || $data_{31..0}$
```

Exceptions:

Coprocessor unusable exception

MFPS

Move from Performance Event Specifier

31 26	25 21	20 16	15 11	10 6	5 1	0
COP0 010000	MF 00000	rt	CP0 25 11001	0 00000	reg	0 0

Format:

MFPS rt, reg **VR5500**

Description:

This instruction loads the contents of performance event specifier *reg* of CP0 to general-purpose register *rt*. With the VR5500, only 0 and 1 are valid as *reg*.

Operation:

```
32    T:      data ← CPR[0, reg]
      T + 1:  GPR[rt] ← data


64    T:      data ← CPR[0, reg]
      T + 1:  GPR[rt] ← $(data_{31})^{32}$ || $data_{31..0}$
```

Exceptions:

Coprocessor unusable exception

MOVN

Move Conditional on Not Zero

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	MOVN 001011

Format:

MOVN rd, rs, rt

MIPS IV

Purpose:

Tests the value of a general-purpose register and then conditionally moves the contents of a general-purpose register.

Description:

If the contents of general-purpose register *rt* are not 0, this instruction moves the contents of general-purpose register *rs* to general-purpose register *rd*.

Operation:

```
32, 64   T:    if GPR[rt] ≠ 0 then
                 GPR[rd] ← GPR[rs]
               endif
```

Exceptions:

Reserved instruction exception

Remark The value tested by this instruction is the result of comparison by the SLT, SLTI, SLTU, or SLTIU instruction with the condition established as true.

MOVZ

Move Conditional on Zero

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	MOVZ 001010

Format:

MOVZ rd, rs, rt

MIPS IV

Purpose:

Tests the value of a general-purpose register and then conditionally moves the contents of a general-purpose register.

Description:

If the contents of general-purpose register *rt* are 0, this instruction moves the contents of general-purpose register *rs* to general-purpose register *rd*.

Operation:

```
32, 64   T:    if GPR[rt] = 0 then
                 GPR[rd] ← GPR[rs]
               endif
```

Exceptions:

Reserved instruction exception

Remark The value tested by this instruction is the result of comparison by the SLT, SLTI, SLTU, or SLTIU instruction with the condition established as false.

MSAC

Multiply, Negate, Accumulate, and Move LO

31 26	25 21	20 16	15 11	10 0
SPECIAL 000000	rs	rt	rd	MSAC 00111011000

Format:

MSAC rd, rs, rt

VR5500

Purpose:

Combines multiplication and subtraction of 32-bit signed integers for execution.

Description:

This instruction multiplies the contents of general-purpose register *rs* by the contents of general-purpose register *rt*. It treats both the operands as 32-bit signed integers.

The lower 32 bits of special register *HI* and the lower 32 bits of special register *LO* are combined and used as an accumulator.

The result of multiplication is subtracted from the contents of the accumulator and the result of this subtraction is stored in the accumulator. The contents of the accumulator are treated as a 64-bit signed integer. The lower 32 bits of the result are also stored in general-purpose register *rd*.

An integer overflow exception does not occur.

Operation:

```
32, 64    T:    $HI_{31..0} \,||\, LO_{31..0} \leftarrow (HI_{31..0} \,||\, LO_{31..0}) - (GPR[rs] * GPR[rt])$
                $GPR[rd]_{31..0} \leftarrow ((HI_{31..0} \,||\, LO_{31..0}) - (GPR[rs] * GPR[rt]))_{31..0}$
```

Exceptions:

None

MSACHI

Multiply, Negate, Accumulate, and Move HI

31 26	25 21	20 16	15 11	10 0
SPECIAL 000000	rs	rt	rd	MSACHI 01111011000

Format:

MSACHI rd, rs, rt **VR5500**

Purpose:

Combines multiplication and subtraction of 32-bit signed integers for execution.

Description:

This instruction multiplies the contents of general-purpose register *rs* by the contents of general-purpose register *rt*. It treats both the operands as 32-bit signed integers.

The lower 32 bits of special register *HI* and the lower 32 bits of special register *LO* are combined and used as an accumulator.

The result of multiplication is subtracted from the contents of the accumulator and the result of this subtraction is stored in the accumulator. The contents of the accumulator are treated as a 64-bit signed integer. The higher 32 bits of the result are also stored in general-purpose register *rd*.

An integer overflow exception does not occur.

Operation:

```
32, 64   T:   $HI_{31..0} \| LO_{31..0} \leftarrow (HI_{31..0} \| LO_{31..0}) - (GPR[rs] * GPR[rt])$
              $GPR[rd]_{31..0} \leftarrow ((HI_{31..0} \| LO_{31..0}) - (GPR[rs] * GPR[rt]))_{63..32}$
```

Exceptions:

None

MSACHIU

Unsigned Multiply, Negate, Accumulate, and Move HI

31 26	25 21	20 16	15 11	10 0
SPECIAL 000000	rs	rt	rd	MSACHIU 01111011001

Format:

MSACHIU rd, rs, rt **VR5500**

Purpose:

Combines multiplication and subtraction of 32-bit unsigned integers for execution.

Description:

This instruction multiplies the contents of general-purpose register *rs* by the contents of general-purpose register *rt*. It treats both the operands as 32-bit unsigned integers.

The lower 32 bits of special register *HI* and the lower 32 bits of special register *LO* are combined and used as an accumulator.

The result of multiplication is subtracted from the contents of the accumulator and the result of this subtraction is stored in the accumulator. The contents of the accumulator are treated as a 64-bit unsigned integer. The higher 32 bits of the result are also stored in general-purpose register *rd*.

An integer overflow exception does not occur.

Operation:

```
32, 64   T:   $HI_{31..0} || LO_{31..0} \leftarrow (HI_{31..0} || LO_{31..0}) - (GPR[rs] * GPR[rt])$
              $GPR[rd]_{31..0} \leftarrow ((HI_{31..0} || LO_{31..0}) - (GPR[rs] * GPR[rt]))_{63..32}$
```

Exceptions:

None

MSACU

Unsigned Multiply, Negate, Accumulate, and Move LO

31 26	25 21	20 16	15 11	10 0
SPECIAL 000000	rs	rt	rd	MSACU 00111011001

Format:

MSACU rd, rs, rt **VR5500**

Purpose:

Combines multiplication and subtraction of 32-bit unsigned integers for execution.

Description:

This instruction multiplies the contents of general-purpose register *rs* by the contents of general-purpose register *rt*. It treats both the operands as 32-bit unsigned integers.

The lower 32 bits of special register *HI* and the lower 32 bits of special register *LO* are combined and used as an accumulator.

The result of multiplication is subtracted from the contents of the accumulator and the result of this subtraction is stored in the accumulator. The contents of the accumulator are treated as a 64-bit unsigned integer. The lower 32 bits of the result are also stored in general-purpose register *rd*.

An integer overflow exception does not occur.

Operation:

```
32, 64    T:    $HI_{31..0} \| LO_{31..0} \leftarrow (HI_{31..0} \| LO_{31..0}) - (GPR[rs] * GPR[rt])$
                $GPR[rd]_{31..0} \leftarrow ((HI_{31..0} \| LO_{31..0}) - (GPR[rs] * GPR[rt]))_{31..0}$
```

Exceptions:

None

MSUB

Multiply and Subtract Word

31 26	25 21	20 16	15 6	5 0
SPECIAL2 011100	rs	rt	0 0000000000	MSUB 000100

Format:

MSUB rs, rt

VR5500

Purpose:

Combines multiplication and subtraction of 32-bit signed integers for execution.

Description:

This instruction multiplies the contents of general-purpose register *rs* by the contents of general-purpose register *rt*. It treats both the operands as signed integers. The result of this multiplication is subtracted from a 64-bit value that combined special register *HI* and *LO*. The lower word of the 64-bit sum from this add operation is sign-extended and loaded to special register *LO* and the higher word is sign-extended and loaded to special register *HI*.

An integer overflow exception does not occur.

Operation:

```
32, 64   T:   $\text{temp1} \leftarrow \text{GPR[rs]} * \text{GPR[rt]}$
              $\text{temp2} \leftarrow (\text{HI}_{31..0} \parallel \text{LO}_{31..0}) - \text{temp1}$
              $\text{LO} \leftarrow (\text{temp2}_{31})^{32} \parallel \text{temp2}_{31..0}$
              $\text{HI} \leftarrow (\text{temp2}_{63})^{32} \parallel \text{temp2}_{63..32}$
```

Exceptions:

None

MSUBU

Multiply and Subtract Word Unsigned

31 26	25 21	20 16	15 6	5 0
SPECIAL2 011100	rs	rt	0 0000000000	MSUBU 000101

Format:

MSUBU rs, rt **VR5500**

Purpose:

Combines multiplication and subtraction of 32-bit unsigned integers for execution.

Description:

This instruction multiplies the contents of general-purpose register *rs* by the contents of general-purpose register *rt*. It treats both the operands as unsigned integers. The result of this multiplication is subtracted from a 64-bit value that combined special register *HI* and *LO*. The lower word of the 64-bit sum from this add operation is sign-extended and loaded to special register *LO* and the higher word is sign-extended and loaded to special register *HI*.

An integer overflow exception does not occur.

Operation:

```
32, 64    T:    temp1 ← $(0^{32}$ || GPR[rs] ) * $(0^{32}$ || GPR[rt] )
                temp2 ← $(HI_{31..0}$ || $LO_{31..0})$ – temp1
                LO ← $(temp2_{31})^{32}$ || $temp2_{31..0}$
                HI ← $(temp2_{63})^{32}$ || $temp2_{63..32}$
```

Exceptions:

None

MTC0

Move to System Control Coprocessor

31 26	25 21	20 16	15 11	10 0
COP0 010000	MT 00100	rt	rd	0 00000000000

Format:

MTC0 rt, rd

MIPS I

Description:

The contents of general-purpose register *rt* are loaded to coprocessor register *rd* of coprocessor 0.

Because the state of the virtual address translation system may be altered by this instruction, the operation of load instructions, store instructions, and TLB operations immediately prior to and after this instruction are undefined.

When using a register used by the MTC0 by means of instructions before and after it, refer to **CHAPTER 19 INSTRUCTION HAZARDS** and place the instructions in the appropriate location.

Operation:

```
32, 64   T:      data ← GPR[rt]
         T + 1:  CPR[0, rd] ← data
```

Exceptions:

Coprocessor unusable exception (64/32-bit user/supervisor mode if CP0 is disabled)

MTCz

Move to Coprocessor z

31 26	25 21	20 16	15 11	10 0
COPz 0100XX[Note]	MT 00100	rt	rd	0 00000000000

Format:

MTCz rt, rd

MIPS I

Description:

The contents of general-purpose register *rd* is loaded to CPz general-purpose register *rd*.

Operation:

```
32    T:      data ← GPR[rt]
      T + 1:  CPR[z, rd] ← data

64    T:      data ← GPR[rt]
      T + 1:  if $rd_0$ = 0 then
                CPR[z, $rd_{4..1}$ || 0] ← CPR[z, $rd_{4..1}$ || 0]$_{63..32}$ || data
              else
                CPR[z, $rd_{4..1}$ || 0] ← data || CPR[z, $rd_{4..1}$ || 0]$_{31..0}$
              endif
```

Exceptions:

Coprocessor unusable exception

Note See the opcode table below, or **17.4 CPU Instruction Opcode Bit Encoding**.

Opcode Table:

	31	30	29	28	27	26	25	24	23	22	21	0
MTC0	0	1	0	0	0	0	0	0	1	0	0	

	31	30	29	28	27	26	25	24	23	22	21	0
MTC1	0	1	0	0	0	1	0	0	1	0	0	

	31	30	29	28	27	26	25	24	23	22	21	0
MTC2	0	1	0	0	1	0	0	0	1	0	0	

Opcode (bits 31–28) Coprocessor No. (bits 27–26) Coprocessor sub-opcode (bits 25–21)

Remark Coprocessor 2 is reserved in the V$_R$5500.

MTHI

Move to HI

31 26	25 21	20 6	5 0
SPECIAL 000000	rs	0 000000000000000	MTHI 010001

Format:

MTHI rs

MIPS I

Description:

The contents of general-purpose register *rs* are loaded to special register *HI*.

If a MTHI operation is executed following a MULT, MULTU, DIV, or DIVU instruction, but before any MFLO, MFHI, MTLO, or MTHI instructions, the contents of special register *LO* are undefined.

Operation:

```
32, 64   T–2:   HI ← undefined
         T–1:   HI ← undefined
         T:     HI ← GPR[rs]
```

Exceptions:

None

MTLO

Move to LO

31 26	25 21	20 6	5 0
SPECIAL 000000	rs	0 000000000000000	MTLO 010011

Format:

MTLO rs

MIPS I

Description:

The contents of general-purpose register *rs* are loaded to special register *LO.*

If an MTLO operation is executed following a MULT, MULTU, DIV, or DIVU instruction, but before any MFLO, MFHI, MTLO, or MTHI instructions, the contents of special register *HI* are undefined.

Operation:

```
32, 64   T–2:   LO ← undefined
         T–1:   LO ← undefined
         T:     LO ← GPR[rs]
```

Exceptions:

None

MTPC

Move to Performance Counter

31 26	25 21	20 16	15 11	10 6	5 1	0
COP0 010000	MT 00100	rt	CP0 25 11001	0 00000	reg	1 1

Format:

MTPC rt, reg **VR5500**

Description:

This instruction loads the contents of general-purpose register *rt* to performance counter *reg* of CP0. With the VR5500, only 0 and 1 are valid as *reg*.

Operation:

```
32, 64   T:      data ← GPR[rt]
         T + 1:  CPR[0, reg] ← data
```

Exceptions:

Coprocessor unusable exception

MTPS

Move to Performance Event Specifier

31 26	25 21	20 16	15 11	10 6	5 1	0
COP0 010000	MT 00100	rt	CP0 25 11001	0 00000	reg	0 0

Format:

MTPS rt, reg **VR5500**

Description:

This instruction loads the contents of general-purpose register *rt* to performance event specifier *reg* of CP0. With the VR5500, only 0 and 1 are valid as *reg*.

Operation:

```
32, 64   T:       data ← GPR[rt]
         T + 1:   CPR[0, reg] ← data
```

Exceptions:

Coprocessor unusable exception

MUL

Multiply and Move LO

31 26	25 21	20 16	15 11	10 0
SPECIAL 000000	rs	rt	rd	MUL 00001011000

Format:

MUL rd, rs, rt

VR5500

Purpose:

Combines multiplication and transfer of 32-bit signed integers for execution.

Description:

This instruction multiplies the contents of general-purpose register *rs* by the contents of general-purpose register *rt*. It treats both the operands as 32-bit signed integers.

The lower 32 bits of special register *HI* and the lower 32 bits of special register *LO* are combined and used as an accumulator.

The result of multiplication is subtracted from the contents of the accumulator and the result of this multiplication is stored in the accumulator. The lower 32 bits of the result are also stored in general-purpose register *rd*.

An integer overflow exception does not occur.

Operation:

```
32, 64    T:    $HI_{31..0}$ || $LO_{31..0}$ ← GPR[rs] * GPR[rt]
                $GPR[rd]_{31..0}$ ← $(GPR[rs] * GPR[rt])_{31..0}$
```

Exceptions:

None

MUL64

Multiply and Move

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL2 011100	rs	rt	rd	0 00000	MUL64 000010

Format:

MUL64 rd, rs, rt

VR5500

Purpose:

Combines multiplication and transfer of 32-bit signed integers for execution.

Description:

This instruction multiplies the contents of general-purpose register *rs* by the contents of general-purpose register *rt*. It treats both the operands as 32-bit signed integers.

The result is also stored in general-purpose register *rd*.

An integer overflow exception does not occur.

The contents of special registers *HI* and *LO* are undefined after execution of this instruction.

Operation:

```
32, 64   T:   $GPR[rd]_{31..0} \leftarrow (GPR[rs] * GPR[rt])_{31..0}$
              $HI \leftarrow$ undefined
              $LO \leftarrow$ undefined
```

Exceptions:

None

MULHI

Multiply and Move HI

31 26	25 21	20 16	15 11	10 0
SPECIAL 000000	rs	rt	rd	MULHI 01001011000

Format:

MULHI rd, rs, rt

VR5500

Purpose:

Combines multiplication and transfer of 32-bit signed integers for execution.

Description:

This instruction multiplies the contents of general-purpose register *rs* by the contents of general-purpose register *rt*. It treats both the operands as 32-bit signed integers.

The lower 32 bits of special register *HI* and the lower 32 bits of special register *LO* are combined and used as an accumulator. The result of multiplication is stored in the accumulator. The higher 32 bits of the result are also stored in general-purpose register *rd*.

An integer overflow exception does not occur.

Operation:

```
32, 64    T:    $HI_{31..0} \parallel LO_{31..0} \leftarrow GPR[rs] * GPR[rt]$
                $GPR[rd]_{31..0} \leftarrow (GPR[rs] * GPR[rt])_{63..32}$
```

Exceptions:

None

MULHIU

Unsigned Multiply and Move HI

31 26	25 21	20 16	15 11	10 0
SPECIAL 000000	rs	rt	rd	MULHIU 01001011001

Format:

MULHIU rd, rs, rt **VR5500**

Purpose:

Combines multiplication and transfer of 32-bit unsigned integers for execution.

Description:

This instruction multiplies the contents of general-purpose register *rs* by the contents of general-purpose register *rt*. It treats both the operands as 32-bit unsigned integers.

The lower 32 bits of special register *HI* and the lower 32 bits of special register *LO* are combined and used as an accumulator. The result of multiplication is stored in the accumulator. The higher 32 bits of the result are also stored in general-purpose register *rd*.

An integer overflow exception does not occur.

Operation:

```
32, 64   T:   $HI_{31..0} \| LO_{31..0} \leftarrow GPR[rs] * GPR[rt]$
              $GPR[rd]_{31..0} \leftarrow (GPR[rs] * GPR[rt])_{63..32}$
```

Exceptions:

None

MULS

Multiply, Negate, and Move LO

31 26	25 21	20 16	15 11	10 0
SPECIAL 000000	rs	rt	rd	MULS 00011011000

Format:

MULS rd, rs, rt

VR5500

Purpose:

Combines multiplication and inversion of 32-bit signed integers for execution.

Description:

This instruction multiplies the contents of general-purpose register *rs* by the contents of general-purpose register *rt* and inverts the result. It treats both the operands as 32-bit signed integers.

The lower 32 bits of special register *HI* and the lower 32 bits of special register *LO* are combined and used as an accumulator, and the result of this inversion is stored in the accumulator. The lower 32 bits of the result are also stored in general-purpose register *rd*.

An integer overflow exception does not occur.

Operation:

```
32, 64    T:    $HI_{31..0} \mathbin{||} LO_{31..0} \leftarrow 0 - (GPR[rs] * GPR[rt])$
                $GPR[rd]_{31..0} \leftarrow (0 - (GPR[rs] * GPR[rt]))_{31..0}$
```

Exceptions:

None

MULSHI

Multiply, Negate, and Move HI

31 26	25 21	20 16	15 11	10 0
SPECIAL 000000	rs	rt	rd	MULSHI 01011011000

Format:

MULSHI rd, rs, rt

VR5500

Purpose:

Combines multiplication and inversion of 32-bit signed integers for execution.

Description:

This instruction multiplies the contents of general-purpose register *rs* by the contents of general-purpose register *rt* and inverts the result. It treats both the operands as 32-bit signed integers.

The lower 32 bits of special register *HI* and the lower 32 bits of special register *LO* are combined and used as an accumulator, and the result of this inversion is stored in the accumulator. The higher 32 bits of the result are also stored in general-purpose register *rd*.

An integer overflow exception does not occur.

Operation:

```
32, 64    T:    $HI_{31..0} || LO_{31..0} \leftarrow 0 - (GPR[rs] * GPR[rt])$
                $GPR[rd]_{31..0} \leftarrow (0 - (GPR[rs] * GPR[rt]))_{63..32}$
```

Exceptions:

None

MULSHIU

Unsigned Multiply, Negate, and Move HI

31 26	25 21	20 16	15 11	10 0
SPECIAL 000000	rs	rt	rd	MULSHIU 01011011001

Format:

MULSHIU rd, rs, rt

VR5500

Purpose:

Combines multiplication and inversion of 32-bit unsigned integers for execution.

Description:

This instruction multiplies the contents of general-purpose register *rs* by the contents of general-purpose register *rt* and inverts the result. It treats both the operands as 32-bit unsigned integers.

The lower 32 bits of special register *HI* and the lower 32 bits of special register *LO* are combined and used as an accumulator, and the result of this inversion is stored in the accumulator. The higher 32 bits of the result are also stored in general-purpose register *rd*.

An integer overflow exception does not occur.

Operation:

```
32, 64    T:    $HI_{31..0} \,||\, LO_{31..0} \leftarrow 0 - (GPR[rs] * GPR[rt])$
                $GPR[rd]_{31..0} \leftarrow (0 - (GPR[rs] * GPR[rt]))_{63..32}$
```

Exceptions:

None

MULSU

Unsigned Multiply, Negate, and Move LO

31 26	25 21	20 16	15 11	10 0
SPECIAL 000000	rs	rt	rd	MULSU 00011011001

Format:

MULSU rd, rs, rt

VR5500

Purpose:

Combines multiplication and inversion of 32-bit unsigned integers for execution.

Description:

This instruction multiplies the contents of general-purpose register *rs* by the contents of general-purpose register *rt* and inverts the result. It treats both the operands as 32-bit unsigned integers.

The lower 32 bits of special register *HI* and the lower 32 bits of special register *LO* are combined and used as an accumulator, and the result of this inversion is stored in the accumulator. The lower 32 bits of the result are also stored in general-purpose register *rd*.

An integer overflow exception does not occur.

Operation:

```
32, 64   T:   $HI_{31..0} || LO_{31..0} \leftarrow 0 – (GPR[rs] * GPR[rt])$
              $GPR[rd]_{31..0} \leftarrow (0 – (GPR[rs] * GPR[rt]))_{31..0}$
```

Exceptions:

None

MULT

Multiply

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 0000000000	MULT 011000

Format:

MULT rs, rt

MIPS I

Purpose:

Multiplies 32-bit signed integers.

Description:

The contents of general-purpose registers *rs* and *rt* are multiplied, treating both operands as signed 32-bit integer. No integer overflow exception occurs under any circumstances.

In 64-bit mode, the operands must be valid 32-bit, sign-extended values.

When the operation completes, the lower word of the double result is loaded to special register *LO*, and the higher word of the double result is loaded to special register *HI*. In 64-bit mode, the results will be sign-extended and stored.

Operation:

```
32    T–2:   LO ← undefined
             HI ← undefined
      T–1:   LO ← undefined
             HI ← undefined
      T:     t ← GPR[rs] * GPR[rt]
             LO ← $t_{31..0}$
             HI ← $t_{63..32}$


64    T–2:   LO ← undefined
             HI ← undefined
      T–1:   LO ← undefined
             HI ← undefined
      T:     t ← $GPR[rs]_{31..0}$ * $GPR[rt]_{31..0}$
             LO ← $(t_{31})^{32} \| t_{31..0}$
             HI ← $(t_{63})^{32} \| t_{63..32}$
```

Exceptions:

None

MULTU

Multiply Unsigned

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	0 0000000000	MULTU 011001

Format:

MULTU rs, rt

MIPS I

Purpose:

Multiplies 32-bit unsigned integers.

Description:

The contents of general-purpose register *rs* and the contents of general-purpose register *rt* are multiplied, treating both operands as unsigned values. No overflow exception occurs under any circumstances.

In 64-bit mode, the operands must be valid 32-bit, sign-extended values.

When the operation completes, the lower word of the double result is loaded to special register *LO*, and the higher word of the double result is loaded to special register *HI*. In 64-bit mode, the results will be sign-extended and stored.

Operation:

```
32    T–2:   LO ← undefined
             HI ← undefined
      T–1:   LO ← undefined
             HI ← undefined
      T:     t ← (0 || GPR[rs]) * (0 || GPR[rt])
             LO ← $t_{31..0}$
             HI ← $t_{63..32}$

64    T–2:   LO ← undefined
             HI ← undefined
      T–1:   LO ← undefined
             HI ← undefined
      T:     t ← (0 || $GPR[rs]_{31..0}$) * (0 || $GPR[rt]_{31..0}$)
             LO ← $(t_{31})^{32}$ || $t_{31..0}$
             HI ← $(t_{63})^{32}$ || $t_{63..32}$
```

Exceptions:

None

MULU

Unsigned Multiply and Move LO

31 26	25 21	20 16	15 11	10 0
SPECIAL 000000	rs	rt	rd	MULU 00001011001

Format:

MULU rd, rs, rt

VR5500

Purpose:

Combines multiplication and transfer of 32-bit unsigned integers for execution.

Description:

This instruction multiplies the contents of general-purpose register *rs* by the contents of general-purpose register *rt*. It treats both the operands as 32-bit unsigned integers.

The lower 32 bits of special register *HI* and the lower 32 bits of special register *LO* are combined and used as an accumulator. The result of multiplication is stored in the accumulator. The lower 32 bits of the result are also stored in general-purpose register *rd*.

An integer overflow exception does not occur.

Operation:

```
32, 64   T:   $HI_{31..0} \| LO_{31..0} \leftarrow GPR[rs] * GPR[rt]$
              $GPR[rd]_{31..0} \leftarrow (GPR[rs] * GPR[rt])_{31..0}$
```

Exceptions:

None

NOR

NOR

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	NOR 100111

Format:

NOR rd, rs, rt

MIPS I

Purpose:

Performs a bit-wise logical NOR operation.

Description:

The contents of general-purpose register *rs* are combined with the contents of general-purpose register *rt* in a bit-wise logical NOR operation. The result is stored in general-purpose register *rd*.

Operation:

```
32, 64    T:      GPR[rd] ← GPR[rs] nor GPR[rt]
```

Exceptions:

None

OR

OR

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	OR 100101

Format:

OR rd, rs, rt **MIPS I**

Purpose:

Performs a bit-wise logical OR operation.

Description:

The contents of general-purpose register *rs* are combined with the contents of general-purpose register *rt* in a bit-wise logical OR operation. 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

31 … 26	25 … 21	20 … 16	15 … 0
ORI 001101	rs	rt	immediate

Format:

ORI rt, rs, immediate **MIPS I**

Purpose:

Performs a bit-wise logical OR operation with a constant.

Description:

The 16-bit *immediate* is zero-extended and combined with the contents of general-purpose register *rs* in a bit-wise logical OR operation. The result is stored in general-purpose register *rt*.

Operation:

```
32    T:    GPR[rt] ← $\text{GPR[rs]}_{31..16}$ || (immediate or $\text{GPR[rs]}_{15..0}$)

64    T:    GPR[rt] ← $\text{GPR[rs]}_{63..16}$ || (immediate or $\text{GPR[rs]}_{15..0}$)
```

Exceptions:

None

PREF

Prefetch

(1/2)

31 26	25 21	20 16	15 0
PREF 110011	base	hint	offset

Format:

PREF hint, offset (base)

MIPS IV

Purpose:

Prefetches data from memory.

Description:

This instruction sign-extends a 16-bit *offset* and adds the result to the contents of general-purpose register *base* to generate a virtual address. It then loads the contents at the specified address position to the data cache.

Bits 20 to 16 (*hint*) of this instruction indicate how the loaded data is used. Note, however, that the contents of *hint* are only used for the processor to judge if prefetching by this instruction is valid or not, and do not affect the actual operation. *hint* indicates the following operations.

hint	Operation	Description
0	Load	Predicts that data is loaded (without modification). Fetches data as if it were loaded.
1 to 31	–	Reserved

This is an auxiliary instruction that improves the program performance. The generated address or the contents of *hint* do not change the status of the processor or system, or the meaning (purpose) of the program.

If this instruction causes a memory access to occur, the access type to be used is determined by the generated address. In other words, the access type used to load/store the generated address is also used for this instruction. However, an access to an uncached area does not occur.

If a translation entry to the specified memory position is not in the TLB, data cannot be prefetched from the map area. This is because no translation entry exists in TLB, it means that no access was made to the memory position recently, therefore, no effect can be expected even if data at such a memory position is prefetched.

Exceptions related to addressing do not occur as a result of executing this instruction. If the condition of an exception is detected, it is ignored, but the prefetch is not executed either. However, even if nothing is prefetched, processing that does not appear, such as writing back a dirty cache line, may be performed.

PREF

Prefetch

(2/2)

Operation:

```
32    T:    $\text{vAddr} \leftarrow ((\text{offset}_{15})^{16} \,||\, \text{offset}_{15..0}) + \text{GPR[base]}$
            (pAddr, CCA) $\leftarrow$ AddressTranslation (vAddr, DATA, LOAD)
            Prefetch (CCA, pAddr, vAddr, DATA, hint)

64    T:    $\text{vAddr} \leftarrow ((\text{offset}_{15})^{48} \,||\, \text{offset}_{15..0}) + \text{GPR[base]}$
            (pAddr, CCA) $\leftarrow$ AddressTranslation (vAddr, DATA, LOAD)
            Prefetch (CCA, pAddr, vAddr, DATA, hint)
```

Exceptions:

Reserved instruction exception

ROR

Rotate Right

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	1 00001	rt	rd	sa	ROR 000010

Format:

ROR rd, rt, sa **VR5500**

Purpose:

Arithmetically shifts a word to the right by the fixed number of bits.

Description:

This instruction shifts the contents of general-purpose register *rt* to the right by the number of bits specified by *sa*. The lower bit that is shifted out is inserted in the higher bit. The result is stored in general-purpose register *rd*.

Operation:

```
32, 64    T:    GPR[rd] ← $GPR[rt]_{sa-1..0}$ || $GPR[rt]_{31..sa}$
```

Exceptions:

None

RORV

Rotate Right Variable

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	1 00001	RORV 000110

Format:

RORV rd, rt, sa

VR5500

Purpose:

Arithmetically shifts a word to the right by the specified number of bits.

Description:

This instruction shifts the contents of general-purpose register *rt* to the right by the number of bits specified by the lower 5 bits of general-purpose register *rs*. The lower bit that is shifted out is inserted in the higher bit. The result is stored in general-purpose register *rd*.

Operation:

```
32, 64   T:   $s \leftarrow GPR[rs]_{4..0}$
              $GPR[rd] \leftarrow GPR[rt]_{s-1..0} \,||\, GPR[rt]_{31..s}$
```

Exceptions:

None

SB

Store Byte

31 26	25 21	20 16	15 0
SB 101000	base	rt	offset

Format:

SB rt, offset (base)

MIPS I

Purpose:

Stores a byte in memory.

Description:

The 16-bit *offset* is sign-extended and added to the contents of general-purpose register *base* to generate a virtual address. The least-significant byte of register *rt* is stored at the effective address.

Operation:

```
32   T:   $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0} \text{ xor } ReverseEndian^{3})$
          $byte \leftarrow vAddr_{2..0} \text{ xor } BigEndianCPU^{3}$
          $data \leftarrow GPR[rt]_{63-8*byte..0} \| 0^{8*byte}$
          StoreMemory (uncached, BYTE, data, pAddr, vAddr, DATA)

64   T:   $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0} \text{ xor } ReverseEndian^{3})$
          $byte \leftarrow vAddr_{2..0} \text{ xor } BigEndianCPU^{3}$
          $data \leftarrow GPR[rt]_{63-8*byte..0} \| 0^{8*byte}$
          StoreMemory (uncached, BYTE, data, pAddr, vAddr, DATA)
```

Exceptions:

TLB refill exception
TLB invalid exception
TLB modified exception
Bus error exception
Address error exception

SC

Store Conditional

(1/2)

31 26	25 21	20 16	15 0
SC 111000	base	rt	offset

Format:

SC rt, offset (base)

MIPS II

Purpose:

Stores a word in memory and completes atomic read-modify-write.

Description:

This instruction sign-extends a 16-bit *offset*, adds it to the contents of general-purpose register *base*, and generates a virtual address. The contents of general-purpose register *rt* are stored in the memory position of the specified address only when the LL bit is set.

If another processor or device has changed the target address after the previous LL instruction, or if the ERET instruction is executed between the LL and SC instructions, the contents of register rt are not stored in memory, and the SC instruction fails.

Whether the SC instruction has been successful or not is indicated by the contents of general-purpose register *rt* after this instruction has been executed. If the SC instruction is successful, the contents of general-purpose register *rt* are set to 1; they are cleared to 0 if the SC instruction has failed.

The operation of the SC instruction is undefined if the address is different from the address used for the last LL instruction.

This instruction can be used in the user mode. It is not necessary that CP0 be enabled.

An address error exception occurs if the lower 2 bits of the address are not 0.

If this instruction has failed and an exception occurs, the exception takes precedence.

This instruction is defined to maintain software compatibility with the other VR Series processors.

SC

Store Conditional

(2/2)

Operation:

```
32    T:    vAddr ← ((offset$_{15}$)$^{16}$ || offset$_{15..0}$ ) + 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] ← 0$^{31}$ || LLbit


64    T:    vAddr ← ((offset$_{15}$)$^{48}$ || offset$_{15..0}$) + 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] ← 0$^{63}$ || LLbit
```

Exceptions:

TLB refill exception
TLB invalid exception
TLB modified exception
Bus error exception
Address error exception

SCD

Store Conditional Doubleword

(1/2)

31 26	25 21	20 16	15 0
SCD 111100	base	rt	offset

Format:

SCD rt, offset (base) **MIPS III**

Purpose:

Stores a doubleword in memory and completes atomic read-modify-write.

Description:

This instruction sign-extends a 16-bit *offset*, adds it to the contents of general-purpose register *base*, and generates a virtual address. The contents of general-purpose register *rt* are stored in the memory position of the specified address only when the LL bit is set.

If another processor or device has changed the target address after the previous LLD instruction, or if the ERET instruction is executed between the LLD and SCD instructions, the contents of register rt are not stored in memory, and the SCD instruction fails.

Whether the SCD instruction has been successful or not is indicated by the contents of general-purpose register *rt* after this instruction has been executed. If the SCD instruction is successful, the contents of general-purpose register *rt* are set to 1; they are cleared to 0 if the SCD instruction has failed.

The operation of the SCD instruction is undefined if the address is different from the address used for the last LLD instruction.

This instruction can be used in the user mode. It is not necessary that CP0 be enabled.

An address error exception occurs if the lower 3 bits of the address are not 0.

If this instruction has failed and an exception occurs, the exception takes precedence.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

This instruction is defined to maintain software compatibility with the other VR Series processors.

Operation:

```
64    T:    vAddr ← $((\text{offset}_{15})^{48} \,\|\, \text{offset}_{15..0})$ + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            data ← GPR[rt]
            if LLbit then
              StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
            endif
            GPR[rt] ← $0^{63}$ || LLbit
```

Remark The higher 32 bits are ignored when a virtual address is generated in the 32-bit kernel mode.

SCD

Store Conditional Doubleword

(2/2)

Exceptions:

TLB refill exception
TLB invalid exception
TLB modified exception
Bus error exception
Address error exception
Reserved instruction exception (32-bit user/supervisor mode)

SD

Store Doubleword

31 26	25 21	20 16	15 0
SD 111111	base	rt	offset

Format:

SD rt, offset (base)

MIPS III

Purpose:

Stores a doubleword in memory.

Description:

The 16-bit *offset* is sign-extended and added to the contents of general-purpose register *base* to generate a virtual address. The contents of general-purpose register *rt* are stored at the memory location specified by the effective address.

An address error exception occurs if the lower 3 bits of the address are not 0.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
64    T:    vAddr ← $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15..0})$ + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            data ← GPR[rt]
            StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
```

Remark The higher 32 bits are ignored when a virtual address is generated in the 32-bit kernel mode.

Exceptions:

TLB refill exception

TLB invalid exception

TLB modified exception

Bus error exception

Address error exception

Reserved instruction exception (32-bit user/supervisor mode)

SDCz

Store Doubleword from Coprocessor z

(1/2)

31 26	25 21	20 16	15 0
SDCz 1111XX[Note]	base	rt	offset

Format:

SDCz rt, offset (base)

MIPS II

Purpose:

Stores a doubleword in memory from the coprocessor general-purpose register.

Description:

The 16-bit *offset* is sign-extended and added to the contents of general-purpose register *base* to generate a virtual address. The contents of the doubleword at CPz register *rt* are stored in the memory location specified by the effective address. Data to be stored is defined for each processor.
An address error exception occurs if the lower 3 bits of the address are not 0.
This instruction set to CP0 is invalid.
If CP1 is specified and if the FR bit of the status register is 0 and the least significant bit of the *rt* field is not 0, the operation of this instruction is undefined. If the FR bit is 1, an odd or even register is specified by *rt*.

Operation:

```
32    T:    vAddr ← $((\text{offset}_{15})^{16} \,||\, \text{offset}_{15..0})$ + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            data ← GPR[rt]
            StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)

64    T:    vAddr ← $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15..0})$ + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            data ← GPR[rt]
            StoreMemory (uncached, DOUBLEWORD, data, pAddr, vAddr, DATA)
```

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception
Coprocessor unusable exception

Note See the opcode table below, or **17.4 CPU Instruction Opcode Bit Encoding**.

SDCz

Store Doubleword from Coprocessor z

(2/2)

Opcode Table:

	31	30	29	28	27	26	0
SDC1	1	1	1	1	0	1	

	31	30	29	28	27	26	0
SDC2	1	1	1	1	1	0	

Opcode (bits 31–28)   Coprocessor No. (bits 27–26)

Remark Coprocessor 2 is reserved in the VR5500.

SDL

Store Doubleword Left

(1/3)

31 26	25 21	20 16	15 0
SDL 101100	base	rt	offset

Format:

SDL rt, offset (base)

MIPS III

Purpose:

Stores the most significant part of a doubleword in unaligned memory.

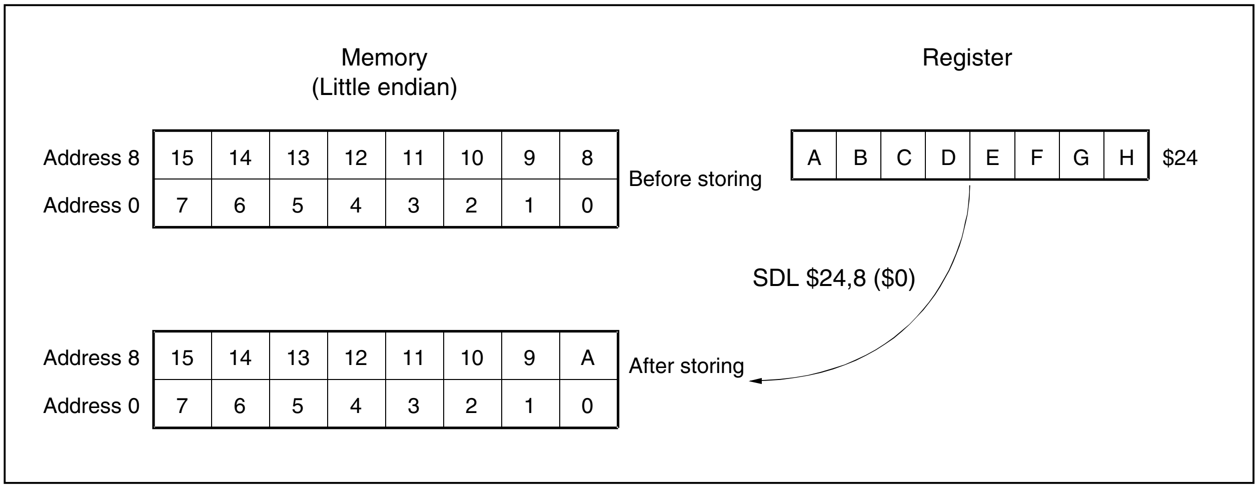
Description:

This instruction can be used in combination with the SDR instruction when storing a doubleword data in the register in a doubleword that does not exist at a doubleword boundary in the memory. The SDL instruction stores the higher word of the data, and the SDR instruction stores the lower word of the data in the memory.

The 16-bit *offset* is sign-extended and added to the contents of general-purpose register *base* to generate a virtual address. Among the doubleword data in the memory whose most significant byte is the byte specified by the virtual address, the higher portion of general-purpose register *rt* is stored in the memory at the same doubleword boundary as the target address.

The number of bytes to be stored varies from one to eight depending on the byte specified.

In other words, the most significant byte of general-purpose register *rt* is stored in the memory specified by the virtual address. As long as there are lower bytes among the bytes at the same doubleword boundary, the operation to store the byte in the next byte of the memory will be continued.



SDL

Store Doubleword Left

(2/3)

An address error exception caused by the specified address not being aligned at a doubleword boundary does not occur.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
64    T:    vAddr ← ((offset$_{15}$)$^{48}$ || offset$_{15..0}$) + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← pAddr$_{PSIZE-1..3}$ || (pAddr$_{2..0}$ xor ReverseEndian$^{3}$)
            if BigEndianMem = 0 then
              pAddr ← pAddr$_{31..3}$ || 0$^{3}$
            endif
            byte ← vAddr$_{2..0}$ xor BigEndianCPU$^{3}$
            data ← 0$^{56-8*byte}$ || GPR[rt]$_{63..56-8*byte}$
            StoreMemory (uncached, byte, data, pAddr, vAddr, DATA)
```

Remark The higher 32 bits are ignored when a virtual address is generated in the 32-bit kernel mode.

SDL

Store Doubleword Left

(3/3)

The relationship between the address assigned to the SDL instruction and its result (each byte of the register) is shown below.

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

vAddr$_{2..0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	Destination	Type	Offset		Destination	Type	Offset	
			LEM	BEM			LEM	BEM
0	I J K L M N O A	0	0	7	A B C D E F G H	7	0	0
1	I J K L M N A B	1	0	6	I A B C D E F G	6	0	1
2	I J K L M A B C	2	0	5	I J A B C D E F	5	0	2
3	I J K L A B C D	3	0	4	I J K A B C D E	4	0	3
4	I J K A B C D E	4	0	3	I J K L A B C D	3	0	4
5	I J A B C D E F	5	0	2	I J K L M A B C	2	0	5
6	I A B C D E F G	6	0	1	I J K L M N A B	1	0	6
7	A B C D E F G H	7	0	0	I J K L M N O A	0	0	7

Remark *Type* AccessType (see **Figure 3-3 Byte Specification Related to Load and Store Instruction**) output to memory

Offset pAddr$_{2..0}$ output to memory

LEM Little-endian memory (BigEndianMem = 0)

BEM Big-endian memory (BigEndianMem = 1)

Exceptions:

TLB refill exception
TLB invalid exception
TLB modified exception
Bus error exception
Address error exception
Reserved instruction exception (32-bit user/supervisor mode)

SDR

Store Doubleword Right

(1/3)

31 — 26	25 — 21	20 — 16	15 — 0
SDR 101101	base	rt	offset

Format:

SDR rt, offset (base)

MIPS III

Purpose:

Stores the least significant part of a doubleword in unaligned memory.

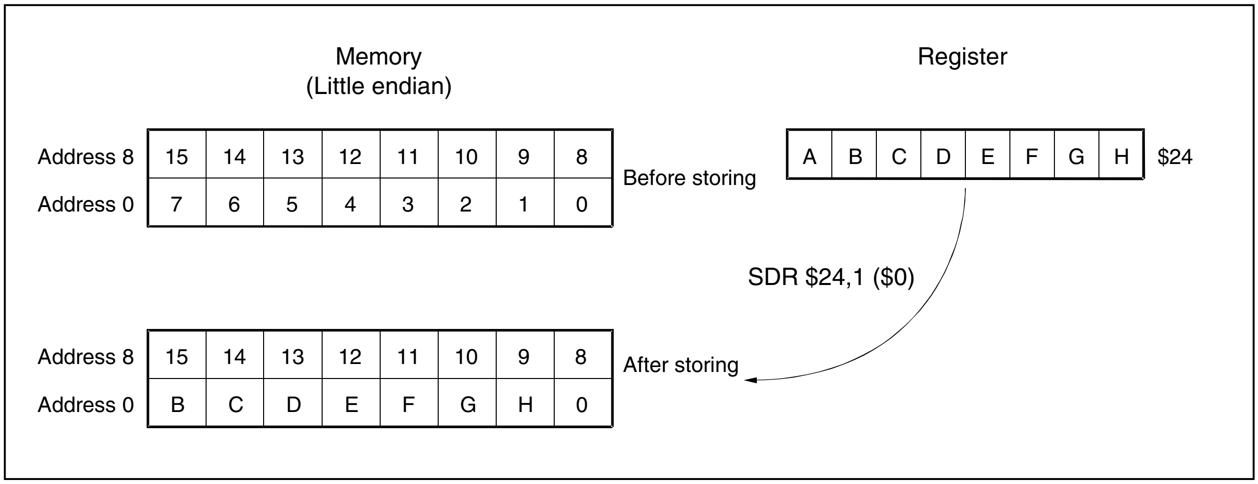
Description:

This instruction can be used in combination with the SDL instruction when storing a doubleword data in the register in a doubleword that does not exist at a doubleword boundary in the memory. The SDL instruction stores the higher word of the data, and the SDR instruction stores the lower word of the data in the memory.

The 16-bit *offset* is sign-extended and added to the contents of general-purpose register *base* to generate a virtual address. Among the doubleword data in the memory whose least significant byte is the byte specified by the virtual address, the lower portion of general-purpose register *rt* is stored in the memory at the same doubleword boundary as the target address.

The number of bytes to be stored varies from one to eight depending on the byte specified.

In other words, the least significant byte of general-purpose register *rt* is stored in the memory specified by the virtual address. As long as there are higher bytes among the bytes at the same doubleword boundary, the operation to store the byte in the next byte of the memory will be continued.



Before storing:

Address 8	15	14	13	12	11	10	9	8
Address 0	7	6	5	4	3	2	1	0

A	B	C	D	E	F	G	H	$24

After storing:

Address 8	15	14	13	12	11	10	9	8
Address 0	B	C	D	E	F	G	H	0

SDR **Store Doubleword Right**

(2/3)

An address error exception caused by the specified address not being aligned at a doubleword boundary does not occur.

This operation is defined in the 64-bit mode and 32-bit kernel mode. A reserved instruction exception occurs if this instruction is executed in the 32-bit user mode or supervisor mode.

Operation:

```
64    T:    vAddr ← $((\text{offset}_{15})^{48} \,||\, \text{offset}_{15..0})$ + GPR[base]
            (pAddr, uncached) ← AddressTranslation (vAddr, DATA)
            pAddr ← $\text{pAddr}_{\text{PSIZE}-1..3} \,||\, (\text{pAddr}_{2..0}$ xor $\text{ReverseEndian}^{3})$
            if BigEndianMem = 0 then
              pAddr ← $\text{pAddr}_{\text{PSIZE}-1..3} \,||\, 0^{3}$
            endif
            byte ← $\text{vAddr}_{2..0}$ xor $\text{BigEndianCPU}^{3}$
            data ← $\text{GPR[rt]}_{63-8*\text{byte}} \,||\, 0^{8*\text{byte}}$
            StoreMemory (uncached, DOUBLEWORD-byte, data, pAddr, vAddr, DATA)
```

Remark The higher 32 bits are ignored when a virtual address is generated in the 32-bit kernel mode.

SDR

Store Doubleword Right

(3/3)

The relationship between the address assigned to the SDR instruction and its result (each byte of the register) is shown below.

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

$vAddr_{2..0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	Destination	Type	Offset		Destination	Type	Offset	
			LEM	BEM			LEM	BEM
0	A B C D E F G H	7	0	0	H J K L M N O P	0	7	0
1	B C D E F G H P	6	1	0	G H K L M N O P	1	6	0
2	C D E F G H O P	5	2	0	F G H L M N O P	2	5	0
3	D E F G H N O P	4	3	0	E F G H M N O P	3	4	0
4	E F G H M N O P	3	4	0	D E F G H N O P	4	3	0
5	F G H L M N O P	2	5	0	C D E F G H O P	5	2	0
6	G H K L M N O P	1	6	0	B C D E F G H P	6	1	0
7	H J K L M N O P	0	7	0	A B C D E F G H	7	0	0

Remark *Type* AccessType (see **Figure 3-3 Byte Specification Related Load and Store Instruction**) output to memory

Offset $pAddr_{2..0}$ output to memory

LEM Little-endian memory (BigEndianMem = 0)

BEM Big-endian memory (BigEndianMem = 1)

Exceptions:

TLB refill exception

TLB invalid exception

TLB modified exception

Bus error exception

Address error exception

Reserved instruction exception (32-bit user/supervisor mode)

SH

Store Halfword

31 26	25 21	20 16	15 0
SH 101001	base	rt	offset

Format:

SH rt, offset (base)

MIPS I

Purpose:

Stores a halfword in memory.

Description:

The 16-bit *offset* is sign-extended and added to the contents of general-purpose register *base* to generate an unsigned effective address. The least-significant halfword of register *rt* is stored at the effective address.

An address error exception occurs if the least-significant bit of the address is not 0.

Operation:

```
32    T:    $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
            $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
            $pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0} xor(ReverseEndian^{2} \| 0))$
            $byte \leftarrow vAddr_{2..0} xor(BigEndianCPU^{2} \| 0)$
            $data \leftarrow GPR[rt]_{63-8*byte..0} \| 0^{8*byte}$
            StoreMemory (uncached, HALFWORD, data, pAddr, vAddr, DATA)

64    T:    $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
            $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
            $pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0} xor(ReverseEndian^{2} \| 0))$
            $byte \leftarrow vAddr_{2..0} xor(BigEndianCPU^{2} \| 0)$
            $data \leftarrow GPR[rt]_{63-8*byte..0} \| 0^{8*byte}$
            StoreMemory (uncached, HALFWORD, data, pAddr, vAddr, DATA)
```

Exceptions:

TLB refill exception

TLB invalid exception

TLB modified exception

Bus error exception

Address error exception

SLL

Shift Left Logical

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	SLL 000000

Format:

SLL rd, rt, sa

MIPS I

Purpose:

Logically shifts a word to the left by the fixed number of bits.

Description:

The contents of general-purpose register *rt* are shifted left by *sa* bits, inserting zeros into the lower bits.

The result is stored in general-purpose register *rd.* In 64-bit mode, the shifted 32-bit value is sign-extended and stored. When the shift amount is set to zero, SLL sign-extends lower 32 bits of a 64-bit value. Using this instruction, the 64-bit value can be generated from a 32-bit value.

Operation:

```
32    T:    $GPR[rd] \leftarrow GPR[rt]_{31-sa..0} \| 0^{sa}$

64    T:    $s \leftarrow 0 \| sa$
            $temp \leftarrow GPR[rt]_{31-s..0} \| 0^{s}$
            $GPR[rd] \leftarrow (temp_{31})^{32} \| temp$
```

Exceptions:

None

Caution SLL with a shift amount of zero may be treated as a NOP by some assemblers, at some optimization levels. If using SLL with a purpose of sign-extension, check the assembler specification.

SLLV

Shift Left Logical Variable

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SLLV 000100

Format:

SLLV rd, rt, rs

MIPS I

Purpose:

Logically shifts a word to the left by the specified number of bits.

Description:

The contents of general-purpose register *rt* are shifted left the number of bits specified by the lower 5 bits contained in general-purpose register *rs*, inserting zeros into the lower bits. The result is stored in general-purpose register *rd*. In 64-bit mode, the shifted 32-bit value is sign-extended and stored. When the shift amount is set to zero, SLLV sign-extends lower 32 bits of a 64-bit value. Using this instruction, the 64-bit value can be generated from a 32-bit value.

Operation:

```
32    T:    $s \leftarrow GPR[rs]_{4..0}$
            $GPR[rd] \leftarrow GPR[rt]_{(31-s)..0} \,||\, 0^{s}$

64    T:    $s \leftarrow 0 \,||\, GPR[rs]_{4..0}$
            $temp \leftarrow GPR[rt]_{(31-s)..0} \,||\, 0^{s}$
            $GPR[rd] \leftarrow (temp_{31})^{32} \,||\, temp$
```

Exceptions:

None

Caution SLLV with a shift amount of zero may be treated as a NOP by some assemblers, at some optimization levels. If using SLLV with a purpose of sign-extension, check the assembler specification.

SLT

Set on Less Than

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SLT 101010

Format:

SLT rd, rs, rt

MIPS I

Purpose:

Stores the result of unequal comparison.

Description:

The contents of general-purpose register *rt* are subtracted from the contents of general-purpose register *rs*. Considering both quantities as signed integers, if the contents of general-purpose register *rs* are less than the contents of general-purpose register *rt*, the result is set to one; otherwise the result is set to zero.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflows.

Operation:

```
32    T:    if GPR[rs] < GPR[rt] then
              GPR[rd] ← $0^{31}$ || 1
            else
              GPR[rd] ← $0^{32}$
            endif


64    T:    if GPR[rs] < GPR[rt] then
              GPR[rd] ← $0^{63}$ || 1
            else
              GPR[rd] ← $0^{64}$
            endif
```

Exceptions:

None

SLTI

Set on Less Than Immediate

31 26	25 21	20 16	15 0
SLTI 001010	rs	rt	immediate

Format:

SLTI rt, rs, immediate **MIPS I**

Purpose:

Stores the result of unequal comparison with a constant.

Description:

The 16-bit *immediate* is sign-extended and subtracted from the contents of general-purpose register *rs.* Considering both quantities as signed integers, if *rs* is less than the sign-extended *immediate*, the result is set to 1; otherwise the result is set to 0.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflows.

Operation:

```
32    T:    if GPR[rs] < $(\text{immediate}_{15})^{16}$ || $\text{immediate}_{15..0}$ then
              GPR[rt] ← $0^{31}$ || 1
            else
              GPR[rt] ← $0^{32}$
            endif

64    T:    if GPR[rs] < $(\text{immediate}_{15})^{48}$ || $\text{immediate}_{15..0}$ then
              GPR[rt] ← $0^{63}$ || 1
            else
              GPR[rt] ← $0^{64}$
            endif
```

Exceptions:

None

SLTIU

Set on Less Than Immediate Unsigned

31 26	25 21	20 16	15 0
SLTIU 001011	rs	rt	immediate

Format:

SLTIU rt, rs, immediate

MIPS I

Purpose:

Stores the result of unsigned unequal comparison with a constant.

Description:

The 16-bit *immediate* is sign-extended and subtracted from the contents of general-purpose register *rs.* Considering both quantities as unsigned integers, if *rs* is less than the sign-extended *immediate*, the result is set to 1; otherwise the result is set to 0.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflows.

Operation:

```
32    T:    if $(0 \| GPR[rs]) < (immediate_{15})^{16} \| immediate_{15..0}$ then
                GPR[rt] ← $0^{31} \| 1$
            else
                GPR[rt] ← $0^{32}$
            endif


64    T:    if $(0 \| GPR[rs]) < (immediate_{15})^{48} \| immediate_{15..0}$ then
                GPR[rt] ← $0^{63} \| 1$
            else
                GPR[rt] ← $0^{64}$
            endif
```

Exceptions:

None

SLTU

Set on Less Than Unsigned

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SLTU 101011

Format:

SLTU rd, rs, rt

MIPS I

Purpose:

Stores the result of unsigned unequal comparison.

Description:

The contents of general-purpose register *rt* are subtracted from the contents of general-purpose register *rs.* Considering both quantities as unsigned integers, if the contents of general-purpose register *rs* are less than the contents of general-purpose register *rt*, the result is set to 1; otherwise the result is set to 0.

No integer overflow exception occurs under any circumstances. The comparison is valid even if the subtraction used during the comparison overflows.

Operation:

```
32    T:    if (0 || GPR[rs] ) < 0 || GPR[rt] then
              GPR[rd] ← $0^{31}$ || 1
            else
              GPR[rd] ← $0^{32}$
            endif

64    T:    if (0 || GPR[rs] ) < 0 || GPR[rt] then
              GPR[rd] ← $0^{63}$ || 1
            else
              GPR[rd] ← $0^{64}$
            endif
```

Exceptions:

None

SRA

Shift Right Arithmetic

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	SRA 000011

Format:

SRA rd, rt, sa **MIPS I**

Purpose:

Arithmetically shifts a word to the right by the fixed number of bits.

Description:

The contents of general-purpose register *rt* are shifted right by the number of bits specified by *sa*, sign-extending the higher bits. The result is stored in general-purpose register *rd*. In 64-bit mode, the operand must be a valid sign-extended, 32-bit value.

Operation:

```
32    T:    $GPR[rd] \leftarrow (GPR[rt]_{31})^{sa} \parallel GPR[rt]_{31..sa}$

64    T:    $s \leftarrow 0 \parallel sa$
            $temp \leftarrow (GPR[rt]_{31})^{s} \parallel GPR[rt]_{31..s}$
            $GPR[rd] \leftarrow (temp_{31})^{32} \parallel temp$
```

Exceptions:

None

SRAV

Shift Right Arithmetic Variable

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SRAV 000111

Format:

SRAV rd, rt, rs

MIPS I

Purpose:

Arithmetically shifts a word to the right by the specified number of bits.

Description:

The contents of general-purpose register *rt* are shifted right by the number of bits specified by the lower 5 bits of general-purpose register *rs*, sign-extending the higher bits. The result is stored in general-purpose register *rd*. In 64-bit mode, the operand must be a valid sign-extended, 32-bit value.

Operation:

```
32    T:    $s \leftarrow GPR[rs]_{4..0}$
            $GPR[rd] \leftarrow (GPR[rt]_{31})^{s} \,||\, GPR[rt]_{31..s}$

64    T:    $s \leftarrow GPR[rs]_{4..0}$
            $temp \leftarrow (GPR[rt]_{31})^{s} \,||\, GPR[rt]_{31..s}$
            $GPR[rd] \leftarrow (temp_{31})^{32} \,||\, temp$
```

Exceptions:

None

SRL

Shift Right Logical

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	rt	rd	sa	SRL 000010

Format:

SRL rd, rt, sa

MIPS I

Purpose:

Logically shifts a word to the right by the fixed number of bits.

Description:

The contents of general-purpose register *rt* are shifted right by the number of bits specified by *sa*, inserting zeros into the higher bits. The result is stored in general-purpose register *rd*. In 64-bit mode, the operand must be a valid sign-extended, 32-bit value.

Operation:

```
32    T:    $GPR[rd] \leftarrow 0^{sa} \,||\, GPR[rt]_{31..sa}$

64    T:    $s \leftarrow 0 \,||\, sa$
            $temp \leftarrow 0^{s} \,||\, GPR[rt]_{31..s}$
            $GPR[rd] \leftarrow (temp_{31})^{32} \,||\, temp$
```

Exceptions:

None

SRLV

Shift Right Logical Variable

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SRLV 000110

Format:

SRLV rd, rt, rs

MIPS I

Purpose:

Logically shifts a word to the right by the specified number of bits.

Description:

The contents of general-purpose register *rt* are shifted right by the number of bits specified by the lower 5 bits of general-purpose register *rs,* inserting zeros into the higher bits. The result is stored in general-purpose register *rd*. In 64-bit mode, the operand must be a valid sign-extended, 32-bit value.

Operation:

```
32      T:      $s \leftarrow GPR[rs]_{4..0}$
                $GPR[rd] \leftarrow 0^{s} \mathbin{||} GPR[rt]_{31..s}$


64      T:      $s \leftarrow GPR[rs]_{4..0}$
                $temp \leftarrow 0^{s} \mathbin{||} GPR[rt]_{31..s}$
                $GPR[rd] \leftarrow (temp_{31})^{32} \mathbin{||} temp$
```

Exceptions:

None

SSNOP

Superscalar NOP

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	0 00000	0 00000	0 00000	1 00001	SLL 000000

Format:

SSNOP **VR5500**

Description:

This instruction consumes the execution time of one instruction without affecting the status of the processor or data.

Actually, execution of the next instruction is postponed until all the instructions executed before this instruction pass through the commit stage. If this instruction is in the branch delay slot, the CPU waits until all the instructions executed before the branch instruction immediately before pass through the commit stage.

Execution of the next instruction is also postponed until all writeback to memory by the load instruction that is executed to the non-blocking area before this instruction is completed.

Operation:

32, 64 T: $GPR0 \leftarrow GPR0_{30..0} \,||\, 0$

Exceptions:

None

SUB

Subtract

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SUB 100010

Format:

SUB rd, rs, rt

MIPS I

Purpose:

Subtracts a 32-bit integer. A trap is performed if an overflow occurs.

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 operands must be valid 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 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] ← $(temp_{31})^{32}$ || $temp_{31..0}$
```

Exceptions:

Integer overflow exception

SUBU

Subtract Unsigned

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	SUBU 100011

Format:

SUBU rd, rs, rt

MIPS I

Purpose:

Subtracts a 32-bit integer.

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 operands must be valid sign-extended, 32-bit values.

The only difference between this instruction and the SUB instruction is that SUBU never causes an integer overflow exception.

Operation:

```
32    T:    $GPR[rd] \leftarrow GPR[rs] - GPR[rt]$


64    T:    $temp \leftarrow GPR[rs] - GPR[rt]$
            $GPR[rd] \leftarrow (temp_{31})^{32} \| temp_{31..0}$
```

Exceptions:

None

SW

Store Word

31 26	25 21	20 16	15 0
SW 101011	base	rt	offset

Format:

SW rt, offset (base)

MIPS I

Purpose:

Stores a word in memory.

Description:

The 16-bit *offset* is sign-extended and added to the contents of general-purpose register *base* to generate a virtual address. The contents of general-purpose register *rt* are stored at the memory location specified by the effective address. An address error exception occurs if the lower 2 bits of the address are not 0.

Operation:

```
32    T:    $vAddr \leftarrow ((offset_{15})^{16} \,||\, offset_{15..0}) + GPR[base]$
            $(pAddr, uncached) \leftarrow$ AddressTranslation (vAddr, DATA)
            $data \leftarrow GPR[rt]_{31..0}$
            StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)

64    T:    $vAddr \leftarrow ((offset_{15})^{48} \,||\, offset_{15..0}) + GPR[base]$
            $(pAddr, uncached) \leftarrow$ AddressTranslation (vAddr, DATA)
            $data \leftarrow GPR[rt]_{31..0}$
            StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
```

Exceptions:

TLB refill exception
TLB invalid exception
TLB modified exception
Bus error exception
Address error exception

SWCz

Store Word from Coprocessor z

(1/2)

31 26	25 21	20 16	15 0
SWCz 1110XX[Note]	base	rt	offset

Format:

SWCz rt, offset (base) **MIPS I**

Purpose:

Stores a word in memory from the coprocessor general-purpose register.

Description:

The 16-bit *offset* is sign-extended and added to the contents of general-purpose register *base* to generate a virtual address. The contents of the CPz register *rt* are stored in the memory location specified by the effective address. Data to be stored is defined for each processor.

If the lower 2 bits of the address are not 0, an address error exception occurs.

This instruction set to CP0 is invalid.

Operation:

```
32  T:  $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
        $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
        $pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0} \text{ xor } (ReverseEndian \| 0^2))$
        $byte \leftarrow vAddr_{2..0} \text{ xor } (BigEndianCPU \| 0^2)$
        $data \leftarrow COPzSW (byte, rt)$
        StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)

64  T:  $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
        $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
        $pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0} \text{ xor } (ReverseEndian \| 0^2))$
        $byte \leftarrow vAddr_{2..0} \text{ xor } (BigEndianCPU \| 0^2)$
        $data \leftarrow COPzSW (byte, rt)$
        StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
```

Exceptions:

TLB refill exception
TLB invalid exception
Bus error exception
Address error exception
Coprocessor unusable exception

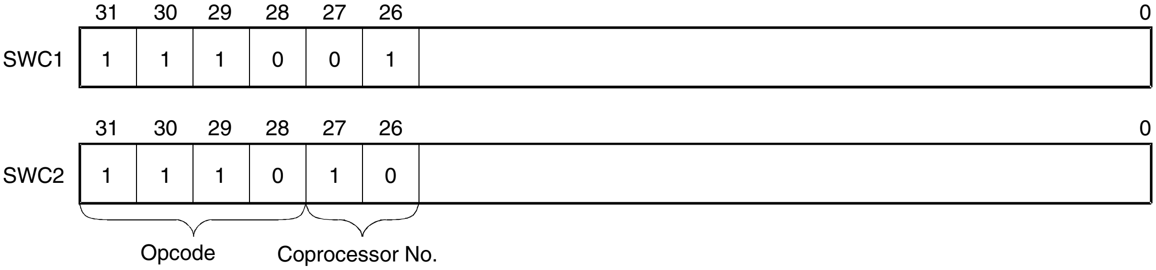
Note See the opcode table below, or **17.4 CPU Instruction Opcode Bit Encoding**.

SWCz

Store Word from Coprocessor z

(2/2)

Opcode Table:



	31	30	29	28	27	26	0
SWC1	1	1	1	0	0	1	

	31	30	29	28	27	26	0
SWC2	1	1	1	0	1	0	

Remark Coprocessor 2 is reserved in the VR5500.

SWL

Store Word Left

(1/3)

31 26	25 21	20 16	15 0
SWL 101010	base	rt	offset

Format:

SWL rt, offset (base)

MIPS I

Purpose:

Stores the most significant part of a word in unaligned memory.

Description:

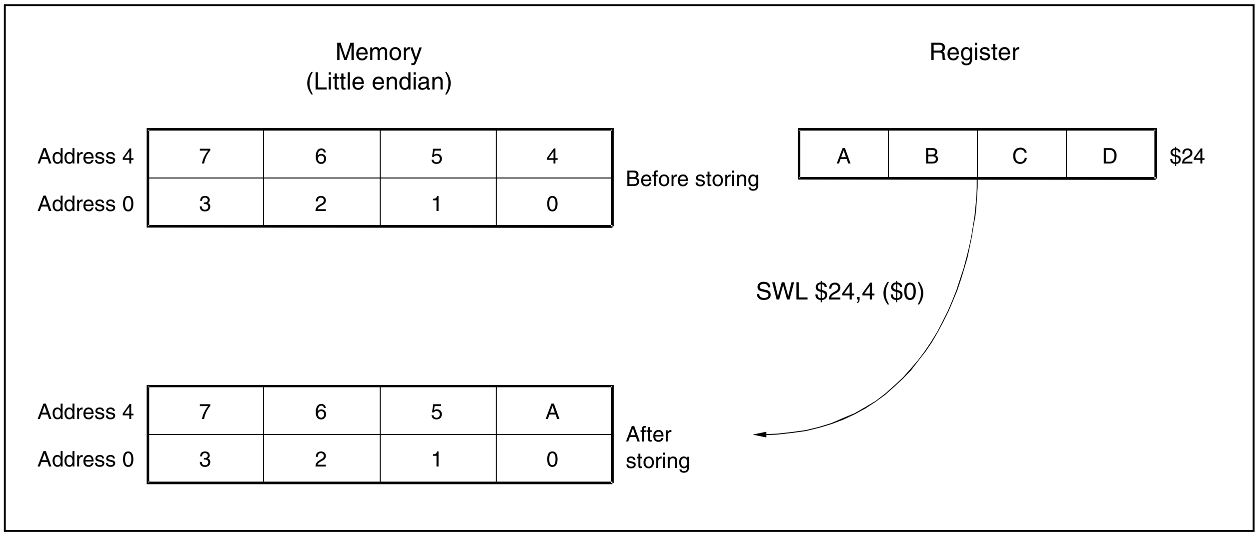
This instruction can be used in combination with the SWR instruction when storing a word data in the register in a word that does not exist at a word boundary in the memory. The SWL instruction stores the higher word of the data, and the SWR instruction stores the lower word of the data in the memory.

The 16-bit *offset* is sign-extended and added to the contents of general-purpose register *base* to generate a virtual address. Among the word data in the memory whose most significant byte is the byte specified by the virtual address, the higher portion of general-purpose register *rt* is stored in the memory at the same word boundary as the target address.

The number of bytes to be stored varies from one to four depending on the byte specified.

In other words, the most significant byte of general-purpose register *rt* is stored in the memory specified by the virtual address. As long as there are lower bytes among the bytes at the same word boundary, the operation to store the byte in the next byte of the memory will be continued.

An address error exception caused by the specified address not being aligned at a word boundary does not occur.



SWL

Store Word Left

(2/3)

Operation:

```
32   T:   $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0}\ xor\ ReverseEndian^{3})$
          if BigEndianMem = 0 then
            $pAddr \leftarrow pAddr_{31..2} \| 0^{2}$
          endif
          $byte \leftarrow vAddr_{1..0}\ xor\ BigEndianCPU^{2}$
          if $(vAddr_{2}\ xor\ BigEndianCPU) = 0$ then
            $data \leftarrow 0^{32} \| 0^{24-8*byte} \| GPR[rt]_{31..24-8*byte}$
          else
            $data \leftarrow 0^{24-8*byte} \| GPR[rt]_{31..24-8*byte} \| 0^{32}$
          endif
          StoreMemory (uncached, byte, data, pAddr, vAddr, DATA)

64   T:   $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
          $(pAddr, uncached) \leftarrow AddressTranslation (vAddr, DATA)$
          $pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0}\ xor\ ReverseEndian^{3})$
          if BigEndianMem = 0 then
            $pAddr \leftarrow pAddr_{31..2} \| 0^{2}$
          endif
          $byte \leftarrow vAddr_{1..0}\ xor\ BigEndianCPU^{2}$
          if $(vAddr_{2}\ xor\ BigEndianCPU) = 0$ then
            $data \leftarrow 0^{32} \| 0^{24-8*byte} \| GPR[rt]_{31..24-8*byte}$
          else
            $data \leftarrow 0^{24-8*byte} \| GPR[rt]_{31..24-8*byte} \| 0^{32}$
          endif
          StoreMemory (uncached, byte, data, pAddr, vAddr, DATA)
```

SWL

Store Word Left

(3/3)

The relationship between the address assigned to the SWL instruction and its result (each byte of the register) is shown below.

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

vAddr$_{2..0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	Destination	Type	Offset		Destination	Type	Offset	
			LEM	BEM			LEM	BEM
0	I J K L M N O E	0	0	7	E F G H M N O P	3	4	0
1	I J K L M N E F	1	0	6	I E F G M N O P	2	4	1
2	I J K L M E F G	2	0	5	I J E F M N O P	1	4	2
3	I J K L E F G H	3	0	4	I J K E M N O P	0	4	3
4	I J K E M N O P	0	4	3	I J K L E F G H	3	0	4
5	I J E F M N O P	1	4	2	I J K L M E F G	2	0	5
6	I E F G M N O P	2	4	1	I J K L M N E F	1	0	6
7	E F G H M N O P	3	4	0	I J K L M N O E	0	0	7

Remark *Type* AccessType (see **Figure 3-3 Byte Specification Related Load and Store Instruction**) output to memory

Offset pAddr$_{2..0}$ output to memory

LEM Little-endian memory (BigEndianMem = 0)

BEM Big-endian memory (BigEndianMem = 1)

Exceptions:

TLB refill exception

TLB invalid exception

TLB modified exception

Bus error exception

Address error exception

SWR

Store Word Right

(1/3)

31 26	25 21	20 16	15 0
SWR 101110	base	rt	offset

Format:

SWR rt, offset (base)

MIPS I

Purpose:

Stores the least significant part of a word in unaligned memory.

Description:

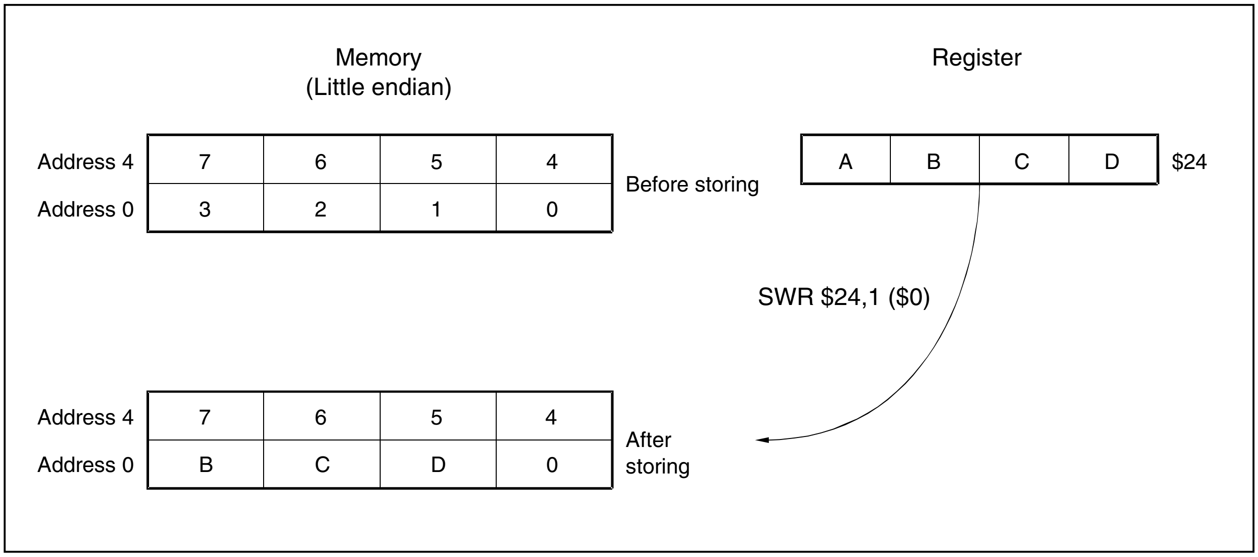
This instruction can be used in combination with the SWL instruction when storing a word data in the register in a word that does not exist at a word boundary in the memory. The SWL instruction stores the higher word of the data, and the SWR instruction stores the lower word of the data in the memory.

The 16-bit *offset* is sign-extended and added to the contents of general-purpose register *base* to generate a virtual address. Among the word data in the memory whose least significant byte is the byte specified by the virtual address, the lower portion of general-purpose register *rt* is stored in the memory at the same word boundary as the target address.

The number of bytes to be stored varies from one to four depending on the byte specified.

In other words, the least significant byte of general-purpose register *rt* is stored in the memory specified by the virtual address. As long as there are higher bytes among the bytes at the same word boundary, the operation to store the byte in the next byte of the memory will be continued.

An address error exception caused by the specified address not being aligned at a word boundary does not occur.



SWR

Store Word Right

(2/3)

Operation:

```
32    T:    $vAddr \leftarrow ((offset_{15})^{16} \| offset_{15..0}) + GPR[base]$
            $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
            $pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0}\ xor\ ReverseEndian^{3})$
            if BigEndianMem = 0 then
              $pAddr \leftarrow pAddr_{31..2} \| 0^{2}$
            endif
            $byte \leftarrow vAddr_{1..0}\ xor\ BigEndianCPU^{2}$
            if $(vAddr_{2}\ xor\ BigEndianCPU) = 0$ then
              $data \leftarrow 0^{32} \| GPR[rt]_{31-8*byte..0} \| 0^{8*byte}$
            else
              $data \leftarrow GPR[rt]_{31-8*byte} \| 0^{8*byte} \| 0^{32}$
            endif
            StoreMemory (uncached, WORD – byte, data, pAddr, vAddr, DATA)


64    T:    $vAddr \leftarrow ((offset_{15})^{48} \| offset_{15..0}) + GPR[base]$
            $(pAddr, uncached) \leftarrow AddressTranslation\ (vAddr, DATA)$
            $pAddr \leftarrow pAddr_{PSIZE-1..3} \| (pAddr_{2..0}\ xor\ ReverseEndian^{3})$
            if BigEndianMem = 0 then
              $pAddr \leftarrow pAddr_{31..2} \| 0^{2}$
            endif
            $byte \leftarrow vAddr_{1..0}\ xor\ BigEndianCPU^{2}$
            if $(vAddr_{2}\ xor\ BigEndianCPU) = 0$ then
              $data \leftarrow 0^{32} \| GPR[rt]_{31-8*byte..0} \| 0^{8*byte}$
            else
              $data \leftarrow GPR[rt]_{31-8*byte} \| 0^{8*byte} \| 0^{32}$
            endif
            StoreMemory (uncached, WORD – byte, data, pAddr, vAddr, DATA)
```

SWR

Store Word Right

(3/3)

The relationship between the address assigned to the SWR instruction and its result (each byte of the register) is shown below.

Register	A	B	C	D	E	F	G	H
Memory	I	J	K	L	M	N	O	P

$vAddr_{2..0}$	BigEndianCPU = 0				BigEndianCPU = 1			
	Destination	Type	Offset		Destination	Type	Offset	
			LEM	BEM			LEM	BEM
0	I J K L E F G H	3	0	4	H J K L M N O P	0	7	0
1	I J K L F G H P	2	1	4	G H K L M N O P	1	6	0
2	I J K L G H O P	1	2	4	F G H L M N O P	2	5	0
3	I J K L H N O P	0	3	4	E F G H M N O P	3	4	0
4	E F G H M N O P	3	4	0	I J K L H N O P	0	3	4
5	F G H L M N O P	2	5	0	I J K L G H O P	1	2	4
6	G H K L M N O P	1	6	0	I J K L F G H P	2	1	4
7	H J K L M N O P	0	7	0	I J K L E F G H	3	0	4

Remark *Type* AccessType (see **Figure 3-3 Byte Specification Related Load and Store Instruction**) output to memory

Offset $pAddr_{2..0}$ output to memory

LEM Little-endian memory (BigEndianMem = 0)

BEM Big-endian memory (BigEndianMem = 1)

Exceptions:

TLB refill exception
TLB invalid exception
TLB modified exception
Bus error exception
Address error exception

SYNC

Synchronize

31 26	25 11	10 6	5 0
SPECIAL 000000	0 000000000000000	stype	SYNC 001111

Format:

SYNC

MIPS II

Purpose:

Determines the order in which the common memory is referenced by a load/store instruction in a multi-processor environment.

Description:

The SYNC instruction is executed as a NOP on the VR5500.

This instruction is defined to maintain software compatibility with the other VR Series processors.

Actually, execution of the next instruction is postponed until all the instructions executed before this instruction pass through the commit stage. If this instruction is in the branch delay slot, the CPU waits until all the instructions executed before the branch instruction immediately before pass through the commit stage.

Execution of the next instruction is postponed until all the system interface requests by the load/store instruction executed before this instruction are issued. In this way, external access or writeback to memory can be processed in the same sequence as the load/store instructions that are executed before or after the SYNC instruction. The CPU does not wait for issuance of a system interface request by an instruction other than a load/store instruction, or issuance of instruction fetch.

The processor treats *stype* field as 0 regardless of the value of this field.

Operation:

```
32, 64   T:      SyncOperation ()
```

Exceptions:

None

SYSCALL

System Call

31 26	25 6	5 0
SPECIAL 000000	code	SYSCALL 001100

Format:

SYSCALL

MIPS I

Purpose:

Generates a system call exception.

Description:

A system call exception occurs, immediately and unconditionally transferring control to the exception handler.
The *code* field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
32, 64    T:     SystemCallException
```

Exceptions:

System call exception

TEQ

Trap if Equal

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	code	TEQ 110100

Format:

TEQ rs, rt

MIPS II

Purpose:

Compares general-purpose registers and executes a conditional trap.

Description:

The contents of general-purpose register *rt* are compared to 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.

The *code* field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
32, 64   T:   if GPR[rs] = GPR[rt] then
                TrapException
              endif
```

Exceptions:

Trap exception

TEQI

Trap if Equal Immediate

31 26	25 21	20 16	15 0
REGIMM 000001	rs	TEQI 01100	immediate

Format:

TEQI rs, immediate **MIPS II**

Purpose:

Compares a general-purpose register and a constant and executes a conditional trap.

Description:

The 16-bit *immediate* is sign-extended and compared to 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] = $(\text{immediate}_{15})^{16} \,\|\, \text{immediate}_{15..0}$ then
              TrapException
            endif


64    T:    if GPR[rs] = $(\text{immediate}_{15})^{48} \,\|\, \text{immediate}_{15..0}$ then
              TrapException
            endif
```

Exceptions:

Trap exception

TGE

Trap if Greater Than or Equal

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	code	TGE 110000

Format:

TGE rs, rt

MIPS II

Purpose:

Compares general-purpose registers and executes a conditional trap.

Description:

The contents of general-purpose register *rt* are compared to the contents of general-purpose register *rs*. Considering both quantities as 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.

The *code* field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
32, 64   T:   if GPR[rs] ≥ GPR[rt] then
                TrapException
              endif
```

Exceptions:

Trap exception

TGEI

Trap if Greater Than or Equal Immediate

31 26	25 21	20 16	15 0
REGIMM 000001	rs	TGEI 01000	immediate

Format:

TGEI rs, immediate **MIPS II**

Purpose:

Compares a general-purpose register and a constant and executes a conditional trap.

Description:

The 16-bit *immediate* is sign-extended and compared to the contents of general-purpose register *rs*. Considering both quantities as 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 $GPR[rs] \geq (immediate_{15})^{16} \| immediate_{15..0}$ then
              TrapException
            endif

64    T:    if $GPR[rs] \geq (immediate_{15})^{48} \| immediate_{15..0}$ then
              TrapException
            endif
```

Exceptions:

Trap exception

TGEIU

Trap if Greater Than or Equal Immediate Unsigned

31 26	25 21	20 16	15 0
REGIMM 000001	rs	TGEIU 01001	immediate

Format:

TGEIU rs, immediate

MIPS II

Purpose:

Compares a general-purpose register and a constant and executes a conditional trap.

Description:

The 16-bit *immediate* is sign-extended and compared to the contents of general-purpose register *rs*. Considering both quantities as 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 $(0 \,||\, \mathrm{GPR}[rs]) \geq (0 \,||\, (\mathrm{immediate}_{15})^{16} \,||\, \mathrm{immediate}_{15..0})$ then
              TrapException
            endif

64    T:    if $(0 \,||\, \mathrm{GPR}[rs]) \geq (0 \,||\, (\mathrm{immediate}_{15})^{48} \,||\, \mathrm{immediate}_{15..0})$ then
              TrapException
            endif
```

Exceptions:

Trap exception

TGEU

Trap if Greater Than or Equal Unsigned

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	code	TGEU 110001

Format:

TGEU rs, rt

MIPS II

Purpose:

Compares general-purpose registers and executes a conditional trap.

Description:

The contents of general-purpose register *rt* are compared to the contents of general-purpose register *rs*. Considering both quantities as 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.

The *code* field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
32, 64   T:    if (0 || GPR[rs] ) ≥ (0 || GPR[rt] ) then
                 TrapException
               endif
```

Exceptions:

Trap exception

TLBP

Probe TLB for Matching Entry

31 26	25	24 6	5 0
COP0 010000	CO 1	0 0000000000000000000	TLBP 001000

Format:

TLBP **MIPS I**

Description:

The Index register is loaded with the address of the TLB entry whose contents match the contents of the EntryHi register. If no TLB entry matches, the higher bit of the Index register is set. If two or more TLB entries that match the contents of the EntryHi register have been found, the TS bit of the Status register is set to 1, and a TLB refill exception occurs.

The operation is undefined if this instruction is executed immediately after the TLBP instruction and if an operation related to memory referencing takes place.

This operation is defined in kernel mode or when CP0 is enabled. Execution of this instruction in user/supervisor mode or when CP0 is not enabled causes a coprocessor unusable exception.

Operation:

```
32   T:   Index ← 1 || $0^{25}$ || $\text{Undefined}^{6}$
          for i in 0..TLBEntries – 1
            if ((TLB[i]$_{95..77}$ and not TLB[I]$_{120..109}$)
            = (EntryHi$_{31..12}$ and not TLB[i]$_{120..109}$)) and
            (TLB[i]$_{76}$ or (TLB[i]$_{71..64}$ = EntryHi$_{7..0}$)) then
                 Index ← $0^{26}$ || i$_{5..0}$
            endif
          endfor


64   T:   Index ← 1 || $0^{25}$ || $\text{Undefined}^{6}$
          for i in 0..TLBEntries – 1
            if (TLB[i]$_{171..141}$ and not ($0^{15}$ || TLB[i]$_{216..205}$))
            = (EntryHi$_{43..13}$ and not ($0^{15}$ || TLB[i]$_{216..205}$)) and
            (TLB[i]$_{140}$ or (TLB[i]$_{135..128}$ = EntryHi$_{7..0}$)) then
                 Index ← $0^{26}$ || i$_{5..0}$
            endif
          endfor
```

Exceptions:

Coprocessor unusable exception

TLB refill exception

TLBR

Read Indexed TLB Entry

31 26	25	24 6	5 0
COP0 010000	CO 1	0 0000000000000000000	TLBR 000001

Format:

TLBR **MIPS I**

Description:

The EntryHi and EntryLo registers are loaded with the contents of the TLB entry pointed at by the contents of the TLB Index register. The *G* bit (which controls ASID matching) read from the TLB is written to both of the EntryLo0 and EntryLo1 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 TLB Index register are greater than the number of TLB entries in the processor.

This operation is defined in kernel mode or when CP0 is enabled. Execution of this instruction in user/supervisor mode or when CP0 is not enabled causes a coprocessor unusable exception.

Operation:

```
32    T:    PageMask ← TLB[Index$_{5..0}$]$_{127..96}$
            EntryHi ← TLB[Index$_{5..0}$]$_{95..64}$ and not TLB[Index$_{5..0}$]$_{127..96}$
            EntryLo1 ← TLB[Index$_{5..0}$]$_{63..32}$
            EntryLo0 ← TLB[Index$_{5..0}$]$_{31..0}$


64    T:    PageMask ← TLB[Index$_{5..0}$]$_{255..192}$
            EntryHi ← TLB[Index$_{5..0}$]$_{191..128}$ and not TLB[Index$_{5..0}$]$_{255..192}$
            EntryLo1 ← TLB[Index$_{5..0}$]$_{127..65}$ || TLB[Index$_{5..0}$]$_{140}$
            EntryLo0 ← TLB[Index$_{5..0}$]$_{63..1}$ || TLB[Index$_{5..0}$]$_{140}$
```

Exceptions:

Coprocessor unusable exception

TLBWI

Write Indexed TLB Entry

31 26	25	24 6	5 0
COP0 010000	CO 1	0 0000000000000000000	TLBWI 000010

Format:

TLBWI

MIPS I

Description:

The TLB entry pointed at by the contents of the TLB Index register is loaded with the contents of the EntryHi and EntryLo registers. The *G* bit of the TLB is written with the logical AND of the *G* bits in the EntryLo0 and EntryLo1 registers.

The operation is invalid if the contents of the TLB Index register are greater than the number of TLB entries in the processor.

Operation:

32, 64 T: TLB[Index$_{5..0}$] ← PageMask || (EntryHi and not PageMask) || EntryLo1 || EntryLo0

Exceptions:

Coprocessor unusable exception

TLBWR

Write Random TLB Entry

31 26	25	24 6	5 0
COP0 010000	CO 1	0 0000000000000000000	TLBWR 000110

Format:

TLBWR **MIPS I**

Description:

The TLB entry pointed at by the contents of the TLB Random register is loaded with the contents of the EntryHi and EntryLo registers. The *G* bit of the TLB is written with the logical AND of the *G* bits in the EntryLo0 and EntryLo1 registers.

Operation:

32, 64	T:	TLB[Random$_{5..0}$] ← PageMask || (EntryHi and not PageMask) || EntryLo1 || EntryLo0

Exceptions:

Coprocessor unusable exception

TLT

Trap if Less Than

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	code	TLT 110010

Format:

TLT rs, rt

MIPS II

Purpose:

Compares general-purpose registers and executes a conditional trap.

Description:

The contents of general-purpose register *rt* are compared to general-purpose register *rs*. Considering both quantities as signed integers, if the contents of general-purpose register *rs* are less than the contents of general-purpose register *rt*, a trap exception occurs.

The *code* field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
32, 64   T:   if GPR[rs] < GPR[rt] then
                TrapException
              endif
```

Exceptions:

Trap exception

TLTI

Trap if Less Than Immediate

31 26	25 21	20 16	15 0
REGIMM 000001	rs	TLTI 01010	immediate

Format:

TLTI rs, immediate

MIPS II

Purpose:

Compares a general-purpose register and a constant and executes a conditional trap.

Description:

The 16-bit *immediate* is sign-extended and compared to the contents of general-purpose register *rs*. Considering both quantities as 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] < $(\text{immediate}_{15})^{16}$ || $\text{immediate}_{15..0}$ then
              TrapException
            endif

64    T:    if GPR[rs] < $(\text{immediate}_{15})^{48}$ || $\text{immediate}_{15..0}$ then
              TrapException
            endif
```

Exceptions:

Trap exception

TLTIU

Trap if Less Than Immediate Unsigned

31 26	25 21	20 16	15 0
REGIMM 000001	rs	TLTIU 01011	immediate

Format:

TLTIU rs, immediate

MIPS II

Purpose:

Compares a general-purpose register and a constant and executes a conditional trap.

Description:

The 16-bit *immediate* is sign-extended and compared to the contents of general-purpose register *rs*. Considering both quantities as 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 || $(\text{immediate}_{15})^{16}$ || $\text{immediate}_{15..0}$) then
              TrapException
            endif

64    T:    if (0 || GPR[rs] ) < (0 || $(\text{immediate}_{15})^{48}$ || $\text{immediate}_{15..0}$) then
              TrapException
            endif
```

Exceptions:

Trap exception

TLTU

Trap if Less Than Unsigned

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	code	TLTU 110011

Format:

TLTU rs, rt

MIPS II

Purpose:

Compares general-purpose registers and executes a conditional trap.

Description:

The contents of general-purpose register *rt* are compared to general-purpose register *rs*. Considering both quantities as unsigned integers, if the contents of general-purpose register *rs* are less than the contents of general-purpose register *rt*, a trap exception occurs.

The *code* field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
32, 64    T:      if (0 || GPR[rs] ) < (0 || GPR[rt] ) then
                    TrapException
                  endif
```

Exceptions:

Trap exception

TNE

Trap if Not Equal

31 26	25 21	20 16	15 6	5 0
SPECIAL 000000	rs	rt	code	TNE 110110

Format:

TNE rs, rt

MIPS II

Purpose:

Compares general-purpose registers and executes a conditional trap.

Description:

The contents of general-purpose register *rt* are compared to 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.

The *code* field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Operation:

```
32, 64   T:   if GPR[rs] ≠ GPR[rt] then
                TrapException
              endif
```

Exceptions:

Trap exception

TNEI

Trap if Not Equal Immediate

31 26	25 21	20 16	15 0
REGIMM 000001	rs	TNEI 01110	immediate

Format:

TNEI rs, immediate

MIPS II

Purpose:

Compares a general-purpose register and a constant and executes a conditional trap.

Description:

The 16-bit *immediate* is sign-extended and compared to 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] $\neq$ $(\text{immediate}_{15})^{16}$ || $\text{immediate}_{15..0}$ then
              TrapException
            endif

64    T:    if GPR[rs] $\neq$ $(\text{immediate}_{15})^{48}$ || $\text{immediate}_{15..0}$ then
              TrapException
            endif
```

Exceptions:

Trap exception

WAIT

Wait

31 26	25	24 6	5 0
COP0 010000	CO 1	Implementation-dependent Information	WAIT 100000

Format:

WAIT

VR5500

Purpose:

Sets the CPU in the standby mode.

Description:

This instruction places the processor in the standby mode.

The processor is kept waiting by this instruction until all the instructions executed before pass through the commit stage. It stops the operation of the pipeline after all the system interface requests, instruction fetch, and writeback to memory have been completed. If all the bits 10 to 6 of the instruction code are cleared to 0, the processor also stops the clock supply. If these bits are not cleared, the clock continued to be supplied.

To release from the standby mode, execute either a reset, NMI request, or all of the enabled interrupts. When the processor has been released from the standby mode, an exception occurs, and the address of the instruction next to the WAIT instruction is stored in the EPC/ErrorEPC register.

The operation of the processor is undefined if this instruction is in the branch delay slot. The operation is also undefined if this instruction is executed when the EXL and ERL bits of the Status register are set to 1.

This operation is defined in kernel mode or when CP0 is enabled. Execution of this instruction in user/supervisor mode or when CP0 is not enabled causes a coprocessor unusable exception.

Operation:

```
32, 64   T:   Standby Operation ()
              if Implementation-dependent Information$_{4..0}$ = 0 then
                  pipeline clock stop
              else
                  pipeline clock not stop
              endif
```

Exceptions:

Coprocessor unusable exception

XOR

Exclusive OR

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	XOR 100110

Format:

XOR rd, rs, rt

MIPS I

Purpose:

Performs a bit-wise logical XOR operation.

Description:

The contents of general-purpose register *rs* are combined with the contents of general-purpose register *rt* in a bit-wise logical exclusive OR operation.

The result is stored in general-purpose register *rd.*

Operation:

```
32, 64    T:        GPR[rd] ← GPR[rs] xor GPR[rt]
```

Exceptions:

None

XORI

Exclusive OR Immediate

31 26	25 21	20 16	15 0
XORI 001110	rs	rt	immediate

Format:

XORI rt, rs, immediate **MIPS I**

Purpose:

Performs a bit-wise logical XOR operation with a constant.

Description:

The 16-bit *immediate* is zero-extended and combined with the contents of general-purpose register *rs* in a bit-wise logical exclusive OR operation.

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

17.4 CPU Instruction Opcode Bit Encoding

Figure 17-1 lists the VR5500 opcode (ISA and extended ISA) encoding.

Figure 17-1. CPU Instruction Opcode Bit Encoding (1/2)

Opcode

28...26 31...29	0	1	2	3	4	5	6	7
0	SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	COP0	COP1	COP2	COP1X	BEQL	BNEL	BLEZL	BGTZL
3	DADDIε	DADDIUε	LDLε	LDRε	SPECIAL2	*	*	*
4	LB	LH	LWL	LW	LBU	LHU	LWR	LWUε
5	SB	SH	SWL	SW	SDLε	SDRε	SWR	CACHEδ
6	LL	LWC1	*	PREF	LLDε	LDC1	*	LDε
7	SC	SWC1	*	*	SCDε	SDC1	*	SDε

SPECIAL function

2...0 5...3	0	1	2	3	4	5	6	7
0	SLL/SSNOP	*	SRLπ	SRA	SLLV	*	SRLVπ	SRAV
1	JR	JALR	MOVZ	MOVN	SYSCALL	BREAK	*	SYNC
2	MFHI	MTHI	MFLO	MTLO	DSLLVε	*	DSRLVπ	DSRAVε
3	MULTπ	MULTUπ	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	*	DSRLπ	DSRAε	DSLL32ε	*	DSRL32π	DSRA32ε

REGIMM rt

18...16 20...19	0	1	2	3	4	5	6	7
0	BLTZ	BGEZ	BLTZL	BGEZL	*	*	*	*
1	TGEI	TGEIU	TLTI	TLTIU	TEQI	*	TNEI	*
2	BLTZAL	BGEZAL	BLTZALL	BGEZALL	*	*	*	*
3	*	*	*	*	*	*	*	*

Figure 17-1. CPU Instruction Opcode Bit Encoding (2/2)

COPz rs

	23...21							
25, 24	0	1	2	3	4	5	6	7
0	MFπ	DMFε	CF	γ	MTπ	DMTε	CT	γ
1	BC	γ	γ	γ	γ	γ	γ	γ
2	CO							
3								

COPz rt

	18...16							
20...19	0	1	2	3	4	5	6	7
0	BCF	BCT	BCFL	BCTL	γ	γ	γ	γ
1	γ	γ	γ	γ	γ	γ	γ	γ
2	γ	γ	γ	γ	γ	γ	γ	γ
3	γ	γ	γ	γ	γ	γ	γ	γ

CP0 Function

	2...0							
5...3	0	1	2	3	4	5	6	7
0	γ	TLBRρ	TLBWIρ	γ	γ	γ	TLBWRρ	γ
1	TLBP	γ	γ	γ	γ	γ	γ	γ
2	γ	γ	γ	γ	γ	γ	γ	γ
3	ERETχ	γ	γ	γ	γ	γ	γ	γ
4	WAITξρ	γ	γ	γ	γ	γ	γ	γ
5	γ	γ	γ	γ	γ	γ	γ	γ
6	γ	γ	γ	γ	γ	γ	γ	γ
7	γ	γ	γ	γ	γ	γ	γ	γ

SPECIAL2 Function

	2...0							
5...3	0	1	2	3	4	5	6	7
0	MADD	MADDU	MUL64	γ	MSUB	MSUBU	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
4	CLZ	CLO	γ	γ	DCLZ	DCLO	γ	γ
5	γ	γ	γ	γ	γ	γ	γ	γ
6	γ	γ	γ	γ	γ	γ	γ	γ
7	γ	γ	γ	γ	γ	γ	γ	γ

Remark The meanings of the symbols in the above figures are as follows.

*: Operation codes marked with an asterisk cause reserved instruction exceptions in current VR5500 implementations and are reserved for future versions of the architecture.

γ: Operation codes marked with a gamma cause a reserved instruction exception. They are reserved for future versions of the architecture.

δ: Operation codes marked with a delta are valid only for processors in which CP0 is enabled, and cause a reserved instruction exception in other processors.

χ: Operation codes marked with a chi are valid only in the VR4000 and VR5000 Series.

ε: Operation codes marked with an epsilon are valid when the processor operates in 64-bit mode or 32-bit kernel mode. These instructions will cause a reserved instruction exception when the processor operates in 32-bit user/supervisor mode.

π: Operation codes marked with a pi are also used in instructions that were added to the VR5500, such as the sum-of-products operation and rotate instructions.

ξ: Operation codes marked with a xi are valid only in the VR5500.

ρ: Operation codes marked with a rho are valid only for operation in kernel mode or for processors in which CP0 is enabled. These instructions will cause a coprocessor unusable exception when the processor operates in 32-bit user/supervisor mode or in processors in which CP0 is disabled.

CHAPTER 18 FPU INSTRUCTION SET

This chapter outlines the floating-point instructions (FPU instructions) and explains the function of each instruction.

18.1 Type of Instruction

The FPU instructions are classified into the following three basic types.

- I type (immediate type) instructions, such as load and store instructions
- R type (register type) instructions, such as floating-point operation instructions using two or three registers
- Other instructions, such as branch and transfer instructions

The floating-point instructions are mapped to the MIPS coprocessor instructions. The MIPS architecture defines coprocessor 1 (CP1) as a floating-point unit.

The instruction types used for the load/store instructions are shown in Figure 18-1.

Figure 18-1. Load/Store Instruction Format

I type (immediate)

31 26	25 21	20 16	15 0
op	base	ft	offset
6	5	5	16

R type (register)

31 26	25 21	20 16	15 11	10 6	5 0
COP1X	base	index	0	fd	function
6	5	5	5	5	6

31 26	25 21	20 16	15 11	10 6	5 0
COP1X	base	index	fs	0	function
6	5	5	5	5	6

op, COP1X	6-bit opcode
base	5-bit base register specifier
index	5-bit index register specifier
ft	5-bit source (for store) or destination (for load) FPU register specifier
fs	5-bit source FPU register specifier
fd	5-bit destination FPU register specifier
offset	16-bit offset of signed immediate
function	6-bit function field

The R type load/store instructions (register + register addressing mode) have been added to the MIPS IV instruction set.

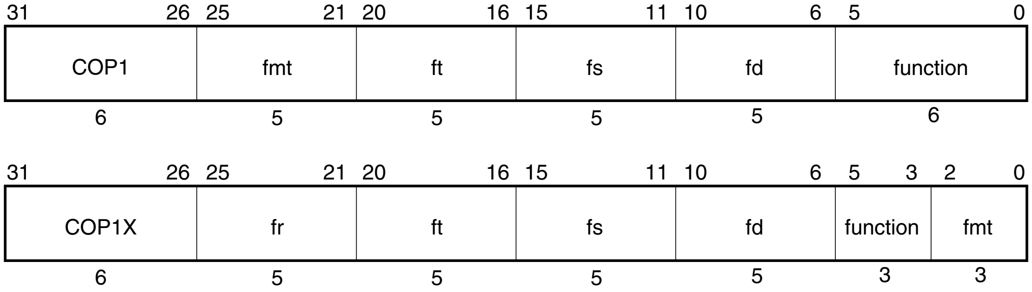
All the load/store instructions of the coprocessor reference data aligned at the word boundary. Therefore, the access type area of a word load/store instruction is always WORD, and the lower 2 bits of the address are always 0. The access type area of a doubleword load/store instruction is always DOUBLEWORD and the lower 3 bits of the address are always 0.

The byte in the accessed field that has the lowest byte address is specified as the address regardless of the byte order (endian). In a big-endian system, this byte is the leftmost byte. It is the rightmost byte in a little-endian system.

Figure 18-2 shows the instruction format of R type instructions used for operation instructions.

Figure 18-2. Operation Instruction Format

R type (register)



COP1, *COP1X* 6-bit opcode
fmt 5-bit or 3-bit format specifier
fs 5-bit source 1 register
ft 5-bit source 2 register
fr 5-bit source 3 register
fd 5-bit destination register
function 6-bit or 3-bit function field

Many formats can be applied to the floating-point instructions. The operand format of an instruction is specified by a 5-bit or 3-bit *fmt* field. The code of this field is shown in Table 18-1.

Table 18-1. Format Field Code

fmt(4:0)	fmt(2:0)	Mnemonic	Size	Format
0 to 15	–	Reserved		
16	0	S	Single precision (32 bits)	Binary floating point
17	1	D	Double precision (64 bits)	Binary floating point
18	2	Reserved		
19	3	Reserved		
20	4	W	32 bits	Binary fixed point
21	5	L	64 bits	Binary fixed point
22 to 31	6, 7	Reserved		

The *function* field indicates the floating-point operation to be executed.

18.1.1 Data format

Each operation is valid only in a specific data format. For execution, these formats and several operations are supported by emulation. However, valid combinations (those marked “V” in Table 18-2) must be supported. Combinations marked “R” in Table 18-2 are not defined by this architecture at present and cause an unimplemented operation exception. These combinations are reserved for future expansion of the architecture.

Table 18-2. Valid Format of FPU Instruction

Operation	Source Format			
	Single	Double	Word	Long Word
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

Remark V: Valid
R: Reserved

18.2 Instruction Notation Conventions

In this chapter, all variable subfields in an instruction format (such as *fs*, *ft*, *immediate*, etc.) are shown in lowercase names. The instruction names (e.g. ADD and SUB) are indicated by upper-case characters. For the sake of clarity, we sometimes use an alias for a variable subfield in the formats of specific instructions. For example, we use *base* instead of *fs* in the format for load and store instructions. Such an alias is always lower case, since it refers to a variable subfield.

The two subfields *op* and *function* of some instructions are 6-bit fixed values. These subfields are indicated by upper-case mnemonics. For example, the floating-point ADD instruction uses *op* = *COP1* and *function* = ADD. In the other cases, both uppercase and lowercase characters are used because a constant area and a variable area exist in one area together.

The architecture level at which the instruction was defined first is indicated on the right of the instruction format. The product name is also shown for instructions that may be incorporated differently depending on the product.

Figures with the actual bit encoding for all the mnemonics and the *function* field are located at the end of this chapter (**18.5 FPU Instruction Opcode Bit Encoding**), and the bit encoding also accompanies each instruction.

In the instruction descriptions that follow, the operation section describes the operation performed by each instruction using a high-level language notation. Special symbols used in the notation are described **in Table 17-1 CPU Instruction Operation Notaions**.

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

Example 1: GPR [rt] $\leftarrow$ immediate || 0^{16}

Sixteen zero bits are concatenated with an *immediate* value (typically 16 bits), and the 32-bit string is assigned to general-purpose register *rt*.

Example 2: $(\text{immediate}_{15})^{16}$ || $\text{immediate}_{15...0}$

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

Example 3: CPR [1, ft] $\leftarrow$ data

Assign data to general-purpose register *ft* of CP1, i.e., floating-point general-purpose register FGR.

The terms FGR and FPR are used in the explanation of each instruction. FGR means 32 FPU floating-point general-purpose registers FGR0 to FGR31, and FPR means the floating-point registers of FPUs.

The load/store instructions, and instructions that transfer data with the CPU use FGRs (may be described as CPR in some cases).

The transfer instructions, operation instructions, and conversion instructions in CP1 use the FPR.

- When the FR bit (bit 26) of the Status register is 0, only even FPRs are valid, and all the 32 FGRs are 32 bits wide.
- When the FR bit (bit 26) of the Status register is 1, both odd and even FPRs are valid, and all the 32 FGRs are 64 bits wide.

To get an FPR value, or to change the value of an FGR, the following routine is used in the description of a floating-point operation.

(1) 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] II FGR[fpr+0]
  end

StoreFPR (fpr, fmt, value):
  /* undefined for odd fpr */
  case fmt of
        S, W:
              FGR[fpr+1] <- undefined
              FGR[fpr+0] <- value
        D:
              FGR[fpr+1] <- value $_{63...32}$
              FGR[fpr+0] <- value $_{31...0}$
  end
```

(2) 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] <- undefined$^{32}$ II value
        D, L:
              FGR[fpr] <- value
  end
```

18.3 Cautions on Using FPU Instructions

18.3.1 Load and store instructions

All data transfers between the floating-point unit (FPU) and memory are executed by coprocessor load/store instructions. These instructions reference the general-purpose registers of the FPU. These instructions do not convert formats as they are independent of data formats. Therefore, a floating-point exception does not occur even if these instructions are executed.

Data can be directly transferred between the FPU and processor by using the MTC or MFC instruction. Like the floating-point load/store instructions, these instructions do not convert formats; therefore, a floating-point exception does not occur.

Five floating-point control registers can be used as the registers of the FPU. Only the CTC1 and CFC1 instructions are supported for these registers.

An instruction immediately after the load instruction can reference the contents of the register that has been loaded, but execution of that instruction may be delayed. Although the VR5500 can cover the load delay with an out-of-order mechanism, scheduling the load delay slot is recommended to improve the performance.

The operation of the load/store instruction differs depending on the bit width of the floating-point general-purpose register (FGR), as follows.

- When the FR bit of the Status register is 0
  The FGR is 32 bits wide. The sixteen even registers of the 32 FGRs can be accessed to hold single-precision floating-point data.
  To hold double-precision floating-point data, sixteen data items can be held by using an even register to hold the lower bits of the data and an odd register to hold the higher bits.
- When the FR bit of the Status register is 1
  The FGR is 64 bits wide. The lower bits of the 32 FGRs are accessed to hold single-precision floating-point data.
  To hold double-precision floating-point data, the 32 FGRs are accessed.

In the load and store descriptions, the functions listed below are used to summarize the handling of virtual addresses and physical memory.

Table 18-3. Load and Store Common Functions

Function	Meaning
AddressTranslation	Uses the TLB to find the physical address given the virtual address. The function fails and a TLB refill exception occurs if the required translation is not present in the TLB.
LoadMemory	Searches the specified data length (doubleword, word) containing the specified physical address in the cache and main memory and loads the contents. If the cache is enabled for this access, the contents are loaded to the cache.
StoreMemory	Searches the contents of the specified data length (doubleword, word) in the cache, write buffer, and main memory and stores the contents in the specified physical address.

18.3.2 Floating-point operation instructions

The operation instructions include all the floating-point operations executed by the FPU.

The instruction set of the FPU includes the following instructions.

- Floating-point addition
- Floating-point subtraction
- Floating-point multiplication
- Floating-point division
- Floating-point square root
- Floating-point reciprocal
- Reciprocal of floating-point square root
- Conversion between fixed-point and floating-point formats
- Conversion between floating-point formats
- Floating-point comparison

These instructions conform to IEEE Standard 754 to ensure accuracy. The result of an operation is the same as the result of infinite accuracy that is rounded in a specific format by using the rounding mode at that time.

The operand format must be specified for an instruction. All the instructions, except the conversion instructions, cannot execute operations in different formats.

18.3.3 FPU branch instruction

The FPU branch instruction can be used with the logic of its conditions inverted. Therefore, only 16 comparisons are necessary for all 32 conditions, as shown in Table 18-4.

The 4-bit condition code of a floating-point comparison instruction specifies a condition in the “True” column of this table. To invert the logic of the condition for the FPU branch instruction, the condition in the “False” column of this table is applied. If Not a Number (NaN) is specified as an operand, the result of comparing a numeric value other NaN is “Unordered” because the numeric great-and-small relationship cannot be established.

Table 18-4. Logical Inversion of Term Depending on True/False of Condition

Condition			Relationship				Occurrence of Invalid Operation Exception in Case of Unordered
Mnemonic		Code	Greater than	Less than	Equal to	Unordered	
True	Faulse						
F	T	0	F	F	F	F	Does not occur
UN	OR	1	F	F	F	T	Does not occur
EQ	NEQ	2	F	F	T	F	Does not occur
UEQ	OGL	3	F	F	T	T	Does not occur
OLT	UGE	4	F	T	F	F	Does not occur
ULT	OGE	5	F	T	F	T	Does not occur
OLE	UGT	6	F	T	T	F	Does not occur
ULE	OGT	7	F	T	T	T	Does not occur
SF	ST	8	F	F	F	F	Occurs
NGLE	GLE	9	F	F	F	T	Occurs
SEQ	SNE	10	F	F	T	F	Occurs
NGL	GL	11	F	F	T	T	Occurs
LT	NLT	12	F	T	F	F	Occurs
NGE	GE	13	F	T	F	T	Occurs
LE	NLE	14	F	T	T	F	Occurs
NGT	GT	15	F	T	T	T	Occurs

Remark F: False
T: True

18.4 FPU Instruction

This section describes the functions of FPU instructions in detail in alphabetical order.

The exception that may occur by executing each instruction is shown in the last of each instruction's description. For details of exceptions and their processes, see **CHAPTER 8 FLOATING-POINT EXCEPTIONS**.

ABS.fmt

Floating-point Absolute Value

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	ABS 000101

Format:

ABS.S fd, fs **MIPS I**

ABS.D fd, fs

Purpose:

Calculates the absolute value of a floating-point value.

Description:

This instruction calculates the absolute value of the contents of floating-point register *fs* and stores the result in floating-point register *fd*. The operand is processed as floating-point format fmt.

The absolute value is arithmetically calculated. If the operand is NaN, therefore, an invalid operation exception occurs.

This instruction is valid only in single-/double-precision floating-point formats.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

Operation:

```
32, 64   T:     StoreFPR (fd, fmt, AbsoluteValue (ValueFPR (fs, fmt)))
```

Exceptions:

Coprocessor unusable exception

Reserved instruction exception

Floating-point operation exception

Floating-point operation exception:

Unimplemented operation exception

Invalid operation exception

ADD.fmt

Floating-point Add

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	ft	fs	fd	ADD 000000

Format:

ADD.S fd, fs, ft **MIPS I**
ADD.D fd, fs, ft

Purpose:

Adds floating-point values.

Description:

This instruction adds the contents of floating-point register *fs* to the contents of floating-point register *ft*, and stores the result in floating-point register *fd*. The operands are processed as floating-point format *fmt*. The operation is executed as if it were of infinite accuracy, and the result is rounded in accordance with the current rounding mode.

This instruction is valid only in single-/double-precision floating-point formats.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

Operation:

```
32, 64    T:      StoreFPR (fd, fmt, ValueFPR (fs, fmt) + ValueFPR (ft, fmt))
```

Exceptions:

Coprocessor unusable exception
Reserved instruction exception
Floating-point operation exception

Floating-point operation exception:

Unimplemented operation exception
Invalid operation exception
Inexact operation exception
Overflow exception
Underflow exception

BC1F

Branch on FPU False (Coprocessor 1)

(1/2)

31 26	25 21	20 18	17	16	15 0
COP1 010001	BC 01000	cc	nd 0	tf 0	offset

Format:

BC1F offset **MIPS I**

BC1F cc, offset **MIPS IV**

Purpose:

Tests the floating-point condition code and executes a PC relative condition branch.

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 condition code bit (*cc* bit) of the floating-point control register (FCR31 or FCR25) specified by *cc* is false (0), execution branches to a branch address with a delay of one instruction. The *cc* bit of FCR31 and FCR25 is set by a floating-point comparison instruction (C.cond.fmt).

nd specifies whether the instruction in the branch delay slot is discarded if the branch condition is not satisfied. *tf* specifies which is used as the branch condition, True or False. The values of *nd* and *tf* are fixed for each instruction.

The MIPS I instruction set architecture provides only 1 bit of a floating-point condition code: the C bit in FCR31. Therefore, the *cc* field of the MIPS I, II, and III instruction set architectures must be 0. The MIPS IV instruction set architecture has seven additional condition code bits. The floating-point comparison instruction and conditional branch instruction specify the condition code bits to be set or tested. Both the assembler formats are valid with the MIPS IV instruction set architecture.

Remark The condition branch range of this instruction is ±128 KB because an 18-bit signed offset is used. To branch to an address outside this range, use the J or JR instruction.

BC1F

Branch on FPU False (Coprocessor 1)

(2/2)

Operation:

```
MIPS I, II, III
 32   T – 1:  condition ← FPConditionCode(0) = 0
      T:      target ← $(\text{offset}_{15})^{14}$ || offset || $0^{2}$
      T + 1:  if condition then
                PC ← PC + target
              endif


 64   T – 1:  condition ← FPConditionCode(0) = 0
      T:      target ← $(\text{offset}_{15})^{46}$ || offset || $0^{2}$
      T + 1:  if condition then
                PC ← PC + target
              endif


MIPS IV
 32   T – 1:  condition ← FPConditionCode(cc) = 0
      T:      target ← $(\text{offset}_{15})^{14}$ || offset || $0^{2}$
      T + 1:  if condition then
                PC ← PC + target
              end if


 64   T – 1:  condition ← FPConditionCode(cc) = 0
      T:      target ← $(\text{offset}_{15})^{46}$ || offset || $0^{2}$
      T + 1:  if condition then
                PC ← PC + target
              end if
```

Exceptions:

Coprocessor unusable exception

Reserved instruction exception

Floating-point operation exception

Floating-point operation exception:

Unimplemented operation exception

BC1FL

Branch on FPU False Likely (Coprocessor 1)

(1/2)

31 26	25 21	20 18	17	16	15 0
COP1 010001	BC 01000	cc	nd 1	tf 0	offset

Format:

BC1FL offset **MIPS II**

BC1FL cc, offset **MIPS IV**

Purpose:

Tests the floating-point condition code and executes a PC relative condition branch. Executes a delay slot only when a given branch condition is satisfied.

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 condition code bit (*cc* bit) of the floating-point control register (FCR31 or FCR25) specified by *cc* is false (0), execution branches to a branch address with a delay of one instruction. If the conditional branch is not taken, the instruction in the branch delay slot is discarded. The *cc* bit of FCR31 and FCR25 is set by a floating-point comparison instruction (C.cond.fmt).

nd specifies whether the instruction in the branch delay slot is discarded if the branch condition is not satisfied. *tf* specifies which is used as the branch condition, True or False. The values of *nd* and *tf* are fixed for each instruction.

The MIPS I instruction set architecture provides only 1 bit of a floating-point condition code: the C bit in FCR31. Therefore, the *cc* field of the MIPS I, II, and III instruction set architectures must be 0. The MIPS IV instruction set architecture has seven additional condition code bits. The floating-point comparison instruction and conditional branch instruction specify the condition code bits to be set or tested. Both the assembler formats are valid with the MIPS IV instruction set architecture.

Remarks 1. The condition branch range of this instruction is ±128 KB because an 18-bit signed offset is used. To branch to an address outside this range, use the J or JR instruction.

2. Use this instruction only when it is expected with a high probability (98% or higher) that a given branch condition is satisfied. If the branch condition is not satisfied or if the branch destination is not known, use the BC1F instruction.

BC1FL

Branch on FPU False Likely (Coprocessor 1)

(2/2)

Operation:

```
MIPS II, III
 32   T – 1:  condition ← FPConditionCode(0) = 0
      T:      target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
      T + 1:  if condition then
                PC ← PC + target
              else
                NulifyCurrentInstruction
              endif

 64   T – 1:  condition ← FPConditionCode(0) = 0
      T:      target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
      T + 1:  if condition then
                PC ← PC + target
              else
                NulifyCurrentInstruction
              endif

MIPS IV
 32   T – 1:  condition ← FPConditionCode(cc) = 0
      T:      target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
      T + 1:  if condition then
                PC ← PC + target
              else
                NulifyCurrentInstruction
              end if

 64   T – 1:  condition ← FPConditionCode(cc) = 0
      T:      target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
      T + 1:  if condition then
                PC ← PC + target
              else
                NulifyCurrentInstruction
              end if
```

Exceptions:

Coprocessor unusable exception
Reserved instruction exception
Floating-point operation exception

Floating-point operation exception:

Unimplemented operation exception

BC1T

Branch on FPU True (Coprocessor 1)

(1/2)

31 26	25 21	20 18	17	16	15 0
COP1 010001	BC 01000	cc	nd 0	tf 1	offset

Format:

BC1T offset **MIPS I**

BC1T cc, offset **MIPS IV**

Purpose:

Tests the floating-point condition code and executes a PC relative condition branch.

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 condition code bit (*cc* bit) of the floating-point control register (FCR31 or FCR25) specified by *cc* is true (1), execution branches to a branch address with a delay of one instruction. The *cc* bit of FCR31 and FCR25 is set by a floating-point comparison instruction (C.cond.fmt).

nd specifies whether the instruction in the branch delay slot is discarded if the branch condition is not satisfied. *tf* specifies which is used as the branch condition, True or False. The values of *nd* and *tf* are fixed for each instruction.

The MIPS I instruction set architecture provides only 1 bit of a floating-point condition code: the C bit in FCR31. Therefore, the *cc* field of the MIPS I, II, and III instruction set architectures must be 0. The MIPS IV instruction set architecture has seven additional condition code bits. The floating-point comparison instruction and conditional branch instruction specify the condition code bits to be set or tested. Both the assembler formats are valid with the MIPS IV instruction set architecture.

Remark The condition branch range of this instruction is ±128 KB because an 18-bit signed offset is used. To branch to an address outside this range, use the J or JR instruction.

BC1T

Branch on FPU True (Coprocessor 1)

(2/2)

Operation:

```
MIPS I, II, III
 32    T – 1:  condition ← FPConditionCode(0) = 1
       T:      target ← $(\text{offset}_{15})^{14} \,||\, \text{offset} \,||\, 0^2$
       T + 1:  if condition then
                 PC ← PC + target
               endif


 64    T – 1:  condition ← FPConditionCode(0) = 1
       T:      target ← $(\text{offset}_{15})^{46} \,||\, \text{offset} \,||\, 0^2$
       T + 1:  if condition then
                 PC ← PC + target
               endif


MIPS IV
 32    T – 1:  condition ← FPConditionCode(cc) = 1
       T:      target ← $(\text{offset}_{15})^{14} \,||\, \text{offset} \,||\, 0^2$
       T + 1:  if condition then
                 PC ← PC + target
               end if


 64    T – 1:  condition ← FPConditionCode(cc) = 1
       T:      target ← $(\text{offset}_{15})^{46} \,||\, \text{offset} \,||\, 0^2$
       T + 1:  if condition then
                 PC ← PC + target
               end if
```

Exceptions:

Coprocessor unusable exception

Reserved instruction exception

Floating-point operation exception

Floating-point operation exception:

Unimplemented operation exception

BC1TL

Branch on FPU True Likely (Coprocessor 1)

(1/2)

31 26	25 21	20 18	17	16	15 0
COP1 010001	BC 01000	cc	nd 1	tf 1	offset

Format:

BC1TL offset **MIPS II**

BC1TL cc, offset **MIPS IV**

Purpose:

Tests the floating-point condition code and executes a PC relative condition branch. Executes a delay slot only when a given branch condition is satisfied.

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 condition code bit (*cc* bit) of the floating-point control register (FCR31 or FCR25) specified by *cc* is true (1), execution branches to a branch address with a delay of one instruction. If the conditional branch is not taken, the instruction in the branch delay slot is discarded. The *cc* bit of FCR31 and FCR25 is set by a floating-point comparison instruction (C.cond.fmt).

nd specifies whether the instruction in the branch delay slot is discarded if the branch condition is not satisfied. *tf* specifies which is used as the branch condition, True or False. The values of *nd* and *tf* are fixed for each instruction.

The MIPS I instruction set architecture provides only 1 bit of a floating-point condition code: the C bit in FCR31. Therefore, the *cc* field of the MIPS I, II, and III instruction set architectures must be 0. The MIPS IV instruction set architecture has seven additional condition code bits. The floating-point comparison instruction and conditional branch instruction specify the condition code bits to be set or tested. Both the assembler formats are valid with the MIPS IV instruction set architecture.

Remarks 1. The condition branch range of this instruction is ±128 KB because an 18-bit signed offset is used. To branch to an address outside this range, use the J or JR instruction.

2. Use this instruction only when it is expected with a high probability (98% or higher) that a given branch condition is satisfied. If the branch condition is not satisfied or if the branch destination is not known, use the BC1T instruction.

BC1TL

Branch on FPU True Likely (Coprocessor 1)

(2/2)

Operation:

```
MIPS II, III
 32     T – 1:  condition ← FPConditionCode(0) = 1
        T:      target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
        T + 1:  if condition then
                  PC ← PC + target
                else
                  NulifyCurrentInstruction
                endif

 64     T – 1:  condition ← FPConditionCode(0) = 1
        T:      target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
        T + 1:  if condition then
                  PC ← PC + target
                else
                  NulifyCurrentInstruction
                endif

MIPS IV
 32     T – 1:  condition ← FPConditionCode(cc) = 1
        T:      target ← $(\text{offset}_{15})^{14}$ || offset || $0^2$
        T + 1:  if condition then
                  PC ← PC + target
                else
                  NulifyCurrentInstruction
                end if

 64     T – 1:  condition ← FPConditionCode(cc) = 1
        T:      target ← $(\text{offset}_{15})^{46}$ || offset || $0^2$
        T + 1:  if condition then
                  PC ← PC + target
                else
                  NulifyCurrentInstruction
                end if
```

Exceptions:

Coprocessor unusable exception
Reserved instruction exception
Floating-point operation exception

Floating-point operation exception:

Unimplemented operation exception

C.cond.fmt

Floating-point Compare

(1/3)

31 26	25 21	20 16	15 11	10 8	7 6	5 4	3 0
COP1 010001	fmt	ft	fs	cc	0 00	FC[Note] 11	cond[Note]

Format:

C.cond.S fs, ft **MIPS I**
C.cond.D fs, ft
C.cond.S cc, fs, ft **MIPS IV**
C.cond.D cc, fs, ft

Purpose:

Compares floating-point values and records the Boolean result of the comparison in a condition code.

Description:

This instruction compares the contents of floating-point register *fs* with the contents of floating-point register *ft* in accordance with comparison condition *cond*, and sets the result in the condition code bit (*cc* bit) of the floating-point control register (FCR31 or FCR25) specified by *cc*. The operands are processed as floating-point format *fmt*. If one of the values is NaN and if the most significant bit of comparison condition *cond* is set, an invalid operation exception occurs. If this exception occurs, the flag bits of FCR31 and FCR26 are set. If the invalid operation exception is enabled (if the enable bits of FCR31 and FCR28 are set), the comparison result is not set, and processing of the exception is started as is. If the enable bits are not set, only the comparison result is set to the cc bit, and the exception is not processed.

The comparison result is also used to test the FPU branch instruction.

Comparison is executed accurately, and neither overflow nor underflow occurs. One of four mutually exclusive relations, “less than”, “equal to”, “greater than”, and “Unordered (comparison impossible)”, occurs. If one or both the operands are NaN, the result of the comparison is always “Unordered”. For details of comparison condition *cond*, refer to **Table 18-4 Logical Inversion of Term Depending on True/False of Condition**.

The sign of 0 is ignored during comparison (+0 = –0).

This instruction is valid only in single-/double-precision floating-point formats.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

The MIPS I instruction set architecture provides only 1 bit of a floating-point condition code: the C bit in FCR31. Therefore, the *cc* field of the MIPS I, II, and III instruction set architectures must be 0. The MIPS IV instruction set architecture has seven additional condition code bits. The floating-point comparison instruction and conditional branch instruction specify the condition code bits to be set or tested. Both the assembler formats are valid with the MIPS IV instruction set architecture.

Note See **18.5 FPU Instruction Opcode Bit Encoding**.

C.cond.fmt

Floating-point Compare

(2/3)

If a floating-point operation instruction, including a comparison instruction, receives SignalingNaN (SNaN), it is regarded as an invalid operation condition. If comparison that also becomes an invalid operation with QuietNaN (QNaN), not only with SNaN, is used, a program that generates an error if NaN is used can be made easy. Consequently, a code that clearly checks QNaN that makes the result Unordered is unnecessary. Instead, an exception occurs if an invalid operation is detected, and errors are processed by an exception processing system. The case of comparison in which two numeric values are checked if they are equal to each other, and an error is detected if the result is Unordered, is shown below.

```
# To test QNaN clearly
   C.EQ.D    $f2, $f4            # Checks if two values are equal
   NOP
   BC1T      L2                  # To L2 if not equal
   C.UN.D    $f2, $f4            # Checks if result is Unordered if not equal
   BC1T             ERROR        # To error processing if Unordered
# Describes processing code if not equal
# Describes processing code if equal
L2:
  :

# To use comparison that reports QNaN
   C.SEQ.D   $f2, $f4            # Checks if two values are equal
   NOP
   BC1T      L2                  # To L2 if equal
   NOP
# Describes processing code if result is not Unordered
# Describes processing code if not equal
# Describes processing code if equal
L2:
  :
```

C.cond.fmt

Floating-point Compare

(3/3)

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)
              SetFPConditionCode (cc, condition)
```

Exceptions:

Coprocessor unusable exception
Reserved instruction exception
Floating-point operation exception

Floating-point operation exception:

Unimplemented operation exception
Invalid operation exception

CEIL.L.fmt

Floating-point Ceiling to Long Fixed-point Format

(1/2)

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	CEIL. L 001010

Format:

CEIL.L.S fd, fs

MIPS III

CEIL.L.D fd, fs

Purpose:

Rounds up a floating-point value to a 64-bit fixed-point value for conversion.

Description:

This instruction arithmetically converts the contents of floating-point register *fs* into a 64-bit floating-point format, and stores the result in floating-point register *fd*. The source operand is processed as floating-point format *fmt*.

The result is rounded toward the direction of +∞ regardless of the current rounding mode.

This instruction is valid only when converting from single-/double-precision floating-point formats.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

If the source operand is infinity or NaN, and if the result of rounding is outside the range of $2^{63} – 1$ to $– 2^{63}$, the flag bits of FCR31 and FCR26 are set to indicate an invalid operation. If an invalid operation exception is not enabled, the exception does not occur, and $2^{63} – 1$ is returned.

This operation is defined 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.

Operation:

```
64      T:      StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt), fmt, L))
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Coprocessor unusable exception

Floating-point operation exception

Reserved instruction exception (32-bit user/supervisor mode)

Floating-point operation exception:

Unimplemented operation exception

Invalid operation exception

Inexact operation exception

Overflow exception

CEIL.L.fmt

Floating-point Ceiling to Long Fixed-point Format

(2/2)

Caution The unimplemented operation exception occurs in the following cases.

- **If overflow occurs when the format is converted into a fixed-point format**
- **If the source operand is infinity**
- **If the source operand is NaN**

Specifically, the exception occurs if the value stored in floating-point register *fd* is outside the range of 2^{53} – 1 (0x001F FFFF FFFF FFFF) to –2^{53} (0xFFE0 0000 0000 0000).

CEIL.W.fmt

Floating-point Ceiling to Single Fixed-point Format

(1/2)

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	CEIL. W 001110

Format:

CEIL.W.S fd, fs **MIPS II**

CEIL.W.D fd, fs

Purpose:

Rounds up a floating-point value to a 32-bit fixed-point value for conversion.

Description:

This instruction arithmetically converts the contents of floating-point register *fs* into a 64-bit floating-point format, and stores the result in floating-point register *fd*. The source operand is processed as floating-point format *fmt*.

The result is rounded toward the direction of +∞ regardless of the current rounding mode.

This instruction is valid only when converting from single-/double-precision floating-point formats.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

If the source operand is infinity or NaN, and if the result of rounding is outside the range of $2^{31} – 1$ to $– 2^{31}$, the flag bits of FCR31 and FCR26 are set to indicate an invalid operation. If an invalid operation exception is not enabled, the exception does not occur, and $2^{31} – 1$ is returned.

Operation:

```
32, 64   T:     StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt), fmt, W))
```

Exceptions:

Coprocessor unusable exception

Floating-point operation exception

Floating-point operation exception:

Unimplemented operation exception

Invalid operation exception

Inexact operation exception

Overflow exception

CEIL.W.fmt

Floating-point Ceiling to Single Fixed-point Format

(2/2)

Caution The unimplemented operation exception occurs in the following cases.

- **If overflow occurs when the format is converted into a fixed-point format**
- **If the source operand is infinity**
- **If the source operand is NaN**

Specifically, the exception occurs if the value stored in floating-point register *fd* is outside the range of 2^{31} – 1 (0x7FFF FFFF) to – 2^{31} (0x8000 0000).

CFC1

Move Control Word from FPU (Coprocessor 1)

31 26	25 21	20 16	15 11	10 0
COP1 010001	CF 00010	rt	fs	0 00000000000

Format:

CFC1 rt, fs **MIPS I**

Purpose:

Copies a word from a FPU control register to a general-purpose register.

Description:

This instruction loads the contents of floating-point control register *fs* to general-purpose register *rt* of the CPU. This instruction is defined only if *fs* is 0, 25, 26, 28, or 31. Otherwise, the result will be undefined.

Remark Of the floating-point control registers, FCR25, FCR26, and FCR28 are provided in the VR5500. Therefore, these registers cannot be specified as *fs* with the MIPS I, II, III, and IV instruction set architectures.

Operation:

```
32      T:      temp ← FCR[fs]
        T + 1:  GPR[rt] ← temp


64      T:      temp ← FCR[fs]
        T + 1:  GPR[rt] ← $(temp_{31})^{32}$ || temp
```

Exceptions:

Coprocessor unusable exception

CTC1

Move Control Word to FPU (Coprocessor 1)

31 26	25 21	20 16	15 11	10 0
COP1 010001	CT 00110	rt	fs	0 00000000000

Format:

CTC1 rt, fs **MIPS I**

Purpose:

Copies a word from a general-purpose register to a FPU control register.

Description:

This instruction loads the contents of general-purpose register *rt* of the CPU to floating-point control register *fs*. This instruction is defined only if *fs* is 0, 25, 26, 28, or 31. Otherwise, the result will be undefined.

If the cause bit of this register and corresponding enable bit are set by writing data to the Control/Status register (FCR31), a floating-point operation exception occurs. Write data to the register before the exception occurs.

Remark Of the floating-point control registers, FCR25, FCR26, and FCR28 are provided in the VR5500. Therefore, these registers cannot be specified as *fs* with the MIPS I, II, III, and IV instruction set architectures.

Operation:

```
32    T:       temp ← GPR[rt]
      T + 1:   FCR[fs] ← temp


64    T:       temp ← GPR[rt]31..0
      T + 1:   FCR[fs] ← temp
```

Exceptions:

Coprocessor unusable exception

Reserved instruction exception

Floating-point operation exception

Floating-point operation exception:

Unimplemented operation exception

Invalid operation exception

Inexact operation exception

Division-by-zero exception

Overflow exception

Underflow exception

CVT.D.fmt

Floating-point Convert to Double Floating-point Format

(1/2)

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	CVT. D 100001

Format:

CVT.D.S fd, fs **MIPS I**
CVT.D.W fd, fs **MIPS III**
CVT.D.L fd, fs

Purpose:

Converts a floating-point value or fixed-point value into a double-precision floating-point value.

Description:

This instruction arithmetically converts the contents of floating-point register *fs* into a double-precision floating-point format in accordance with the current rounding mode, and stores the result in floating-point register *fd*. The source operand is processed as floating-point format *fmt*.

This instruction is valid only when converting from a single-precision floating-point format or from a 32-bit or 64-bit fixed-point format.

This conversion operation is executed accurately, without the accuracy affected, in the single-precision floating-point format and 32-bit fixed-point format.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both 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 operation exception

Floating-point operation exception:

Unimplemented operation exception
Invalid operation exception
Inexact operation exception

CVT.D.fmt

Floating-point Convert to Double Floating-point Format

(2/2)

Caution The unimplemented operation exception occurs in the following cases.

- **If overflow occurs when the format is converted into a fixed-point format**
- **If the source operand is infinity**
- **If the source operand is NaN**

Specifically, the unimplemented operation exception occurs if conversion is executed when the format of the source operand is outside the range of 2^{55} – 1 (0x007F FFFF FFFF FFFF) to – 2^{55} (0xFF80 0000 0000 0000).

CVT.L.fmt

Floating-point Convert to Long Fixed-point Format

(1/2)

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	CVT. L 100101

Format:

CVT.L.S fd, fs

CVT.L.D fd, fs

MIPS III

Purpose:

Converts a floating-point value into a 64-bit fixed-point value.

Description:

This instruction arithmetically converts the contents of floating-point register *fs* into a 64-bit floating-point format in accordance with the current rounding mode, and stores the result in floating-point register *fd*. The source operand is processed as floating-point format *fmt*.

This instruction is valid only when converting from single-/double-precision floating-point formats.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

If the source operand is infinity or NaN, and if the result of rounding is outside the range of $2^{63} – 1$ to $– 2^{63}$, the flag bits of FCR31 and FCR26 are set to indicate an invalid operation. If an invalid operation exception is not enabled, the exception does not occur, and $2^{63} – 1$ is returned.

This operation is defined 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.

Operation:

```
64      T:      StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt), fmt, L))
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Coprocessor unusable exception

Floating-point operation exception

Reserved instruction exception (32-bit user/supervisor mode)

Floating-point operation exception:

Unimplemented operation exception

Invalid operation exception

Inexact operation exception

Overflow exception

CVT.L.fmt

Floating-point Convert to Long Fixed-point Format

(2/2)

Caution The unimplemented operation exception occurs in the following cases.

- **If overflow occurs when the format is converted into a fixed-point format**
- **If the source operand is infinity**
- **If the source operand is NaN**

Specifically, the exception occurs if the value stored in floating-point register *fd* is outside the range of 2^{53} – 1 (0x001F FFFF FFFF FFFF) to –2^{53} (0xFFE0 0000 0000 0000).

CVT.S.fmt

Floating-point Convert to Single Floating-point Format

(1/2)

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	CVT. S 100000

Format:

CVT.S.D fd, fs **MIPS I**
CVT.S.W fd, fs **MIPS III**
CVT.S.L fd, fs

Purpose:

Converts a floating-point value or fixed-point value into a single-precision floating-point value.

Description:

This instruction arithmetically converts the contents of floating-point register *fs* into a single-precision floating-point format in accordance with the current rounding mode, and stores the result in floating-point register *fd*. The source operand is processed as floating-point format *fmt*. The result is rounded in accordance with the current rounding mode.

This instruction is valid only when converting from a double-precision floating-point format or from a 32-bit or 64-bit fixed-point format.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

Operation:

```
32, 64    T:    StoreFPR (fd, S, ConvertFmt (ValueFPR (fs, fmt), fmt, S))
```

Exceptions:

Coprocessor unusable exception
Reserved instruction exception
Floating-point operation exception

Floating-point operation exception:

Unimplemented operation exception
Invalid operation exception
Inexact operation exception
Overflow exception
Underflow exception

CVT.S.fmt

Floating-point Convert to Single Floating-point Format

(2/2)

Caution The unimplemented operation exception occurs in the following cases.

- **If overflow occurs when the format is converted into a fixed-point format**
- **If the source operand is infinity**
- **If the source operand is NaN**

Specifically, the unimplemented operation exception occurs if conversion is executed when the format of the source operand is outside the range of 2^{55} – 1 (0x007F FFFF FFFF FFFF) to – 2^{55} (0xFF80 0000 0000 0000).

CVT.W.fmt

Floating-point Convert to Single Fixed-point Format

(1/2)

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	CVT. W 100100

Format:

CVT.W.S fd, fs **MIPS I**

CVT.W.D fd, fs

Purpose:

Converts a floating-point value into a 32-bit fixed-point value.

Description:

This instruction arithmetically converts the contents of floating-point register *fs* into a 32-bit floating-point format, and stores the result in floating-point register *fd*. The source operand is processed as floating-point format *fmt*.

This instruction is valid only when converting from single-/double-precision floating-point formats.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

If the source operand is infinity or NaN, and if the result of rounding is outside the range of $2^{31} – 1$ to $– 2^{31}$, the flag bits of FCR31 and FCR26 are set to indicate an invalid operation. If an invalid operation exception is not enabled, the exception does not occur, and $2^{31} – 1$ is returned.

Operation:

```
32, 64    T:    StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt), fmt, W))
```

Exceptions:

Coprocessor unusable exception

Reserved instruction exception

Floating-point operation exception

Floating-point operation exception:

Unimplemented operation exception

Invalid operation exception

Inexact operation exception

Overflow exception

CVT.W.fmt

Floating-point Convert to Single Fixed-point Format

(2/2)

Caution The unimplemented operation exception occurs in the following cases.

- **If overflow occurs when the format is converted into a fixed-point format**
- **If the source operand is infinity**
- **If the source operand is NaN**

Specifically, the exception occurs if the value stored in floating-point register *fd* is outside the range of 2^{31} – 1 (0x7FFF FFFF) to – 2^{31} (0x8000 0000).

DIV.fmt

Floating-point Divide

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	ft	fs	fd	DIV 000011

Format:

DIV.S fd, fs, ft **MIPS I**
DIV.D fd, fs, ft

Purpose:

Divides a floating-point value.

Description:

This instruction divides the contents of floating-point register *fs* by the contents of floating-point register *ft*, and stores the result in floating-point register *fd*. The operand is processed as floating-point format *fmt*. The operation is executed as if it were of infinite accuracy, and the result is rounded in accordance with the current rounding mode.

This instruction is valid only in single-/double-precision floating-point formats.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

Operation:

```
32, 64   T:      StoreFPR (fd, fmt, ValueFPR (fs, fmt) / ValueFPR (ft, fmt))
```

Exceptions:

Coprocessor unusable exception
Reserved instruction exception
Floating-point operation exception

Floating-point operation exception:

Unimplemented operation exception
Invalid operation exception
Inexact operation exception
Division-by-zero exception
Overflow exception
Underflow exception

DMFC1

Doubleword Move from FPU (Coprocessor 1)

31 26	25 21	20 16	15 11	10 0
COP1 010001	DMF 00001	rt	fs	0 00000000000

Format:

DMFC1 rt, fs

MIPS III

Purpose:

Copies a doubleword from a floating-point register to a general-purpose register.

Description:

This instruction loads the contents of floating-point general-purpose register *fs* to general-purpose register *rt* of the CPU.

The FR bit of the Status register indicates that all the 32 registers of the processor can be specified or not. If the FR bit is 0 and if the least significant bit of *fs* is 1, this instruction is undefined.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

This operation is defined in 64-bit mode or in 32-bit kernel mode.

Operation:

```
64      T:      if $SR_{26}$ = 1 then
                  data ← FGR [fs]
                else
                if $fs_0$ = 0 then
                  data ← FGR [fs+1] || FGR[fs]
                else
                  data ← $undefined_{64}$
                endif
        T + 1:  GPR [rt] ← data
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Coprocessor unusable exception

Reserved instruction exception (32-bit user/supervisor mode)

DMTC1

Doubleword Move to FPU (Coprocessor 1)

31 26	25 21	20 16	15 11	10 0
COP1 010001	DMT 00101	rt	fs	0 00000000000

Format:

DMTC1 rt, fs

MIPS III

Purpose:

Copies a doubleword from a general-purpose register to a floating-point register.

Description:

This instruction loads the contents of general-purpose register *rt* of the CPU to floating-point general-purpose register *fs*.

The FR bit of the Status register indicates that all the 32 registers of the processor can be specified or not. If the FR bit is 0 and if the least significant bit of *fs* is 1, this instruction is undefined.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

This operation is defined in 64-bit mode or in 32-bit kernel mode.

Operation:

```
64    T:      data ← GPR [rt]
      T + 1:  if $SR_{26}$ = 1 then
                FGR [fs] ← data
              else
              if $fs_0$ = 0 then
                FGR [fs+1] ← $data_{63..32}$
                FGR [fs] ← $data_{31..0}$
              else
                undefined_result
              endif
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Coprocessor unusable exception

Reserved instruction exception (32-bit user/supervisor mode)

FLOOR.L.fmt

Floating-point Floor to Long Fixed-point Format

(1/2)

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	FLOOR. L 001011

Format:

FLOOR.L.S fd, fs
FLOOR.L.D fd, fs

MIPS III

Purpose:

Rounds down a floating-point value to a 64-bit fixed-point value for conversion.

Description:

This instruction arithmetically converts the contents of floating-point register *fs* into a 64-bit floating-point format, and stores the result in floating-point register *fd*. The source operand is processed as floating-point format *fmt*.

The result is rounded toward the direction of $-\infty$ regardless of the current rounding mode.

This instruction is valid only when converting from single-/double-precision floating-point formats.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

If the source operand is infinity or NaN, and if the result of rounding is outside the range of $2^{63} - 1$ to $- 2^{63}$, the flag bits of FCR31 and FCR26 are set to indicate an invalid operation. If an invalid operation exception is not enabled, the exception does not occur, and $2^{63} - 1$ is returned.

This operation is defined 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.

Operation:

```
64      T:      StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt), fmt, L))
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Coprocessor unusable exception
Floating-point operation exception
Reserved instruction exception (32-bit user/supervisor mode)

Floating-point operation exception:

Unimplemented operation exception
Invalid operation exception
Inexact operation exception
Overflow exception

FLOOR.L.fmt

Floating-point Floor to Long Fixed-point Format

(2/2)

Caution The unimplemented operation exception occurs in the following cases.

- **If overflow occurs when the format is converted into a fixed-point format**
- **If the source operand is infinity**
- **If the source operand is NaN**

Specifically, the exception occurs if the value stored in floating-point register *fd* is outside the range of 2^{53} – 1 (0x001F FFFF FFFF FFFF) to – 2^{53} (0xFFE0 0000 0000 0000).

FLOOR.W.fmt

Floating-point Floor to Single Fixed-point Format

(1/2)

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	FLOOR. W 001111

Format:

FLOOR.W.S fd, fs

MIPS II

FLOOR.W.D fd, fs

Purpose:

Rounds down a floating-point value to a 32-bit fixed-point value for conversion.

Description:

This instruction arithmetically converts the contents of floating-point register *fs* into a 32-bit floating-point format, and stores the result in floating-point register *fd*. The source operand is processed as floating-point format *fmt*.

The result is rounded toward the direction of –∞ regardless of the current rounding mode.

This instruction is valid only when converting from single-/double-precision floating-point formats.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

If the source operand is infinity or NaN, and if the result of rounding is outside the range of $2^{31} – 1$ to $– 2^{31}$, the flag bits of FCR31 and FCR26 are set to indicate an invalid operation. If an invalid operation exception is not enabled, the exception does not occur, and $2^{31} – 1$ is returned.

Operation:

```
32, 64    T:        StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt), fmt, W))
```

Exceptions:

Coprocessor unusable exception

Reserved instruction exception

Floating-point operation exception

Floating-point operation exception:

Unimplemented operation exception

Invalid operation exception

Inexact operation exception

Overflow exception

FLOOR.W.fmt

Floating-point Floor to Single Fixed-point Format

(2/2)

Caution The unimplemented operation exception occurs in the following cases.

- **If overflow occurs when the format is converted into a fixed-point format**
- **If the source operand is infinity**
- **If the source operand is NaN**

Specifically, the exception occurs if the value stored in floating-point register *fd* is outside the range of 2^{31} – 1 (0x7FFF FFFF) to – 2^{31} (0x8000 0000).

LDC1

Load Doubleword to FPU (Coprocessor 1)

(1/2)

31 26	25 21	20 16	15 0
LDC1 110101	base	ft	offset

Format:

LDC1 rt, offset (base)

MIPS II

Purpose:

Loads a doubleword from memory to a floating-point register.

Description:

This instruction sign-extends a 16-bit *offset* and adds the result to the contents of general-purpose register *base* to generate a virtual address.

If the FR bit of the Status register is 0, the contents of the doubleword at the memory position specified by the virtual address are loaded to floating-point registers *ft* and *ft* + 1. At this time, the higher 32 bits of the doubleword are stored in the odd-numbered register specified by *ft* + 1, and the lower 32 bits are stored in the even-numbered register specified by *ft*. If the least significant bit of the *ft* field is not 0, the operation is undefined.

If the FR bit is1, the contents of the doubleword at the memory position specified by the virtual address are loaded to floating-point register *ft*.

An address error exception occurs if the lower 3 bits of the address are not 0.

LDC1

Load Doubleword to FPU (Coprocessor 1)

(2/2)

Operation:

```
32    T:    vAddr ← ((offset$_{15}$)$^{16}$ || offset$_{15..0}$) + GPR [base]
            (pAddr, uncached) ← Address Translation (vAddr, DATA)
            data ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
            if SR$_{26}$ = 1 then
              FGR [ft] ← data
            elseif ft$_0$ = 0 then
              FGR [ft+1] ← data$_{63..32}$
              FGR [ft] ← data$_{31..0}$
            else
              undefined_result
            endif


64    T:    vAddr ← ((offset$_{15}$)$^{48}$ || offset$_{15..0}$) + GPR [base]
            (pAddr, uncached) ← Address Translation (vAddr, DATA)
            data ← LoadMemory (uncached, DOUBLEWORD, pAddr, vAddr, DATA)
            if SR$_{26}$ = 1 then
              FGR [ft] ← data
            elseif ft$_0$ = 0 then
              FGR [ft+1] ← data$_{63..32}$
              FGR [ft] ← data$_{31..0}$
            else
              undefined_result
            endif
```

Exceptions:

Coprocessor unusable exception
TLB refill exception
TLB invalid exception
Bus error exception
Address error exception
Reserved instruction exception

LDXC1

Load Doubleword Indexed to FPU (Coprocessor 1)

31 26	25 21	20 16	15 11	10 6	5 0
COP1X 010011	base	index	0 00000	fd	LDXC1 000001

Format:

LDXC1 fd, index (base) **MIPS IV**

Purpose:

Loads a doubleword from memory to a floating-point register (general-purpose register + general-purpose register addressing).

Description:

This instruction adds the contents of CPU general-purpose register *index* and the contents of CPU general-purpose register *base* to generate a virtual address.

If the FR bit of the Status register is 0, the contents of the doubleword at the memory position specified by the virtual address are loaded to floating-point registers *fd* and *fd* + 1. At this time, the higher 32 bits of the doubleword are stored in the odd-numbered register specified by *fd* + 1, and the lower 32 bits are stored in the even-numbered register specified by *fd*. If the least significant bit of the *fd* field is not 0, the operation is undefined.

If the FR bit is1, the contents of the doubleword at the memory position specified by the virtual address are loaded to floating-point register *fd*.

The operation is undefined if bits 63 and 62 of the virtual address are not the same as bits 63 and 62 of general-purpose register *base*.

An address error exception occurs if the lower 3 bits of the virtual address are not 0.

Operation:

```
32, 64   T:   vAddr ← GPR[base]+GPR[index]
              (pAddr, CCA) ← Address Translation (vAddr, DATA)
              data ← LoadMemory (CCA, DOUBLEWORD, pAddr, vAddr, DATA)
              if $SR_{26}$ = 1 then
                FGR[fd] ← data
              elseif $fd_0$ = 0 then
                FGR[fd+1] ← $data_{63..32}$
                FGR[fd] ← $data_{31..0}$
              else
                undefined_result
              endif
```

Exceptions:

Coprocessor unusable exception

TLB refill exception

TLB invalid exception

Address error exception

Reserved instruction exception

LUXC1

Load Doubleword Indexed Unaligned to FPU (Coprocessor 1)

(1/2)

31 26	25 21	20 16	15 11	10 6	5 0
COP1X 010011	base	index	0 00000	fd	LUXC1 000101

Format:

LUXC1 fd, index (base)

MIPS V

Purpose:

Loads a doubleword from memory to a floating-point register (general-purpose register + general-purpose register addressing).

Description:

This instruction adds the contents of CPU general-purpose register *index* and the contents of CPU general-purpose register *base* to generate a virtual address. The lower 3 bits of the virtual address are masked by 0. Therefore, an address error exception does not occur even if the lower 3 bits of the virtual address are not 0.

If the FR bit of the Status register is 0, the contents of the doubleword at the memory position specified by the virtual address are loaded to floating-point registers *fd* and *fd* + 1. At this time, the higher 32 bits of the doubleword are stored in the odd-numbered register specified by *fd* + 1, and the lower 32 bits are stored in the even-numbered register specified by *fd*. If the least significant bit of the *fd* field is not 0, the operation is undefined.

If the FR bit is1, the contents of the doubleword at the memory position specified by the virtual address are loaded to floating-point register *fd*.

The operation is undefined if bits 63 and 62 of the virtual address are not the same as bits 63 and 62 of general-purpose register *base*.

Operation:

```
32, 64   T:   vAddr ← $(GPR[base]+GPR[index])_{63..3} \| 0^3$
              (pAddr, CCA) ← Address Translation (vAddr, DATA)
              data ← LoadMemory (CCA, DOUBLEWORD, pAddr, vAddr, DATA)
              if $SR_{26}$ = 1 then
                FGR[fd] ← data
              elseif $fd_0$ = 0 then
                FGR[fd+1] ← $data_{63..32}$
                FGR[fd] ← $data_{31..0}$
              else
                undefined_result
              endif
```

LUXC1 Load Doubleword Indexed Unaligned to FPU (Coprocessor 1)

(2/2)

Exceptions:

Coprocessor unusable exception
TLB refill exception
TLB invalid exception
TLB modified exception
Bus error exception
Address error exception
Reserved instruction exception

LWC1

Load Word to FPU (Coprocessor 1)

(1/2)

31 26	25 21	20 16	15 0
LWC1 110001	base	ft	offset

Format:

LWC1 ft, offset (base) **MIPS I**

Purpose:

Loads a word from memory to a floating-point register.

Description:

This instruction sign-extends a 16-bit *offset* and adds the result to the contents of general-purpose register *base* to generate a virtual address. The contents of the word at the memory position 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 of the *ft* field is 0, the contents of the word are stored in the lower 32 bits of floating-point register *ft*. If the least significant bit of the *ft* field is 1, the contents of the word are stored in the higher 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 in floating-point register *ft*. The values of the higher 32 bits are undefined.

An address error exception occurs if the lower 2 bits of the address are not 0.

LWC1

Load Word to FPU (Coprocessor 1)

(2/2)

Operation:

```
32    T:    vAddr ← $((\text{offset}_{15})^{16} \mid\mid \text{offset}_{15..0})$ + GPR [base]
            (pAddr, uncached) ← Address Translation (vAddr, DATA)
            data ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
            if $\text{SR}_{26}$ = 1 then
              FGR [ft] ← $\text{undefined}^{32}$ || data
            else
              FGR [ft] ← data
            endif

64    T:    vAddr ← $((\text{offset}_{15})^{48} \mid\mid \text{offset}_{15..0})$ + GPR [base]
            (pAddr, uncached) ← Address Translation (vAddr, DATA)
            data ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
            if $\text{SR}_{26}$ = 1 then
              FGR [ft] ← $\text{undefined}^{32}$ || data
            else
              FGR [ft] ← data
            endif
```

Exceptions:

Coprocessor unusable exception
TLB refill exception
TLB invalid exception
Bus error exception
Address error exception
Reserved instruction exception

LWXC1

Load Word Indexed to FPU (Coprocessor 1)

31 26	25 21	20 16	15 11	10 6	5 0
COP1X 010011	base	index	0 00000	fd	LWXC1 000000

Format:

LWXC1 fd, index (base)

MIPS IV

Purpose:

Loads a word from memory to a floating-point register (general-purpose register + general-purpose register addressing).

Description:

This instruction adds the contents of CPU general-purpose register *index* and the contents of CPU general-purpose register *base* to generate a virtual address. The contents of the word at the memory position specified by the virtual address are loaded to floating-point register *fd*.

If the FR bit of the Status register is 0 and if the least significant bit of the *fd* field is 0, the contents of the word are stored in the lower 32 bits of floating-point register *fd*. If the least significant bit of the *fd* field is 1, the contents of the word are stored in the higher 32 bits of floating-point register *fd* – 1.

If the FR bit is1, the contents of the word at the memory position specified by the virtual address are stored in floating-point register *fd*. The values of the higher 32 bits are undefined.

The operation is undefined if bits 63 and 62 of the virtual address are not the same as bits 63 and 62 of general-purpose register *base*.

An address error exception occurs if the lower 2 bits of the virtual address are not 0.

Operation:

```
32, 64   T:   vAddr ← GPR[base] + GPR[index]
              (pAddr, CCA) ← Address Translation (vAddr, DATA)
              data ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
              if SR$_{26}$ = 1 then
                FGR[fd] ← undefined$^{32}$ || data
              elseif fd$_0$ = 0 then
                FGR[fd] ← FGR[fd]$_{63..32}$ || data
              else
                FGR[fd – 1] ← data || FGR[fd – 1]$_{31..0}$
              endif
```

Exceptions:

Coprocessor unusable exception

TLB refill exception

TLB invalid exception

Address error exception

Reserved instruction exception

MADD.fmt

Floating-point Multiply-Add

31 26	25 21	20 16	15 11	10 6	5 3	2 0
COP1X 010011	fr	ft	fs	fd	MADD 100	fmt

Format:

MADD.S fd, fr, fs, ft **MIPS IV**

MADD.D fd, fr, fs, ft

Purpose:

Combines multiplication and addition of floating-point values for execution.

Description:

This instruction multiplies the contents of floating-point register *fs* by the contents of floating-point register *ft*, adds the contents of floating-point register *fr* to the result, and stores the result of the addition in floating-point register *fd*. The operation is executed as if it were of infinite accuracy, and the result is rounded in accordance with the current rounding mode. The operand is processed as floating-point format *fmt*.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

If the condition of an exception is detected but the exception does not occur, the cause bit and flag bit of a floating-point control register are ORed and the result is written to the flag bit.

Operation:

32, 64	T:	StoreFPR (fd, fmt, ValueFPR (fr, fmt) + ValueFPR (fs, fmt) * ValueFPR (ft, fmt))

Exceptions:

Coprocessor unusable exception

Reserved instruction exception

Floating-point operation exception

Floating-point operation exceptions:

Unimplemented operation exception

Invalid operation exception

Inexact operation exception

Overflow exception

Underflow exception

Caution If the result of multiplication is a denormalized number, or an underflow or overflow occurs, an unimplemented operation exception actually occurs.

MFC1

Move Word from FPU (Coprocessor 1)

31 26	25 21	20 16	15 11	10 0
COP1 010001	MF 00000	rt	fs	0 00000000000

Format:

MFC1 rt, fs

MIPS I

Purpose:

Copies a word from a FPU (CP1) general-purpose register to a general-purpose register.

Description:

This instruction loads the contents of floating-point general-purpose register *fs* to general-purpose register *rt* of the CPU.

If the FR bit of the Status register is 0 and if the least significant bit of *fs* is 0, the lower 32 bits of floating-point register *fs* are stored in general-purpose register *rt*. If the least significant bit of *fs* is 1, the higher 32 bits of floating-point register *fs* – 1 are stored in general-purpose register *rt*.

If the FR bit is 1, all the 64-bit floating-point registers can be accessed. Therefore, the lower 32 bits of floating-point register fs are stored in 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] ← (data$_{31}$)$^{32}$ || data
```

Exceptions:

Coprocessor unusable exception

MOV.fmt

Floating-point Move

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	MOV 000110

Format:

MOV.S fd, fs **MIPS I**

MOV.D fd, fs

Purpose:

Transfers a floating-point value between floating-point registers.

Description:

This instruction stores the contents of floating-point register *fs* in floating-point register *fd*. The operand is processed as floating-point format *fmt*.

This instruction is non-arithmetically executed and the IEEE754 exception does not occur.

This instruction is valid only in single-/double-precision floating-point formats.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

Operation:

```
32, 64   T:     StoreFPR (fd, fmt, ValueFPR (fs, fmt))
```

Exceptions:

Coprocessor unusable exception

Reserved instruction exception

Floating-point operation exception

Floating-point operation exceptions:

Unimplemented operation exception

MOVF

Move Conditional on FPU False

31 26	25 21	20 18	17	16	15 11	10 6	5 0
SPECIAL 000000	rs	cc	0 0	tf 0	rd	0 00000	MOVCI 000001

Format:

MOVF rd, rs, cc

MIPS IV

Purpose:

Tests a floating-point condition code and conditionally moves the contents of a general-purpose register.

Description:

If the condition code bit (*cc* bit) of the floating-point control register (FCR31 or FCR25) specified by cc is false (0), the contents of CPU general-purpose register *rs* are stored in CPU general-purpose register *rd*. The *cc* bit of FCR31 and FCR25 is set by a floating-point comparison instruction (C.cond.fmt).
tf specifies which is used as the branch condition, True or False. The value of *tf* is fixed for each instruction.

Operation:

```
32, 64    T:      if FPConditionCode(cc) = 0 then
                    GPR[rd] ← GPR[rs]
                  endif
```

Exceptions:

Coprocessor unusable exception
Reserved instruction exception

MOVF.fmt

Floating-point Move Conditional on FPU False

31 26	25 21	20 18	17	16	15 11	10 6	5 0
COP1 010001	fmt	cc	0 0	tf 0	fs	fd	MOVCF 010001

Format:

MOVF.S fd, fs, cc

MOVF.D fd, fs, cc

MIPS IV

Purpose:

Tests a floating-point condition code and conditionally moves a floating-point value.

Description:

If the condition code bit (*cc* bit) of the floating-point control register (FCR31 or FCR25) specified by *cc* is false (0), the contents of floating-point register *fs* are stored in floating-point register *fd*. The source and destination operands are processed as floating-point format *fmt*. The *cc* bit of FCR31 and FCR25 is set by a floating-point comparison instruction (C.cond.fmt).

tf specifies which is used as the branch condition, True or False. The value of *tf* is fixed for each instruction.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

This instruction is non-arithmetically executed and the IEEE754 exception does not occur.

Operation:

```
32, 64   T:   if FPConditionCode(cc) = 0 then
                StoreFPR (fd, fmt, ValueFPR (fs, fmt))
              else
                StoreFPR (fd, fmt, ValueFPR (fd, fmt))
              endif
```

Exceptions:

Coprocessor unusable exception

Reserved instruction exception

Floating-point operation exception

Floating-point operation exceptions:

Unimplemented operation exception

MOVN.fmt

Floating-point Move Conditional on Not Zero

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	rt	fs	fd	MOVN 010011

Format:

MOVN.S fd, fs, rt **MIPS IV**
MOVN.D fd, fs, rt

Purpose:

Tests the value of a general-purpose register and conditionally moves a floating-point value.

Description:

If the contents of CPU general-purpose register *rt* are not 0, this instruction stores the contents of floating-point register *fs* in floating-point register *fd*. The source and destination operands are processed as floating-point format *fmt*.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

This instruction is non-arithmetically executed and the IEEE754 exception does not occur.

Operation:

```
32, 64   T:   if GPR[rt] ≠ 0 then
                StoreFPR (fd, fmt, ValueFPR (fs, fmt))
              else
                StoreFPR (fd, fmt, ValueFPR (fd, fmt))
              endif
```

Exceptions:

Coprocessor unusable exception
Reserved instruction exception
Floating-point operation exception

Floating-point operation exceptions:

Unimplemented operation exception

MOVT

Move Conditional on FPU True

31 26	25 21	20 18	17	16	15 11	10 6	5 0
SPECIAL 000000	rs	cc	0 0	tf 1	rd	0 00000	MOVCI 000001

Format:

MOVT rd, rs, cc

MIPS IV

Purpose:

Tests a floating-point condition code and conditionally moves the contents of a general-purpose register.

Description:

If the condition code bit (*cc* bit) of the floating-point control register (FCR31 or FCR25) specified by *cc* is true (1), the contents of CPU general-purpose register *rs* are stored in CPU general-purpose register *rd*. The *cc* bit of FCR31 and FCR25 is set by a floating-point comparison instruction (C.cond.fmt).

tf specifies which is used as the branch condition, True or False. The value of *tf* is fixed for each instruction.

Operation:

```
32, 64    T:     if FPConditionCode(cc) = 1 then
                   GPR[rd] ← GPR[rs]
                 endif
```

Exceptions:

Coprocessor unusable exception

Reserved instruction exception

MOVT.fmt

Floating-point Move Conditional on FPU True

31 26	25 21	20 18	17	16	15 11	10 6	5 0
COP1 010001	fmt	cc	0 0	tf 1	fs	fd	MOVCF 010001

Format:

MOVT.S fd, fs, cc
MOVT.D fd, fs, cc

MIPS IV

Purpose:

Tests a floating-point condition code and conditionally moves a floating-point value.

Description:

If the condition code bit (*cc* bit) of the floating-point control register (FCR31 or FCR25) specified by *cc* is true (1), the contents of floating-point register *fs* are stored in floating-point register *fd*. The source and destination operands are processed as floating-point format *fmt*. The *cc* bit of FCR31 and FCR25 is set by a floating-point comparison instruction (C.cond.fmt).

tf specifies which is used as the branch condition, True or False. The value of *tf* is fixed for each instruction.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

This instruction is non-arithmetically executed and the IEEE754 exception does not occur.

Operation:

```
32, 64   T:     if FPConditionCode(cc) = 1 then
                  StoreFPR (fd, fmt, ValueFPR (fs, fmt))
                else
                  StoreFPR (fd, fmt, ValueFPR (fd, fmt))
                endif
```

Exceptions:

Coprocessor unusable exception

Reserved instruction exception

Floating-point operation exception

Floating-point operation exceptions:

Unimplemented operation exception

MOVZ.fmt

Floating-point Move Conditional on Zero

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	rt	fs	fd	MOVZ 010010

Format:

MOVZ.S fd, fs, rt **MIPS IV**
MOVZ.D fd, fs, rt

Purpose:

Tests the value of a general-purpose register and conditionally moves a floating-point value.

Description:

If the contents of CPU general-purpose register *rt* are 0, this instruction stores the contents of floating-point register *fs* in floating-point register *fd*. The source and destination operands are processed as floating-point format *fmt*.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

This instruction is non-arithmetically executed and the IEEE754 exception does not occur.

Operation:

```
32, 64   T:   if GPR[rt] = 0 then
                StoreFPR (fd, fmt, ValueFPR (fs, fmt))
              else
                StoreFPR (fd, fmt, ValueFPR (fd, fmt))
              endif
```

Exceptions:

Coprocessor unusable exception
Reserved instruction exception
Floating-point operation exception

Floating-point operation exceptions:

Unimplemented operation exception

MSUB.fmt

Floating-point Multiply-Subtract

31 26	25 21	20 16	15 11	10 6	5 3	2 0
COP1X 010011	fr	ft	fs	fd	MSUB 101	fmt

Format:

MSUB.S fd, fr, fs, ft

MSUB.D fd, fr, fs, ft

MIPS IV

Purpose:

Combines multiplication and subtraction of floating-point values for execution.

Description:

This instruction multiplies the contents of floating-point register *fs* by the contents of floating-point register *ft*, subtracts the contents of floating-point register *fr* from the result, and stores the result of the subtraction in floating-point register *fd*. The operation is executed as if it were of infinite accuracy, and the result is rounded in accordance with the current rounding mode. The operand is processed as floating-point format *fmt*.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

If the condition of an exception is detected but the exception does not occur, the cause bit and flag bit of a floating-point control register are ORed and the result is written to the flag bit.

Operation:

```
32, 64    T:      StoreFPR (fd, fmt, ValueFPR (fs, fmt) * ValueFPR (ft, fmt) – ValueFPR (fr, fmt))
```

Exceptions:

Coprocessor unusable exception

Reserved instruction exception

Floating-point operation exception

Floating-point operation exceptions:

Unimplemented operation exception

Invalid operation exception

Inexact operation exception

Overflow exception

Underflow exception

Caution If the result of multiplication is a denormalized number, or an underflow or overflow occurs, an unimplemented operation exception actually occurs.

MTC1

Move Word to FPU (Coprocessor 1)

31 26	25 21	20 16	15 11	10 0
COP1 010001	MT 00100	rt	fs	0 00000000000

Format:

MTC1 rt, fs

MIPS I

Purpose:

Copies a word from a general-purpose register to an FPU (CP1) general-purpose register.

Description:

This instruction stores the contents of CPU general-purpose register *rt* in floating-point general-purpose register *fs*.

How the floating-point general-purpose register is accessed differs depending on the setting of the FR bit of the Status register.

If the FR bit is 0, all the 32 floating-point general-purpose registers can be accessed. To transfer double-precision data, access an odd register for the higher 32 bits and an even register for the lower 32 bits, depending on the format of the floating-point operation instruction.

If the FR bit is 1, all the 32 floating-point general-purpose registers can be accessed, but the lower 32 bits of the registers are accessed for data.

Operation:

```
32, 64  T:      data ← GPR [rt]$_{31..0}$
        T + 1:  if SR$_{26}$ = 1 then
                  FGR [fs] ← undefined$^{32}$ || data
                else
                  FGR [fs] ← data
                endif
```

Exceptions:

Coprocessor unusable exception

MUL.fmt

Floating-point Multiply

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	ft	fs	fd	MUL 000010

Format:

MUL.S fd, fs, ft
MUL.D fd, fs, ft

MIPS I

Purpose:

Multiplies floating-point values.

Description:

This instruction multiplies the contents of floating-point register *fs* by the contents of floating-point register *ft*, and stores the result in floating-point register *fd*. The operand is processed as floating-point format *fmt*.
This instruction is valid only in single-/double-precision floating-point formats.
If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

Operation:

```
32, 64   T:      StoreFPR (fd, fmt, ValueFPR (fs, fmt)* ValueFPR (ft, fmt))
```

Exceptions:

Coprocessor unusable exception
Reserved instruction exception
Floating-point operation exception

Floating-point operation exceptions:

Unimplemented operation exception
Invalid operation exception
Inexact operation exception
Overflow exception
Underflow exception

NEG.fmt

Floating-point Negate

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	NEG 000111

Format:

NEG.S fd, fs

NEG.D fd, fs

MIPS I

Purpose:

Executes a negation operation of a floating-point value.

Description:

This instruction inverts the sign of the contents of floating-point register *fs* and stores the result in floating-point register *fd*. The operand is processed as floating-point format *fmt*.

The sign is arithmetically inverted. Therefore, an instruction whose operand is NaN is invalid.

This instruction is valid only in single-/double-precision floating-point formats.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

Operation:

```
32, 64   T:      StoreFPR (fd, fmt, Negate (ValueFPR (fs, fmt)))
```

Exceptions:

Coprocessor unusable exception

Reserved instruction exception

Floating-point operation exception

Floating-point operation exceptions:

Unimplemented operation exception

Invalid operation exception

NMADD.fmt

Floating-point Negate Multiply-Add

31 26	25 21	20 16	15 11	10 6	5 3	2 0
COP1X 010011	fr	ft	fs	fd	NMADD 110	fmt

Format:

NMADD.S fd, fr, fs, ft
NMADD.D fd, fr, fs, ft

MIPS IV

Purpose:

Combines multiplication and addition of floating-point values for execution and executes a negation operation on the results.

Description:

This instruction multiplies the contents of floating-point register *fs* by the contents of floating-point register *ft*, inverts the sign of the result, and stores the result in floating-point register *fd*. The operation is executed as if it were of infinite accuracy, and the result is rounded in accordance with the current rounding mode. The operand is processed as floating-point format *fmt*.

The sign is arithmetically inverted. Therefore, an instruction whose operand is NaN is invalid.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

If the condition of an exception is detected but the exception does not occur, the cause bit and flag bit of a floating-point control register are ORed and the result is written to the flag bit.

Operation:

```
32, 64    T:    StoreFPR (fd, fmt, Negate (ValueFPR (fr, fmt) + ValueFPR (fs, fmt) * ValueFPR (ft, fmt)))
```

Exceptions:

Coprocessor unusable exception
Reserved instruction exception
Floating-point operation exception

Floating-point operation exceptions:

Unimplemented operation exception
Invalid operation exception
Inexact operation exception
Overflow exception
Underflow exception

Caution If the result of multiplication is a denormalized number, or an underflow or overflow occurs, an unimplemented operation exception actually occurs.

NMSUB.fmt

Floating-point Negate Multiply-Subtract

31 26	25 21	20 16	15 11	10 6	5 3	2 0
COP1X 010011	fr	ft	fs	fd	NMSUB 111	fmt

Format:

NMSUB.S fd, fr, fs, ft

NMSUB.D fd, fr, fs, ft

MIPS IV

Purpose:

Combines multiplication and subtraction of floating-point values for execution and executes a negation operation on the results.

Description:

This instruction multiplies the contents of floating-point register *fs* by the contents of floating-point register *ft*, subtracts the contents of floating-point register *fr*, inverts the sign of the result, and stores the result in floating-point register *fd*. The operation is executed as if it were of infinite accuracy, and the result is rounded in accordance with the current rounding mode. The operand is processed as floating-point format *fmt*.

The sign is arithmetically inverted. Therefore, an instruction whose operand is NaN is invalid.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

If the condition of an exception is detected but the exception does not occur, the cause bit and flag bit of a floating-point control register are ORed and the result is written to the flag bit.

Operation:

```
32, 64    T:    StoreFPR (fd, fmt, Negate (ValueFPR (fs, fmt) * ValueFPR (ft, fmt) – ValueFPR (fr, fmt)))
```

Exceptions:

Coprocessor unusable exception

Reserved instruction exception

Floating-point operation exception

Floating-point operation exceptions:

Unimplemented operation exception

Invalid operation exception

Inexact operation exception

Overflow exception

Underflow exception

Caution If the result of multiplication is a denormalized number, or an underflow or overflow occurs, an unimplemented operation exception actually occurs.

PREFX

Prefetch Indexed

(1/2)

31 26	25 21	20 16	15 11	10 6	5 0
COP1X 010011	base	index	hint	0 00000	PREFX 001111

Format:

PREFX hint, index (base)

MIPS IV

Purpose:

Prefetches data from memory (general-purpose register + general-purpose register addressing).

Description:

This instruction adds the contents of CPU general-purpose register *base* and the contents of CPU general-purpose register *index* to generate a virtual address. It then loads the contents at the specified address position to the data cache.

Bits 15 to 11 (*hint*) of this instruction indicate how the loaded data is used. Note, however, that the contents of *hint* are only used for the processor to judge if prefetching by this instruction is valid or not, and do not affect the actual operation. *hint* indicates the following operations.

hint	Operation	Description
0	Load	Predicts that data is loaded (without modification). Fetches data as if it were loaded.
1 to 31	–	Reserved

This is an auxiliary instruction that improves the program performance. The generated address or the contents of *hint* do not change the status of the processor or system, or the meaning (purpose) of the program.

If this instruction causes a memory access to occur, the access type to be used is determined by the generated address. In other words, the access type used to load/store the generated address is also used for this instruction. However, an access to an uncached area does not occur.

If a translation entry to the specified memory position is not in the TLB, data cannot be prefetched from the map area. This is because no translation entry exists in TLB, it means that no access was made to the memory position recently, therefore, no effect can be expected even if data at such a memory position is prefetched.

Exceptions related to addressing do not occur as a result of executing this instruction. If the condition of an exception is detected, it is ignored, but the prefetch is not executed either. However, even if nothing is prefetched, processing that does not appear, such as writing back a dirty cache line, may be performed.

The operation is undefined if bits 63 and 62 of the virtual address are not the same as bits 63 and 62 of general-purpose register *base*.

PREFX

Prefetch Indexed

(2/2)

Operation:

```
32, 64   T:   vAddr ← GPR[base] + GPR[index]
              (pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
              Prefetch (CCA, pAddr, vAddr, DATA, hint)
```

Exceptions:

Coprocessor unusable exception
Reserved instruction exception

RECIP.fmt

Reciprocal

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	RECIP 010101

Format:

RECIP.S fd, fs

RECIP.D fd, fs

MIPS IV

Purpose:

Calculates the approximate value of the reciprocal of a floating-point value (high speed).

Description:

This instruction calculates the reciprocal of the contents of floating-point register *fs* and stores the result in floating-point register *fd*. The operation is executed as if it were of infinite accuracy, and the result is rounded in accordance with the current rounding mode. The operand is processed as floating-point format *fmt*.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

Operation:

```
32, 64    T:        StoreFPR (fd, fmt, 1.0 / ValueFPR (fs, fmt))
```

Exceptions:

Coprocessor unusable exception
Reserved instruction exception
Floating-point operation exception

Floating-point operation exceptions:

Unimplemented operation exception
Invalid operation exception
Inexact operation exception
Division-by-zero exception
Overflow exception
Underflow exception

ROUND.L.fmt

Floating-point Round to Long Fixed-point Format

(1/2)

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	ROUND. L 001000

Format:

ROUND.L.S fd, fs **MIPS III**

ROUND.L.D fd, fs

Purpose:

Converts a floating-point value into a 64-bit fixed-point value rounded to the closest value.

Description:

This instruction arithmetically converts the contents of floating-point register *fs* into a 64-bit fixed-point format, and stores the result in floating-point register *fd*. The source operand is processed as floating-point format *fmt*.

The result is rounded to the closest value or an even number regardless of the current rounding mode.

This instruction is valid only when converting from single-/double-precision floating-point formats.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

If the source operand is infinity or NaN, and if the result of rounding is outside the range of 2^{63} – 1 to – 2^{63}, the flag bits of FCR31 and FCR26 are set to indicate an invalid operation. If an invalid operation exception is not enabled, the exception does not occur, and 2^{63} – 1 is returned.

This operation is defined 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.

Operation:

```
64      T:      StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt), fmt, L))
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Coprocessor unusable exception

Floating-point operation exception

Reserved instruction exception (32-bit user/supervisor mode)

Floating-point operation exceptions:

Unimplemented operation exception

Invalid operation exception

Inexact operation exception

Overflow exception

ROUND.L.fmt

Floating-point Round to Long Fixed-point Format

(2/2)

Caution The unimplemented operation exception occurs in the following cases.

- **If overflow occurs when the format is converted into a fixed-point format**
- **If the source operand is infinity**
- **If the source operand is NaN**

Specifically, the exception occurs if the value stored in floating-point register *fd* is outside the range of 2^{53} – 1 (0x001F FFFF FFFF FFFF) to –2^{53} (0xFFE0 0000 0000 0000).

ROUND.W.fmt

Floating-point Round to Single Fixed-point Format

(1/2)

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	ROUND. W 001100

Format:

ROUND.W.S fd, fs

ROUND.W.D fd, fs

MIPS II

Purpose:

Converts a floating-point value into a 32-bit fixed-point value rounded to the closest value.

Description:

This instruction arithmetically converts the contents of floating-point register *fs* into a 32-bit fixed-point format, and stores the result in floating-point register *fd*. The source operand is processed as floating-point format *fmt*.

The result is rounded to the closest value or an even number regardless of the current rounding mode.

This instruction is valid only when converting from single-/double-precision floating-point formats.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

If the source operand is infinity or NaN, and if the result of rounding is outside the range of $2^{31} – 1$ to $– 2^{31}$, the flag bits of FCR31 and FCR26 are set to indicate an invalid operation. If an invalid operation exception is not enabled, the exception does not occur, and $2^{31} – 1$ is returned.

Operation:

```
32, 64    T:      StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt), fmt, W))
```

Exceptions:

Coprocessor unusable exception

Reserved instruction exception

Floating-point operation exception

Floating-point operation exceptions:

Unimplemented operation exception

Invalid operation exception

Inexact operation exception

Overflow exception

ROUND.W.fmt

Floating-point Round to Single Fixed-point Format

(2/2)

Caution The unimplemented operation exception occurs in the following cases.

- **If overflow occurs when the format is converted into a fixed-point format**
- **If the source operand is infinity**
- **If the source operand is NaN**

Specifically, the exception occurs if the value stored in floating-point register *fd* is outside the range of 2^{31} – 1 (0x7FFF FFFF) to – 2^{31} (0x8000 0000).

RSQRT.fmt

Reciprocal Square Root

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	RSQRT 010110

Format:

RSQRT.S fd, fs

MIPS IV

RSQRT.D fd, fs

Purpose:

Calculates the approximate value of the reciprocal of the square root of a floating-point value (high speed).

Description:

This instruction calculates the positive arithmetic square root of the contents of floating-point register *fs*, inverts the result, and stores the result in floating-point register *fd*. The operation is executed as if it were of infinite accuracy, and the result is rounded in accordance with the current rounding mode. The operand is processed as floating-point format *fmt*.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

Operation:

```
32, 64    T:      StoreFPR (fd, fmt, 1.0 / SquareRoot (ValueFPR (fs, fmt)))
```

Exceptions:

Coprocessor unusable exception
Reserved instruction exception
Floating-point operation exception

Floating-point operation exceptions:

Unimplemented operation exception
Invalid operation exception
Inexact operation exception
Division-by-zero exception
Overflow exception
Underflow exception

SDC1

Store Doubleword from FPU (Coprocessor 1)

(1/2)

31 26	25 21	20 16	15 0
SDC1 111101	base	ft	offset

Format:

SDC1 ft, offset (base) **MIPS II**

Purpose:

Stores a doubleword from a floating-point register to memory.

Description:

This instruction sign-extends a 16-bit *offset* and adds the result to the contents of general-purpose register *base* to generate a virtual address.

If the FR bit of the Status register is 0, this instruction stores the contents of floating-point registers *ft* and *ft* + 1 in the memory specified by the virtual address as a doubleword. At this time, the contents of the odd-numbered register specified by *ft* + 1 correspond to the higher 32 bits of the doubleword, and the contents of the even-numbered register specified by *ft* correspond to the lower 32 bits.

The operation is undefined if the least significant bit of the *ft* field is not 0.

If the FR bit is 1, the contents of floating-point register *ft* are stored in the memory specified by the virtual address as a doubleword.

If the lower 3 bits of the address are not 0, an address error exception occurs.

SDC1

Store Doubleword from FPU (Coprocessor 1)

(2/2)

Operation:

```
32    T:    vAddr ← ((offset$_{15}$)$^{16}$ || offset$_{15..0}$) + GPR [base]
            (pAddr, uncached) ← Address Translation (vAddr, DATA)
            if SR$_{26}$ = 1 then
              data ← FGR [ft]$_{63..0}$
            elseif ft$_0$ = 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 ← ((offset$_{15}$)$^{48}$ || offset$_{15..0}$) + GPR [base]
            (pAddr, uncached) ← Address Translation (vAddr, DATA)
            if SR$_{26}$ = 1 then
              data ← FGR [ft]$_{63..0}$
            elseif ft$_0$ = 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 exception
TLB refill exception
TLB invalid exception
TLB modified exception
Bus error exception
Address error exception
Reserved instruction exception

SDXC1

Store Doubleword Indexed to FPU (Coprocessor 1)

31 26	25 21	20 16	15 11	10 6	5 0
COP1X 010011	base	index	fs	0 00000	SDXC1 001001

Format:

SDXC1 fs, index (base)

MIPS IV

Purpose:

Stores a doubleword from a floating-point register to memory (general-purpose register + general-purpose register addressing).

Description:

This instruction adds the contents of CPU general-purpose register *index* and the contents of CPU general-purpose register *base* to generate a virtual address.

If the FR bit of the Status register is 0, this instruction stores the contents of floating-point registers *fs* and *fs* + 1 in the memory specified by the virtual address as a doubleword. At this time, the contents of the odd-numbered register specified by *fs* + 1 correspond to the higher 32 bits of the doubleword, and the contents of the even-numbered register specified by *fs* correspond to the lower 32 bits.

The operation is undefined if the least significant bit of the *fs* field is not 0.

If the FR bit is 1, the contents of floating-point register *fs* are stored in the memory specified by the virtual address as a doubleword.

The operation is undefined if bits 63 and 62 of the virtual address are not the same as bits 63 and 62 of general-purpose register *base*.

An address error exception occurs if the lower 3 bits of the virtual address are not 0.

Operation:

```
32, 64   T:   vAddr ← GPR[base] + GPR[index]
              (pAddr, CCA) ← Address Translation (vAddr, DATA)
              if $SR_{26}$ = 1 then
                data ← FGR[fs]
              elseif $fs_0$ = 0 then
                data ← FGR[fs + 1] || FGR[fs]
              else
                data ← $undefined^{64}$
              endif
              StoreMemory (CCA, DOUBLEWORD, data, pAddr, vAddr, DATA)
```

Exceptions:

Coprocessor unusable exception

TLB refill exception

TLB invalid exception

TLB modified exception

Address error exception

Reserved instruction exception

SQRT.fmt

Floating-point Square Root

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	SQRT 000100

Format:

SQRT.S fd, fs **MIPS II**

SQRT.D fd, fs

Purpose:

Calculates the square root of a floating-point value.

Description:

This instruction calculates the positive arithmetic square root of the contents of floating-point register *fs* and stores the result in floating-point register *fd*. The operand is processed as floating-point format *fmt*. The operation is executed as if it were of infinite accuracy, and the result is rounded in accordance with the current rounding mode. The result is –0 if the value of the source operand is –0.

This instruction is valid only in single-/double-precision floating-point formats.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

Operation:

```
32, 64   T:     StoreFPR (fd, fmt, SquareRoot (ValueFPR (fs, fmt)))
```

Exceptions:

Coprocessor unusable exception

Reserved instruction exception

Floating-point operation exception

Floating-point operation exceptions:

Unimplemented operation exception

Invalid operation exception

Inexact operation exception

SUB.fmt

Floating-point Subtract

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	ft	fs	fd	SUB 000001

Format:

SUB.S fd, fs, ft

MIPS I

SUB.D fd, fs, ft

Purpose:

Subtracts a floating-point value.

Description:

This instruction subtracts the contents of floating-point register *ft* from the contents of floating-point register *fs*, and stores the result in floating-point register *fd*. The operation is executed as if it were of infinite accuracy, and the result is rounded in accordance with the current rounding mode.

This instruction is valid only in single-/double-precision floating-point formats.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both 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 operation exception

Floating-point operation exceptions:

Unimplemented operation exception
Invalid operation exception
Inexact operation exception
Overflow exception
Underflow exception

SUXC1

Store Doubleword Indexed Unaligned to FPU (Coprocessor 1)

(1/2)

31 26	25 21	20 16	15 11	10 6	5 0
COP1X 010011	base	index	fs	0 00000	SUXC1 001101

Format:

SUXC1 fs, index (base)

MIPS V

Purpose:

Stores a doubleword from a floating-point register to memory (general-purpose register + general-purpose register addressing).

Description:

This instruction adds the contents of CPU general-purpose register *index* and the contents of CPU general-purpose register *base* to generate a virtual address. The lower 3 bits of the virtual address are masked by 0. Therefore, an address error exception does not occur even if the lower 3 bits of the virtual address are not 0.

If the FR bit of the Status register is 0, this instruction stores the contents of floating-point registers *fs* and *fs* + 1 in the memory specified by the virtual address as a doubleword. At this time, the contents of the odd-numbered register specified by *fs* + 1 correspond to the higher 32 bits of the doubleword, and the contents of the even-numbered register specified by *fs* correspond to the lower 32 bits.

The operation is undefined if the least significant bit of the *fs* field is not 0.

The operation is undefined if bits 63 and 62 of the virtual address are not the same as bits 63 and 62 of general-purpose register *base*.

Operation:

```
32, 64   T:   vAddr ← $(GPR[base] + GPR[index])_{63..3} \| 0^3$
              (pAddr, CCA) ← Address Translation (vAddr, DATA)
              if $SR_{26}$ = 1 then
                data ← FGR[fs]
              elseif $fs_0$ = 0 then
                data ← FGR[fs + 1] || FGR[fs]
              else
                data ← $undefined^{64}$
              endif
              StoreMemory (CCA, DOUBLEWORD, data, pAddr, vAddr, DATA)
```

SUXC1 Store Doubleword Indexed Unaligned to FPU (Coprocessor 1)

(2/2)

Exceptions:

Coprocessor unusable exception
TLB refill exception
TLB invalid exception
TLB modified exception
Bus error exception
Address error exception
Reserved instruction exception

SWC1

Store Word from FPU (Coprocessor 1)

31 … 26	25 … 21	20 … 16	15 … 0
SWC1 111001	base	ft	offset

Format:

SWC1 ft, offset (base)

MIPS I

Purpose:

Stores a word from a floating-point register to memory.

Description:

This instruction sign-extends a 16-bit *offset* and adds the result to the contents of general-purpose register *base* to generate a virtual address. The contents of floating-point general-purpose register *ft* are stored in the memory at the specified address.

If the FR bit of the Status register is 0 and if the least significant bit of the *ft* field is 0, the contents of the lower 32 bits of floating-point register *ft* are stored. If the least significant bit of the *ft* field is 1, the contents of the higher 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 lower 32 bits of the *ft* field are stored.

If the lower 2 bits of the address are not 0, an address error exception occurs.

Operation:

```
32    T:    vAddr ← ((offset$_{15}$)$^{16}$ || offset$_{15..0}$) + GPR [base]
            (pAddr, uncached) ← Address Translation (vAddr, DATA)
            data ← FGR [ft]$_{31..0}$
            StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)


64    T:    vAddr ← ((offset$_{15}$)$^{48}$ || offset$_{15..0}$) + GPR [base]
            (pAddr, uncached) ← Address Translation (vAddr, DATA)
            data ← FGR [ft]$_{31..0}$
            StoreMemory (uncached, WORD, data, pAddr, vAddr, DATA)
```

Exceptions:

Coprocessor unusable exception
TLB refill exception
TLB invalid exception
TLB modified exception
Bus error exception
Address error exception
Reserved instruction exception

SWXC1

Store Word Indexed from FPU (Coprocessor 1)

31 26	25 21	20 16	15 11	10 6	5 0
COP1X 010011	base	index	fs	0 00000	SWXC1 001000

Format:

SWXC1 fs, index (base) **MIPS IV**

Purpose:

Stores a word from a floating-point register to memory (general-purpose register + general-purpose register addressing).

Description:

This instruction adds the contents of CPU general-purpose register *index* and the contents of CPU general-purpose register *base* to generate a virtual address. The contents of floating-point register *fs* are stored in the memory specified by the virtual address.

If the FR bit of the Status register is 0 and if the least significant bit of the *fs* field is 0, the contents of the lower 32 bits of floating-point register *fs* are stored. If the least significant bit of the *fs* field is 1, the contents of the higher 32 bits of floating-point register *fs* – 1 are stored.

If the FR bit is 1, the contents of floating-point register *fs* are stored in the memory specified by the virtual address.

The operation is undefined if bits 63 and 62 of the virtual address are not the same as bits 63 and 62 of general-purpose register *base*.

If the lower 2 bits of the virtual address are not 0, an address error exception occurs.

Operation:

```
32, 64   T:   vAddr ← GPR[base] + GPR[index]
              (pAddr, CCA) ← Address Translation (vAddr, DATA)
              if $SR_{26}$ = 1 then
                   data ← $data_{63..32}$ || $FGR[fs]_{31..0}$
              elseif $fs_0$ = 0 then
                   data ← $data_{63..32}$ || $FGR[fd]_{31..0}$
              else
                   data ← $FGR[fd–1]_{63..32}$ || $data_{31..0}$
              endif
              StoreMemory (CCA, WORD, data, pAddr, vAddr, DATA)
```

Exceptions:

Coprocessor unusable exception

TLB refill exception

TLB invalid exception

TLB modified exception

Address error exception

Reserved instruction exception

TRUNC.L.fmt

Floating-point Truncate to Long Fixed-point Format

(1/2)

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	TRUNC. L 001001

Format:

TRUNC.L.S fd, fs **MIPS III**

TRUNC.L.D fd, fs

Purpose:

Converts a floating-point value into a 64-bit fixed-point value rounded to the direction of zero.

Description:

This instruction arithmetically converts the contents of floating-point register *fs* into a 64-bit fixed-point format, and stores the result in floating-point register *fd*. The source operand is processed as floating-point format *fmt*.

The result is rounded toward the direction of zero regardless of the current rounding mode.

This instruction is valid only when converting from single-/double-precision floating-point formats.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

If the source operand is infinity or NaN, and if the result of rounding is outside the range of $2^{63} – 1$ to $– 2^{63}$, the flag bits of FCR31 and FCR26 are set to indicate an invalid operation. If an invalid operation exception is not enabled, the exception does not occur, and $2^{63} – 1$ is returned.

This operation is defined 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.

Operation:

```
64      T:      StoreFPR (fd, L, ConvertFmt (ValueFPR (fs, fmt), fmt, L))
```

Remark The operation is the same in the 32-bit kernel mode.

Exceptions:

Coprocessor unusable exception

Floating-point operation exception

Reserved instruction exception (32-bit user/supervisor mode)

Floating-point operation exceptions:

Unimplemented operation exception

Invalid operation exception

Inexact operation exception

Overflow exception

TRUNC.L.fmt

Floating-point Trancate to Long Fixed-point Format

(2/2)

Caution The unimplemented operation exception occurs in the following cases.

- **If overflow occurs when the format is converted into a fixed-point format**
- **If the source operand is infinity**
- **If the source operand is NaN**

Specifically, the exception occurs if the value stored in floating-point register *fd* is outside the range of 2^{53} – 1 (0x001F FFFF FFFF FFFF) to $–2^{53}$ (0xFFE0 0000 0000 0000).

TRUNC.W.fmt

Floating-point Truncate to Single Fixed-point Format

(1/2)

31 26	25 21	20 16	15 11	10 6	5 0
COP1 010001	fmt	0 00000	fs	fd	TRUNC. W 001101

Format:

TRUNC.W.S fd, fs **MIPS II**

TRUNC.W.D fd, fs

Purpose:

Converts a floating-point value into a 32-bit fixed-point value rounded to the direction of zero.

Description:

This instruction arithmetically converts the contents of floating-point register *fs* into a 32-bit fixed-point format, and stores the result in floating-point register *fd*. The source operand is processed as floating-point format *fmt*.

The result is rounded toward the direction of zero regardless of the current rounding mode.

This instruction is valid only when converting from single-/double-precision floating-point formats.

If the FR bit of the Status register is 0, only an even register number can be specified because a pair of even and odd numbers adjoining each other is used as the register number of a floating-point register. If an odd number is specified, the operation is undefined. If the FR bit of the Status register is 1, both odd and even register numbers are valid.

If the source operand is infinity or NaN, and if the result of rounding is outside the range of $2^{31} – 1$ to $– 2^{31}$, the flag bits of FCR31 and FCR26 are set to indicate an invalid operation. If an invalid operation exception is not enabled, the exception does not occur, and $2^{31} – 1$ is returned.

Operation:

```
32, 64    T:     StoreFPR (fd, W, ConvertFmt (ValueFPR (fs, fmt), fmt, W))
```

Exceptions:

Coprocessor unusable exception

Floating-point operation exception

Floating-point operation exceptions:

Unimplemented operation exception

Invalid operation exception

Inexact operation exception

Overflow exception

TRUNC.W.fmt

Floating-point Truncate to Single Fixed-point Format

(2/2)

Caution The unimplemented operation exception occurs in the following cases.

- **If overflow occurs when the format is converted into a fixed-point format**
- **If the source operand is infinity**
- **If the source operand is NaN**

Specifically, the exception occurs if the value stored in floating-point register *fd* is outside the range of 2^{31} – 1 (0x7FFF FFFF) to – 2^{31} (0x8000 0000).

18.5 FPU Instruction Opcode Bit Encoding

Figure 18-3 lists the VR5500 instruction opcode bit encoding.

Figure 18-3. FPU Instruction Opcode Bit Encoding (1/2)

28...26 **Opcode**

31...29	0	1	2	3	4	5	6	7
0	SPECIAL							
1								
2		COP1		COP1X				
3								
4								
5								
6		LWC1				LDC1		
7		SWC1				SDC1		

23...21 **sub**

25…24	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

18...16 **br**

20...19	0	1	2	3	4	5	6	7
0	BCF	BCT	BCFL	BCTL	*	*	*	*
1	*	*	*	*	*	*	*	*
2	*	*	*	*	*	*	*	*
3	*	*	*	*	*	*	*	*

2...0 **SPECIAL Function**

5...3	0	1	2	3	4	5	6	7
0	*	MOVF/MOVT	*	*	*	*	*	*
1	*	*	*	*	*	*	*	*
2	*	*	*	*	*	*	*	*
3	*	*	*	*	*	*	*	*
4	*	*	*	*	*	*	*	*
5	*	*	*	*	*	*	*	*
6	*	*	*	*	*	*	*	*
7	*	*	*	*	*	*	*	*

Figure 18-3. FPU Instruction Opcode Bit Encoding (2/2)

2...0 **COP1 Function**

5...3	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	MOVZ	MOVN	γ	RECIP	RSQRT	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

2...0 **COP1X Function**

5...3	0	1	2	3	4	5	6	7
0	LWXC1	LDXC1	γ	γ	γ	LUXC1	γ	γ
1	SWXC1	SDXC1	γ	γ	γ	SUXC1	γ	PREFX
2	γ	γ	γ	γ	γ	γ	γ	γ
3	γ	γ	γ	γ	γ	γ	γ	γ
4	MADD.S	MADD.D	γ	γ	γ	γ	γ	γ
5	MSUB.S	MSUB.D	γ	γ	γ	γ	γ	γ
6	NMADD.S	NMADD.D	γ	γ	γ	γ	γ	γ
7	NMSUB.S	NMSUB.D	γ	γ	γ	γ	γ	γ

Remark The meaning of the symbols in the above figures are as follows.

*: Execution of operation codes marked with an asterisk cause reserved instruction exceptions. They are reserved for future versions of the architecture.

γ: Execution of operation codes marked with a gamma cause an unimplemented operation instruction exception. They are reserved for future versions of the architecture.

η: If the operation code marked with an eta is executed, the result is valid only when the MIPS III instruction set can be used. If the operation is executed when the instruction set cannot be used (32-bit user/supervisor mode), an unimplemented operation exception occurs.

CHAPTER 19 INSTRUCTION HAZARDS

19.1 Overview

Depending on the combination of instructions, the result cannot be provided if two or more system events such as a cache miss, interrupt, and exception, occur during execution. Do not use such instruction combinations.

Many hazards are caused by instructions that change the status or read data in different pipeline stages. These hazards are caused by a combination of instructions; no single instruction causes a hazard. Other hazards occur when an instruction is re-executed after exception processing.

19.2 Details of Instruction Hazard

With the VR5500, the hardware automatically avoids hazards, except those related to instruction fetch.

The following table shows the combinations of operations and sources that cause hazards related to instruction fetch which make the operation unstable and prediction of the result impossible.

Table 19-1. Instruction Hazard of VR5500

Operation	Source	Number of Hazards
Instruction fetch (during address translation)	EntryHi.ASID, TLB	**Note**
Instruction fetch (during address error detection)	Status.KSU, Status.EXL, Status.ERL, Status.KX, Status.SX, Status.UX	**Note**
Instruction decode (during detection of coprocessor and enable privileged instruction)	Status.XX, Status.CU, Status.KSU, Status.EXL, Status.ERL, Status.KX, Status.SX, Status.UX	1

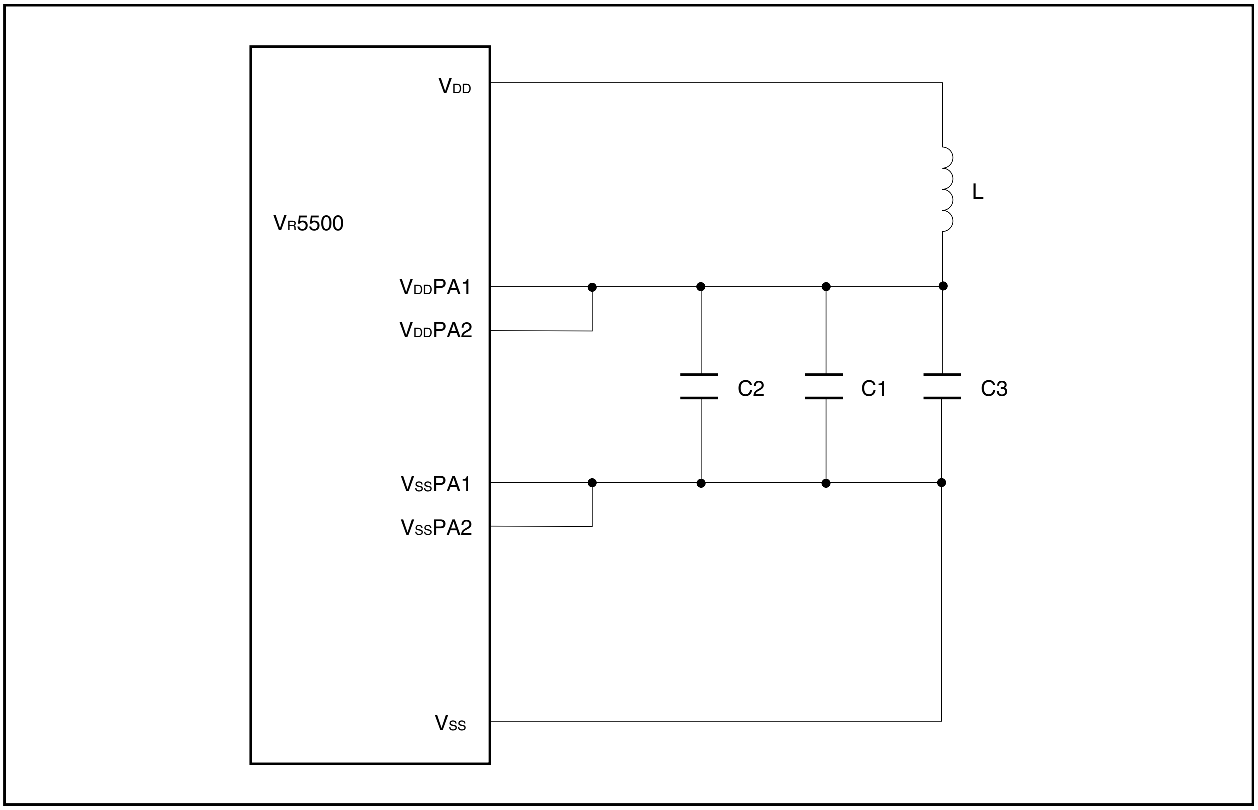
Note If a change is made in the exception handler, it is accurately reflected after the ERET instruction has been executed (compatible with MIPS64).

CHAPTER 20 PLL PASSIVE ELEMENTS

Connect some passive elements externally to the V_{DD}PA1, V_{DD}PA2, V_{SS}PA1 and V_{SS}PA2 pins for proper operation of the V_R5500. Connect the passive elements as close as possible to each pin.

Figure 20-1 shows a connection diagram of the PLL passive elements.

Figure 20-1. Example of Connection of PLL Passive Elements



It is essential to isolate the analog power supply (V_{DD}PA1, V_{DD}PA2) and ground (V_{SS}PA1, V_{SS}PA2) for the PLL circuit from the regular power supply (V_{DD}) and ground (V_{SS}). Examples of each passive element value are as follows.

L = 10 μH	C1 = 0.1 μF	C2 = 100 pF	C3 = 10 μF

Since the optimum values for the filter elements depend on the application and the system noise environment, these values should be considered as starting points for further experimentation within your specific application.

CHAPTER 21 DEBUGGING AND TESTING

This chapter explains the debug and test functions of the VR5500 when a debugging tool is used.

The debug functions explained in this chapter have nothing to do with debugging using the WatchLo and WatchHi registers of the CP0, and realize more sophisticated debugging.

The debugging tool is connected via a test interface.

21.1 Overview

If a debug break occurs, the processor transfers control to the debug exception vector, and enters the debug mode from the normal mode (normal operating status). In the debug mode, the resources of the processor are accessed and controlled internally or externally. Test interfaces (JTAG interface conforming to IEEE1149.1 and debug intereface conforming to the N-Wire specifications) are used to access the processor's resources from an external device.

(1) Internal access

This access is made by the program located at the debug exception vector, using debug instructions.

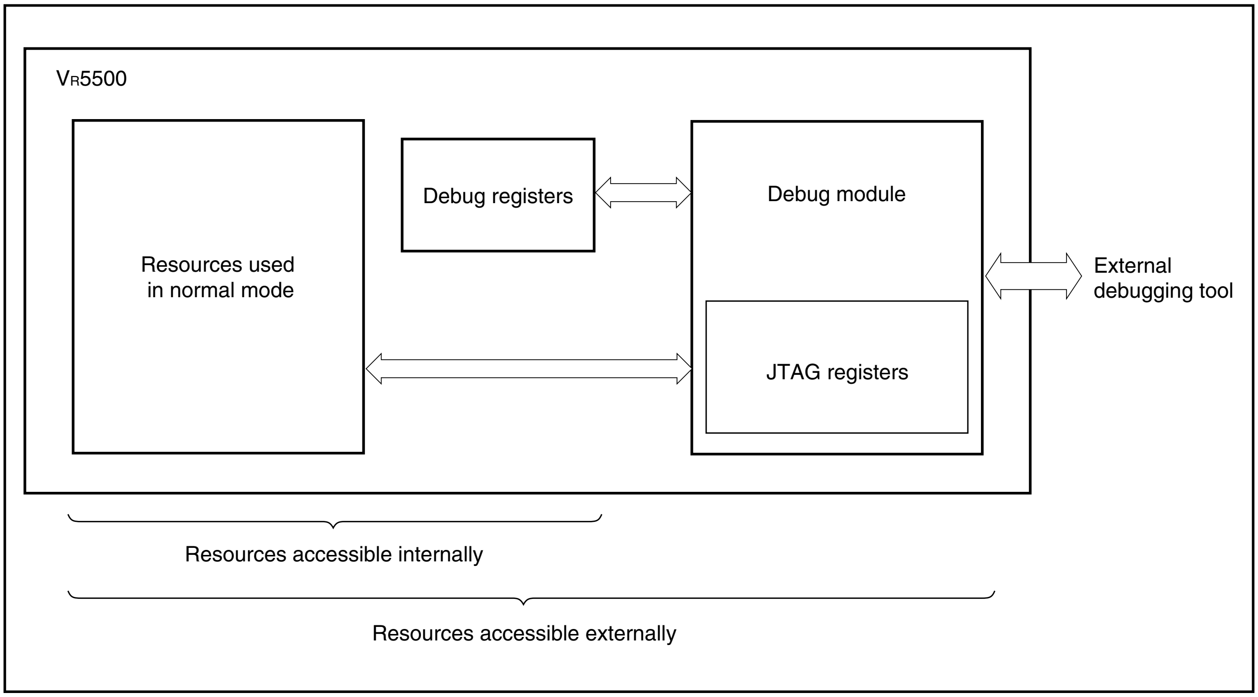
Of the resources of the processor, all the resources used in the normal mode (such as register files, caches, external memory, and external I/O) and debug registers can be accessed.

(2) External access

This access is made by the debugging tool externally connected via a test interface.

All the resources of the processor (such as resources used in the normal mode, the debug registers, and the JTAG registers) can be accessed.

Figure 21-1. Access to Processor Resources in Debug Mode


VR5500
Debug registers
Debug module
Resources used in normal mode
External debugging tool
JTAG registers
Resources accessible internally
Resources accessible externally

The debug registers can be accessed internally or externally only in the debug mode. These registers are used to set breakpoints and their statuses, and change the status of the processor. These registers can be accessed only by using debug instructions. The debug instructions are used to manipulate the debug registers, the resources used in the normal mode, execute debug break, and restore the normal mode.

The externally accessed debug functions have been expanded by the N-Wire specification debug interface. By using this interface, all the resources in the system, including the processor resources, can be monitored from an external debugging tool. For example, data can be loaded to the external memory in the debug mode, then the mode can be changed to the normal mode, and the result of the operation using this data can be monitored. The N-Wire specification also allows an access to the JTAG registers.

Because both the debug registers and JTAG registers can be accessed externally, the scope of control of the processor can be expanded compared with internal access.

Note N of N-Wire indicates the data bus width of the debug interface. Because NTrcData(3:0) specifies the bus width of the VR5500, N = 4.

21.2 Test Interface Signals

Table 21-1. Test Interface Signals

Pin Name	I/O	Function	Recommended Connection When Not Used
JTCK	Input	JTAG clock input Serial clock input signal for JTAG	Pull up
JTMS	Input	JTAG mode selection JTAG test mode selection signal	Pull up
JTDI	Input	JTAG data input Serial data input for JTAG	Pull up
JTDO	Output	JTAG data output Serial data output for JTAG	Leave open
JTRST#	Input	JTAG reset input Signal for initializing JTAG test module (only Ver. 2.0 or later)	Pull down
NTrcData(3:0)	Output	Trace data Data output of test interface	Leave open
NTrcEnd	Output	Trace end Signal indicating delimiting (end) of trace data packet	Leave open
NTrcClk	Output	Trace clock Clock for test interface. Same clock as SysClock is output.	Leave open
RMode#/BKTGIO#	I/O	Reset mode/break trigger output Debug reset input signal while JTRST# signal (ColdReset# signal of Ver. 1.x) is active. Break or trigger I/O signal during normal operation	Pull up

Remark # indicates active low.

(1) JTCK (input)

Input a serial clock for JTAG to this pin. The maximum operating frequency is 33 MHz. This clock can operate asynchronously to the system clock (SysClock).

The JTDI and JTMS signals are sampled at the rising edge of JTCK. The status of the JTDO signal changes at the falling edge of JTCK.

(2) JTMS (input)

Input a command, such as that for selecting mode, for controling the test operation of JTAG. The input command is decoded by the TAP (test access port) controller.

When an external debugging tool is not connected, pull up this signal (this signal is not internally pulled up).

(3) JTDI (input)

Input serial data for scanning to this pin.

When an external debugging tool is not connected, pull up this signal (this signal is not internally pulled up).

(4) JTDO (3-state output)

This pin outputs scanned serial data.

If the data is not correctly scanned, this pin goes into a high-impedance state as defined by IEEE1149.1.

(5) JTRST# (input)

Input a low level to this pin to reset the debug module. This invalidates the debug functions.

- Low level: Initializes the debug module and invalidates the debug functions.
- High level: Clears resetting of the debug module and validates the debug functions.

Remark Because this signal is not provided in VR5500 Ver. 1.x, the function of this signal is implemented by the ColdReset# signal.

(6) NTrcData(3:0) (output)

These pins output a trace packet that is generated as a result of an operation of the processor. It takes one or more cycles to output data of one packet.

(7) NTrcEnd (output)

This signal is asserted when the last data of a trace packet is output to NTrcData(3:0).

(8) NTrcClk (output)

This pin ouptuts a clock of the same frequency as SysClock. This clock can be used when a reference clock is necessary for processing trace information, etc.

(9) RMode#/BKTGIO# (input/output)

This pin functions as the RMode# signal while the JTRST# signal (ColdReset# signal with Ver. 1.x) is active, and as the BKTGIO# signal at other times.

(a) RMode# (input)

Input a signal that sets a debug reset to this pin. This signal is sampled when the JTRST# signal (ColdReset# signal with Ver. 1.x) is deasserted. Setting of a debug reset by the RMode# signal is reflected in a debug register.

- Low level: Executes a debug reset to the processor. Actually, the contents of the reset implemented by asserting the RMode# signal are the same as those implemented by asserting the Reset# signal. The reset bit of the debug register is set to 1.
- High level: Does not execute a debug reset to the processor.

(b) BKTGIO# (input/output)

Input a signal that requests generation of a debug break to this pin when it is set in the input mode.

When this pin is set in the output mode, it outputs a signal that indicates occurrence of a debug trigger or the debug mode status of the processor.

This pin is set in the input mode by default, but the mode can be changed later by setting of debug register.

(i) In input mode

Input a low level to this pin for the duration of only one cycle to generate a debug break.

The processor then enters the debug mode when possible. If the processor is already in the debug mode or if a request for occurrence of a debug break has already been made, inputting a low level to this pin is meaningless.

- Low level: Generates a debug break and places the processor in the debug mode.
- High level: Leaves the processor in the normal mode.

(ii) In output mode

The VR5500 can report detection of a trigger event every 2 SysClock cycles at the fastest. All the trigger events that occur after a trigger was output by the previous BKTGIO# signal are combined into one and output. A trigger event that is not reported when the processor enters the debug mode will not be reported later.

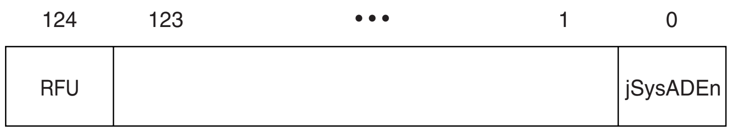
- Low level: Indicates that a trigger event is detected inside the processor if the number of cycles is 1.
  If the number of cycles is 2, this signal indicates that the processor is in the debug mode.
- High level: Indicates that the processor is in the normal mode.

Because the internal circuitry of the VR5500 has superscalar structure and operates at a frequency higher than that of the system interface, a trigger event may occur much earlier than the BKTGIO# signal reports its occurrence.

21.3 Boundary Scan

The Boundary Scan register, one of the JTAG registers, is a 125-bit shift register and holds the status of all the pins of the VR5500. The least significant bit (jSysADEn) of this register is the JTAG output enable bit. When this bit is set to 1, JTAG output is enabled for all the outputs of the processor.

Figure 21-2. Boundary Scan Register



The Boundary Scan register is scanned starting from the least significant bit. The sequence of scanning the register bits is shown below.

Table 21-2. Boundary Scan Sequence

No.	Signal Name	No.	Signal Name	No.	Signal Name	No.	Signal Name	No.	Signal Name
1	jSysADEn	26	SysAD11	51	SysAD55	76	SysID0	101	SysCmd7
2	DrvCon	27	SysAD43	52	SysAD24	77	SysID1	102	SysCmd8
3	RFU (Always input 0.)	28	SysAD12	53	SysAD56	78	SysID2	103	TIntSel
4	SysAD0	29	SysAD44	54	SysAD25	79	RFU (Always input 0.)	104	Int0#
5	SysAD32	30	SysAD13	55	SysAD57	80	BusMode	105	Int1#
6	SysAD1	31	SysAD45	56	SysAD26	81	ValidOut#	106	Int2#
7	SysAD33	32	SysAD14	57	SysAD58	82	ValidIn#	107	Int3#
8	SysAD2	33	SysAD46	58	SysAD27	83	RdRdy#	108	Int4#
9	SysAD34	34	SysAD15	59	SysAD59	84	WrRdy#	109	Int5#
10	SysAD3	35	SysAD47	60	SysAD28	85	ExtRqst#	110	BKTGIO#
11	SysAD35	36	SysAD16	61	SysAD60	86	PReq#	111	RFU (Always input 1.)
12	SysAD4	37	SysAD48	62	SysAD29	87	Release#	112	NMI#
13	SysAD36	38	SysAD17	63	SysAD61	88	Reset#	113	RFU (Always input 1.)
14	SysAD5	39	SysAD49	64	SysAD30	89	ColdReset#	114	BigEndian
15	SysAD37	40	SysAD18	65	SysAD62	90	RFU (Always input 0.)	115	DivMode0
16	SysAD6	41	SysAD50	66	SysAD31	91	O3Return#	116	DivMode1
17	SysAD38	42	SysAD19	67	SysAD63	92	DWBTrans#	117	DivMode2
18	SysAD7	43	SysAD51	68	SysADC0	93	DisDValidO#	118	RFU (Always input 1.)
19	SysAD39	44	SysAD20	69	SysADC4	94	SysCmd0	119	NTrcClk
20	SysAD8	45	SysAD52	70	SysADC1	95	SysCmd1	120	NTrcData0
21	SysAD40	46	SysAD21	71	SysADC5	96	SysCmd2	121	NTrcData1
22	SysAD9	47	SysAD53	72	SysADC2	97	SysCmd3	122	NTrcData2
23	SysAD41	48	SysAD22	73	SysADC6	98	SysCmd4	123	NTrcData3
24	SysAD10	49	SysAD54	74	SysADC3	99	SysCmd5	124	NTrcEnd
25	SysAD42	50	SysAD23	75	SysADC7	100	SysCmd6	125	RFU (Always input 1.)

Remark # indicates active low.

21.4 Connecting Debugging Tool

To use the debug functions of the VR5500, a circuit for connecting an external debugging tool is necessary on the target board.

This section explains the circuit connection when the Kyoto Microcomputer in-circuit emulator PARTNER-ET II is used as the debugging tool.

Caution When evaluating connection of an in-circuit emulator with a trace clock of 100 MHz or more, consult NEC before designing the board. The frequency of the trace clock (NTrcClk) of the VR5500 is the same as that of SysClock.

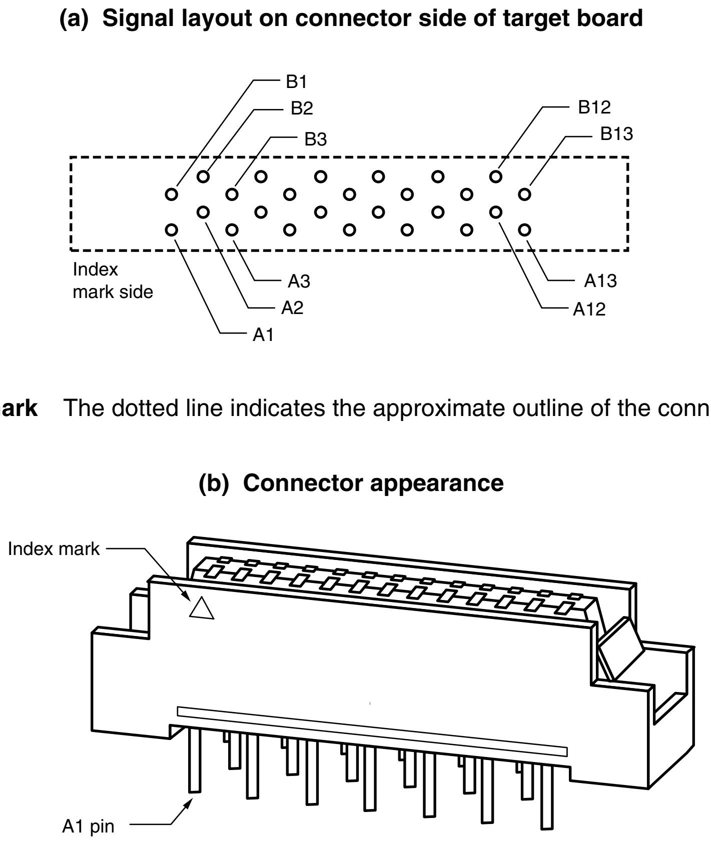
21.4.1 Connecting in-circuit emulator and target board

Use of the following Kell's connectors is recommended when using the PARTNER-ET II.

- 8830E-026-170S (26-pin, straight-angle type)
- 8830E-026-170L (26-pin, light-angle type)

The pins of these recommended connectors are laid out as follows.

Figure 21-3. IE Connection Connector Pin Layout



(a) Signal layout on connector side of target board

Remark The dotted line indicates the approximate outline of the connector.

(b) Connector appearance

Allocate functions to the pins of the recommended connectors as follows when using the PARTNER-ET II.

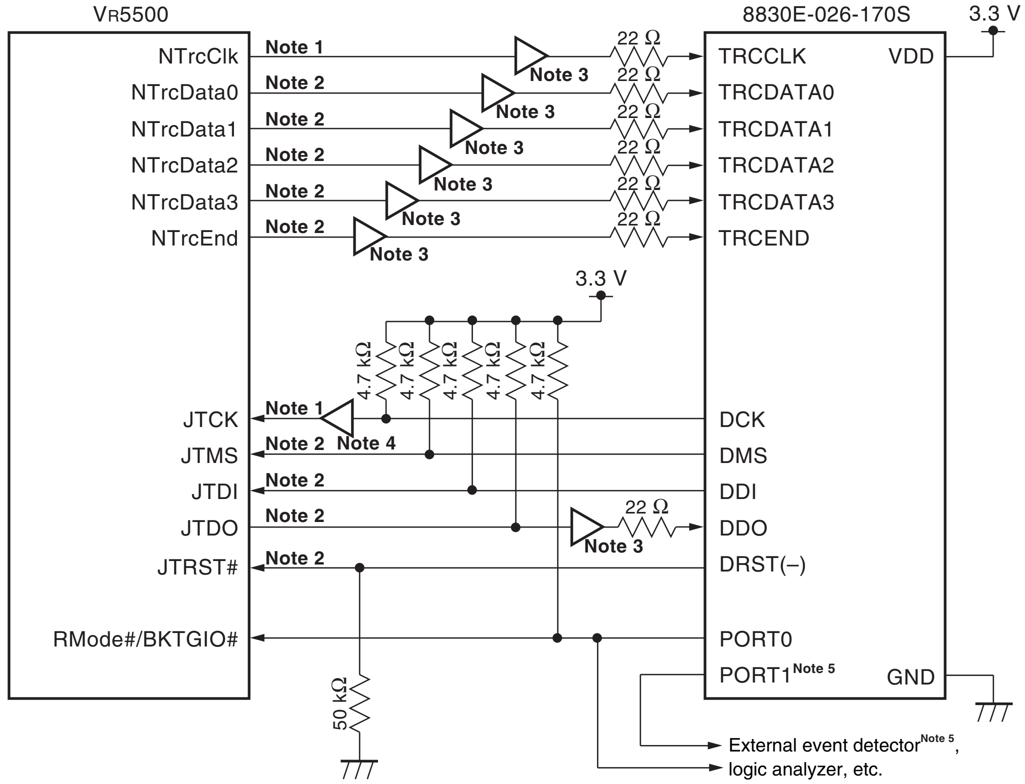
Table 21-3. IE Connector Pin Functions

Pin No.	Signal Name	I/O on IE Connection Side	Function
A1	TRCCLK	O	Trace clock output
A2	TRCDATA0	O	Trace data 0 output
A3	TRCDATA1	O	Trace data 1 output
A4	TRCDATA2	O	Trace data 2 output
A5	TRCDATA3	O	Trace data 3 output
A6	TRCEND	O	Trace data end output
A7	DDI	I	Data input for debug serial interface
A8	DCK	I	Clock input for debug serial interface
A9	DMS	I	Transfer mode select input for debug serial interface
A10	DDO	O	Data output for debug serial interface
A11	DRST(–)	I	Debug control unit reset input (active low)
A12	PORT0	O	General-purpose control signal output 0 (3-state output)
A13	PORT1	O	General-purpose control signal output 1 (3-state output)
B1	GND	–	Ground potential
B2	GND	–	Ground potential
B3	GND	–	Ground potential
B4	GND	–	Ground potential
B5	GND	–	Ground potential
B6	GND	–	Ground potential
B7	GND	–	Ground potential
B8	GND	–	Ground potential
B9	GND	–	Ground potential
B10	GND	–	Ground potential
B11	Reserved	–	Leave this pin open.
B12	Reserved	–	Leave this pin open.
B13	VDD	–	3.3 V (for monitoring target power application)

21.4.2 Connection circuit example

The figure below shows an example of the connection circuit when the Kell's connector 8830E-026-170S is used.

Figure 21-4. Debugging Tool Connection Circuit Example (When Trace Function Is Used)



Notes 1. Keep the clock pattern length as short as possible, and shield the pattern by enclosing it with GND. Keep the pattern length to within 100 mm.

2. Keep the pattern length as short as possible; at least within 100 mm.

3. Use a 3.3 V buffer.

4. Use a clock buffer.

5. When using the BKTGIO function as a debug interrupt input from an external event detector, use PORT1 as a three-state control signal of the detection output signal of the external event detector (control the detection output signal so that it goes into a high-impedance state when PORT1 is high).

Caution **Directly connect the JTDO pin only to the in-circuit emulator. If the JTDO pin is connected as the boundary scan of the next stage, the system may hang up.**

Remark VDD of the connector (B13) is used only to detect power application to the target board. However, it may be used as power source for a signal driver, such as DCK, depending on the tool used. Directly connect it to the power supply of the target board.

APPENDIX A SUB-BLOCK ORDER

A block of data elements (byte, halfword, word, or doubleword) can be extracted from the memory by two methods: sequential ordering and sub-block ordering. This appendix explains these methods, with an emphasis placed on sub-block ordering.

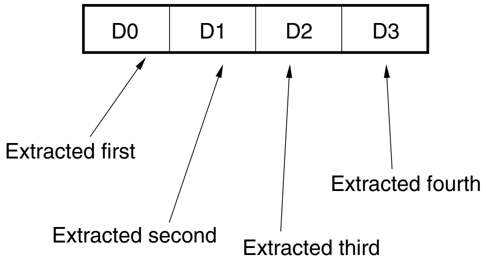
The minimum data element of block transfer of the VR5500 differs depending on the bus width of the system interface. In the 64-bit bus mode, doubleword is the minimum unit. In the 32-bit bus mode, word is the minimum unit. In this appendix, the minimum data element is indicated as D.

(1) Sequential ordering

With sequential ordering, the data elements of a block are extracted serially, i.e., sequentially.

Figure A-1 illustrates the sequential order. In this example, D0 is extracted first, and D3 last.

Figure A-1. Extracting Data Blocks in Sequential Order

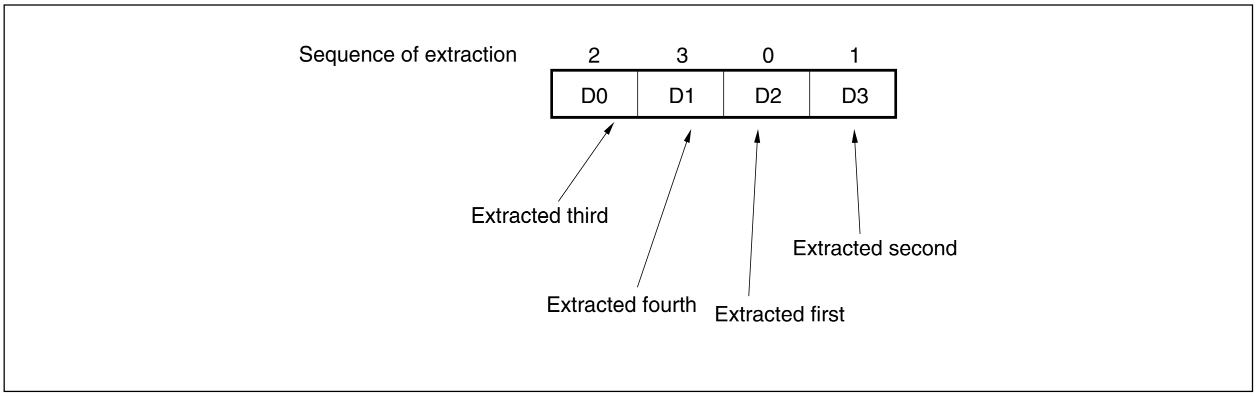


(2) Sub-block ordering

With sub-block ordering, the sequence in which the data elements are to be extracted can be defined.

Figure A-2 shows the sequence in which a data block consisting of four elements is extracted. In this example, D2 is extracted first.

Figure A-2. Extracting Data in Sub-Block Order



The sub-block ordering circuit generates this address by XORing each bit of the start block address with the output of a binary counter that is incremented starting from D0 (00_2) each time a data element has been extracted.

Tables A-1 to A-3 show sub-block ordering in which data is extracted from a block of four elements using this method, where the start block address is 10_2, 11_2, and 01_2, respectively. To generate sub-block ordering, the address of a sub-block (10, 11, or 01) is XORed with the binary count (00_2 to 11_2) of a doubleword. For example, to identify the element that is extracted the third from a data block with a start address of 10_2, XOR address 10_2 with binary count 10_2. The result is 00_2, i.e., D0.

Table A-1. Transfer Sequence by Sub-Block Ordering: Where Start Address Is 10_2

Cycle	Start Block Address	Binary Count	Extracted Element
1	10	00	10
2	10	01	11
3	10	10	00
4	10	11	01

Table A-2. Transfer Sequence by Sub-Block Ordering: Where Start Address Is 11_2

Cycle	Start Block Address	Binary Count	Extracted Element
1	11	00	11
2	11	01	10
3	11	10	01
4	11	11	00

Table A-3. Transfer Sequence by Sub-Block Ordering: Where Start Address Is 01_2

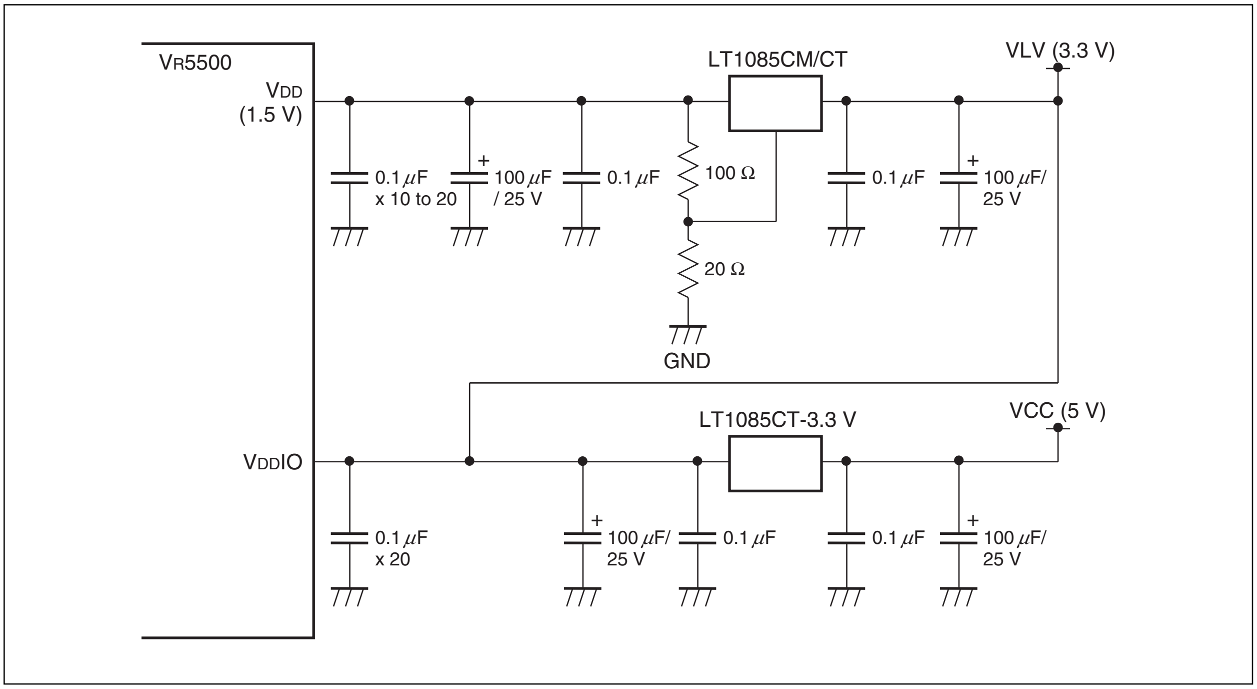
Cycle	Start Block Address	Binary Count	Extracted Element
1	01	00	01
2	01	01	00
3	01	10	11
4	01	11	10

APPENDIX B RECOMMENDED POWER SUPPLY CIRCUIT

Figure B-1 shows an example of the connection of a power supply circuit.

This figure is for reference only. For mass production, thoroughly evaluate and select each element (such as capacitors and regulators).

Figure B-1. Example of Recommended Power Supply Circuit Connection



APPENDIX C RESTRICTIONS ON VR5500

This appendix explains the restrictions on the VR5500 and action to be taken.

C.1 Restrictions on Ver.1.x

C.1.1 During normal operation

(1) Return address in case of address error exception

With VR5500 Ver. 1.x, when the return address (contents of the EPC register) to which execution is to return from an exception handler by executing the ERET instruction is in the address error area, a value different from the contents of the program counter (PC + 0x04 or PC + 0x08) is stored in the EPC register if an interrupt occurs immediately after execution of the ERET instruction.

Therefore, detect an address error and stop program execution in the exception handler.

This restriction does not apply to Ver. 2.0 or later.

(2) Uncached accelerated store operation

With VR5500 Ver. 1.x, a store operation to the uncached accelerated area is not performed correctly if the system interface is in the 32-bit mode.

This restriction does not apply to Ver. 2.0 or later.

(3) Instruction fetch in uncached area

With VR5500 Ver. 1.x, when an instruction is fetched from the uncached area while the system interface is in the 64-bit bus mode, the subsequent instruction may not be correctly executed depending on the combination of the instructions of an even address (lower word) and an odd address (higher word) (mainly combination of jump/branch instructions).

Remark The VR5500 fetches instructions from the uncached area in word (32-bit) units. Therefore, of the data output to the 64-bit bus, the instruction in the lower word (even address) is fetched, and the instruction in the higher word (odd address) is not used.

Therefore, make sure that the instruction at the odd address is not a jump/branch instruction, or that the code of the instruction at the even address is identical to that at the odd address.

This restriction does not apply to Ver. 2.0 or later.

The combinations in which the instruction is not correctly executed are shown below.

(a) Branch instruction at even address with condition satisfied and branch instruction at odd address

The branch destination of the branch instruction at the even address is calculated by using the offset (lower 16 bits) of the branch instruction at the odd address.

There is no problem if the offset of the branch instruction at the odd address is the same as that at the even address.

(b) J or JAL instruction at even address and branch instruction at odd address

The jump destination of the jump instruction at the even address is calculated by using the code (lower 26 bits) of the branch instruction at the odd address.

(c) J or JAL instruction at even address and J or JAL instruction at odd address

The jump destination of the jump instruction at the even address is calculated by using the code (lower 26 bits) of the jump instruction at the odd address.

There is no problem if the code (lower 26 bits) of the jump instruction at the odd address is the same as that at the even address.

(d) Branch instruction at even address with condition satisfied and J or JAL instruction at odd address

The branch destination of the branch instruction at the even address is calculated by using the code (lower 16 bits) of the jump instruction at the odd address.

(4) Operation in low-power mode

VR5500 Ver. 1.x does not stop the internal pipeline clock even when the WAIT instruction is executed (the power consumption is not reduced).

This restriction does not apply to Ver. 2.0 or later.

(5) Clock output on clearing reset

With VR5500 Ver. 1.x, the clock for the serial interface may not be output if a multiplication rate of 2, 3.5, 4, 4.5, or 5.5 is selected when generating an internal clock from an external clock.

Therefore, select a multiplication rate of 2.5, 3, or 5.

This restriction does not apply to Ver. 2.1 or later.

C.1.2 When debug function is used

Caution The operation or result produced by the restrictions described below differs depending on the external debugging tool connected. For details when using the debug function of the VR5500, therefore, consult the manufacturer of the debugging tool to be used.

(1) Trace data when JR/JALR instruction is executed

With VR5500 Ver. 1.x, the contents of the internal TPC packet changes before a TPC packet that indicates the jump destination address of the first jump instruction is output when two or more JR or JALR instructions are executed within 16 PClocks. Consequently, the wrong contents are output as the first TPC packet.

This restriction does not apply to Ver. 2.0 or later.

(2) Trace data when branch instruction is executed

With VR5500 Ver. 1.x, contents that indicate that a branch has been satisfied two times are output as an NSEQ packet if a branch instruction that satisfies a branch and a branch instruction that does not satisfy a branch are executed consecutively.

This restriction does not apply to Ver. 2.0 or later.

(3) Trace data when exception occurs

With VR5500 Ver. 1.x, a TPC packet or NSEQ packet is output instead of an EXP packet, which indicates occurrence of an exception, if an exception occurs as a result of executing the instruction in the branch delay slot.

This restriction does not apply to Ver. 2.0 or later.

(4) Trace data when EXL bit = 1

With VR5500 Ver. 1.x, a packet indicating occurrence of a TLB exception is output instead of a packet indicating occurrence of an ordinary exception if a TLB exception occurs while the EXL bit is set to 1.

This restriction does not apply to Ver. 2.0 or later.

(5) Operation of BKTGIO# signal

With VR5500 Ver. 1.x, an event trigger is output from the BKTGIO# pin if an instruction cache miss conflicts with the match of an instruction address when match of an instruction address is specified as a break trigger.

This restriction does not apply to Ver. 2.0 or later.

(6) Operation when instruction address break occurs

With VR5500 Ver. 1.x, the processor deadlocks if an interrupt or exception conflicts with an instruction address match when an instruction address match is specified as a break trigger.

This restriction does not apply to Ver. 2.0 or later.

(7) Setting of mask register for read access

With VR5500 Ver. 1.x, a break does not occur if the mask register is set taking endian into consideration when a data data trap for read access is set.

When setting a data data trap for read access, therefore, set the mask register without taking endian into consideration.

This restriction does not apply to Ver. 2.0 or later.

(8) Debug reset signal

VR5500 Ver. 1.x does not have a dedicated signal to execute a debug reset and uses the ColdReset# signal instead. However, the ColdReset# signal may be asserted during boundary scan, and therefore, an error may occur during boundary scan.

This restriction does not apply to Ver. 2.0 or later because a dedicated JTRST# signal has been added.

(9) Trace output in debug mode

VR5500 Ver. 1.x ouptuts trace data even in the debug mode.

Therefore, ignore the data output from the NTrcData(3:0) pins from when a debug exception packet is output to when a DRET packet is output.

This restriction does not affect the data output from the NTrcData(3:0) pins in the debug mode because it is ignored by the in-circuit emulator.

This restriction does not apply to Ver. 2.0 or later.

C.2 Restrictions on Ver. 2.0

C.2.1 During normal operation

(1) Clock output on clearing reset

With VR5500 Ver. 2.0, the clock for the serial interface may not be output if a multiplication rate of 2, 3.5, 4, 4.5, or 5.5 is selected when generating an internal clock from an external clock.

Therefore, select a multiplication rate of 2.5, 3, or 5.

This restriction does not apply to Ver. 2.1 or later.

(2) Operation of Release# signal in out-of-order return mode

The Release# signal is not asserted (low level) and the right to control the system interface is not released to the external agent even if the RdRdy# signal is deasserted (high level) in the cycle in which the first request of the successive read requests is issued when VR5500 Ver. 2.0 is set in the pipeline mode in the out-of-order return mode.

This restriction does not apply to Ver. 2.1 or later.

(3) Return address in case of address error exception

With VR5500 Ver. 2.0, if a jump/branch instruction is located two instructions before the boundary with the address error space and if a branch prediction miss (including RAS miss), ERET instruction commitment, exception (except the address error exception mentioned) does not occur (is not committed) between execution of the above jump/branch instruction and occurrence (commitment) of an address error exception due to a specific cause (refer below), the address stored in the BadVAddr register by the processing of the above address error exception is the address at the position (boundary with the address space) two instructions after the jump/branch instruction. However, the correct address is stored in the EPC register.

Therefore, do not locate a jump/branch instruction at the position two instructions before the boundary with the address space.

This restriction applies to the following causes of the address error exception.

- If an attempt is made to fetch an instruction in the kernel address space in the user or supervisor mode
- If an attempt is made to fetch an instruction in the supervisor address space in the user mode
- If an attempt is made to fetch an instruction not located at the word boundary
- If an attempt is made to reference the address error space in the kernel mode

C.2.2 When using debug function

Caution The operation or result produced by the restrictions described below differs depending on the external debugging tool connected. For details when using the debug function of the VR5500, therefore, consult the manufacturer of the debugging tool to be used.

(1) Initialization of debug registers

VR5500 Ver. 2.0 initializes the Monitor Data register in the debug module when the RESET# signal is asserted. However, because the RESET# signal is masked on the emulator side, this restriction has no influence.

(2) Operation when break trigger and exception conflict

With VR5500 Ver. 2.0, if the Data Break Control register in the debug module is set so that only a trigger occurs and if a break trigger and an address error exception or TLB exception occur in the same load/store instruction, the address error exception or TLB exception is indicated by the cause code. However, 0xBFC0 1000 for the debug exception is used as the exception vector address.

(3) Masking NMI request

With VR5500 Ver. 2.0, an NMI exception occurs even if occurrence of NMI is masked by the Debug Mode Control register in the debug module when the NMI request is already held pending internally.

C.3 Restrictions on Ver. 2.1 or Later

C.3.1 During normal operation

(1) Return address in case of address error exception

With VR5500 Ver. 2.1 or later, if a jump/branch instruction is located two instructions before the boundary with the address error space and if a branch prediction miss (including RAS miss), ERET instruction commitment, exception (except the address error exception mentioned) does not occur (is not committed) between execution of the above jump/branch instruction and occurrence (commitment) of an address error exception due to a specific cause (refer below), the address stored in the BadVAddr register by the processing of the above address error exception is the address at the position (boundary with the address space) two instructions after the jump/branch instruction. However, the correct address is stored in the EPC register.

Therefore, do not locate a jump/branch instruction at the position two instructions before the boundary with the address space.

This restriction applies to the following causes of the address error exception.

- If an attempt is made to fetch an instruction in the kernel address space in the user or supervisor mode
- If an attempt is made to fetch an instruction in the supervisor address space in the user mode
- If an attempt is made to fetch an instruction not located at the word boundary
- If an attempt is made to reference the address error space in the kernel mode

C.3.2 When using debug function

Caution The operation or result produced by the restrictions described below differs depending on the external debugging tool connected. For details when using the debug function of the VR5500, therefore, consult the manufacturer of the debugging tool to be used.

(1) Initialization of debug registers

VR5500 Ver. 2.1 or later initializes the Monitor Data register in the debug module when the RESET# signal is asserted.

However, because the RESET# signal is masked on the emulator side, this restriction has no influence.

(2) Operation when break trigger and exception conflict

With VR5500 Ver. 2.1 or later, if the Data Break Control register in the debug module is set so that only a trigger occurs and if a break trigger and an address error exception or TLB exception occur in the same load/store instruction, the address error exception or TLB exception is indicated by the cause code. However, 0xBFC0 1000 for the debug exception is used as the exception vector address.

(3) Masking NMI request

With VR5500 Ver. 2.1 or later, an NMI exception occurs even if occurrence of NMI is masked by the Debug Mode Control register in the debug module when the NMI request is already held pending internally.

[MEMO]

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
Market Communication Dept.
Fax: +49-211-6503-274

South America
NEC do Brasil S.A.
Fax: +55-11-6462-6829

Hong Kong, Philippines, Oceania
NEC Electronics Hong Kong Ltd.
Fax: +852-2886-9022/9044

Korea
NEC Electronics Hong Kong Ltd.
Seoul Branch
Fax: +82-2-528-4411

P.R. China
NEC Electronics Shanghai, Ltd.
Fax: +86-21-6841-1137

Taiwan
NEC Electronics Taiwan Ltd.
Fax: +886-2-2719-5951

Asian Nations except Philippines
NEC Electronics Singapore Pte. Ltd.
Fax: +65-250-3583

Japan
NEC Semiconductor Technical Hotline
Fax: +81- 44-435-9608

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	❏	❏	❏	❏